Николай Прохоренок

Python

САМОЕ НЕОБХОДИМОЕ

Санкт-Петербург
«БХВ-Петербург»
2010

УДК 681.3.068+800.92Python
ББК 32.973.26-018.1
П84

Прохоренок Н. А.

П84 Python. Самое необходимое. — СПб.: БХВ-Петербург, 2011. — 416 с.: ил. + Видеокурс (на DVD)

ISBN 978-5-9775-0614-4

Описан базовый синтаксис языка Python: типы данных, операторы, условия, циклы, регулярные выражения, встроенные функции, объектно-ориентированное программирование, часто используемые модули стандартной библиотеки. Даны основы SQLite, описан интерфейс доступа к базам данных SQLite и MySQL. Рассмотрены работа с изображениями с помощью библиотеки PIL и получение данных из Интернета. Книга содержит более двухсот практических примеров, помогающих начать программировать на языке Python самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прилагаемый DVD содержит листинги описанных в книге примеров и видеоролики.

Для программистов

УДК 681.3.068+800.92Python
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.07.10.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 33,54.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.005770.05.09
от 26.05.2009 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0614-4

Оглавление

Введение

Добро пожаловать в мир Python!

Python — это интерпретируемый объектно-ориентированный язык программирования высокого уровня, предназначенный для самого широкого круга задач. С его помощью можно обрабатывать различные данные, создавать изображения, работать с базами данных, разрабатывать Web-сайты и приложения с графическим интерфейсом. Python является кроссплатформенным языком, позволяющим создавать программы, которые будут работать во всех операционных системах. В этой книге мы рассмотрим базовые возможности Python 2.6 применительно к операционной системе Windows.

Согласно официальной версии название языка произошло вовсе не от змеи. Создатель языка Гвидо ван Россум (Guido van Rossum) назвал свое творение в честь британского комедийного телешоу BBC "Monty Python's Flying Circus". Поэтому более правильно будет "Пайтон". Тем не менее, многие считают, что для русского человека более привычно называть язык "Питон". Так или иначе, в этой книге мы будем придерживаться традиционного написания слова Python на английском языке.

Программа на языке Python представляет собой обычный текстовый файл с расширением py (консольная программа) или pyw (программа с графическим интерфейсом). Все инструкции из этого файла выполняются интерпретатором построчно. Для ускорения работы при первом импорте модуля создается промежуточный байт-код и сохраняется в одноименном файле с расширением pyc. При последующих запусках, если модуль не был изменен, исполняется именно байт-код. Для выполнения низкоуровневых операций и задач, требующих высокой скорости работы, можно написать модуль на языке C, скомпилировать его, а затем подключить к основной программе.

Python является объектно-ориентированным языком. Это означает, что практически все данные являются объектами, даже сами типы данных. В переменной всегда сохраняется только ссылка на объект, а не сам объект. Например, можно создать функцию, сохранить ссылку на нее в переменной, а затем вызвать функцию через эту переменную. Данное обстоятельство делает язык Python идеальным инструментом для создания программ, использующих функции обратного вызова, например при разработке графического интерфейса. Тот факт, что язык является объектно-ориентированным, отнюдь не означает, что ООП-стиль программирования является обязательным. На языке Python можно писать программы как в ООП-стиле, так и в процедурном стиле. Какой стиль использовать — зависит от конкретной ситуации и предпочтений программиста.

Python — самый стильный язык программирования в мире, не допускающий двоякого написания кода. Например, в языке Perl существует зависимость от контекста и множественность синтаксиса. Часто два программиста, пишущих на Perl, просто не понимают код друг друга. В Python код можно написать только одним способом. В нем отсутствуют лишние конструкции. Все программисты должны придерживаться стандарта, описанного в документе **http://python.org/dev/peps/pep-0008/**. Более читаемого кода нет ни в одном другом языке программирования.

Синтаксис языка Python вызывает много нареканий у программистов, знакомых с другими языками. На первый взгляд может показаться, что отсутствие ограничительных символов (фигурных скобок или конструкции `begin...end`) для выделения блоков и обязательная вставка пробелов впереди инструкций может приводить к ошибкам. Однако это только первое и неправильное впечатление. Хороший стиль программирования в любом языке обязывает выделять инструкции внутри блока одинаковым количеством пробелов. В этой ситуации ограничительные символы просто являются лишними. Бытует мнение, что программа будет по-разному смотреться в разных редакторах. Это неверно. Согласно стандарту для выделения блоков необходимо использовать *четыре пробела*. Четыре пробела в любом редакторе будут смотреться одинаково. Если в другом языке вас не приучили к хорошему стилю программирования, то язык Python быстро это исправит. Если количество пробелов внутри блока будет разным, то интерпретатор выведет сообщение о фатальной ошибке, и программа будет остановлена. Таким образом, язык Python приучает программистов писать красивый и понятный код.

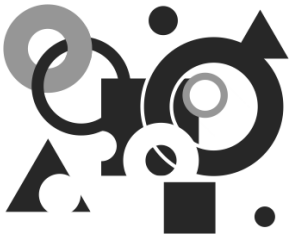
Так как программа на языке Python представляет собой обычный текстовый файл, его можно редактировать с помощью любого текстового редактора, например, с помощью Notepad++. Однако лучше воспользоваться специализированными редакторами, которые не только подсвечивают код, но и выводят различные подсказки и позволяют отладить программу. Таких редакторов очень много, например PyScripter, PythonWin, UliPad, Eclipse + PyDev, Netbeans и др. Полный список редакторов расположен на странице **http://wiki.python.org/moin/PythonEditors**. На всем протяжении этой книги мы будем пользоваться редактором IDLE, который входит в состав стандартной библиотеки Python в Windows. Этот редактор идеально подходит для изучения языка Python.

Ну что, приступим к изучению Python? Язык достоин того, чтобы его знал каждый программист! Но не забывайте, что книги по программированию нужно не только читать, но и выполнять все примеры, а также экспериментировать, изменяя что-нибудь в примерах. Все листинги из книги, а также видеоролики вы найдете на прилагаемом к книге DVD.

Ваши замечания и пожелания можно оставить в гостевой книге или на форуме моего сайта **http://wwwadmin.ru/**. Все замеченные опечатки прошу присылать по адресу **mail@bhv.ru**.

Желаю приятного прочтения и надеюсь, что эта книга станет верным спутником в вашей повседневной жизни.

Глава 1



Первые шаги

1.1. Установка Python

Прежде чем начать изучение основ языка, необходимо установить на компьютер интерпретатор Python.

1. Для загрузки дистрибутива переходим на страницу **http://python.org/download/** и скачиваем файл python-2.6.5.msi. Затем запускаем программу установки с помощью двойного щелчка на значке файла.
2. В открывшемся окне (рис. 1.1) устанавливаем переключатель **Install for all users** (Установить для всех пользователей) и нажимаем кнопку **Next.**
3. На следующем шаге (рис. 1.2) предлагается выбрать каталог для установки. Оставляем каталог по умолчанию (C:\Python26\) и нажимаем кнопку **Next.**
4. В следующем диалоговом окне (рис. 1.3) можно выбрать компоненты, которые следует установить. По умолчанию устанавливаются все компоненты и прописывается ассоциация с файловыми расширениями py, pyw и др. В этом случае запускать программы можно с помощью двойного щелчка на значке файла. Оставляем выбранными все компоненты и нажимаем кнопку **Next.**
5. После завершения установки будет выведено окно, изображенное на рис. 1.4. Нажимаем кнопку **Finish** для выхода из программы установки.

В результате установки исходные файлы интерпретатора будут скопированы в папку C:\Python26. В этой папке расположены два исполняемых файла: python.exe и pythonw.exe. Файл python.exe предназначен для выполнения консольных приложений. Именно эта программа запускается при двойном щелчке на значке файла с расширением py. Файл pythonw.exe используется для запуска оконных приложений. В этом случае окно консоли выводиться не будет. Эта программа запускается при двойном щелчке на значке файла с расширением pyw. В этой книге мы будем запускать программы только с помощью файла python.exe.

Если сделать двойной щелчок на файле python.exe, то запустится интерактивная оболочка в окне консоли (рис. 1.5). Символы >>> в этом окне означают приглашение для ввода выражений на языке Python. Если после этих символов ввести, например, 2 + 2 и нажать клавишу <Enter>, то на следующей строке сразу будет выведен результат выполнения, а затем опять приглашение для ввода нового выражения.

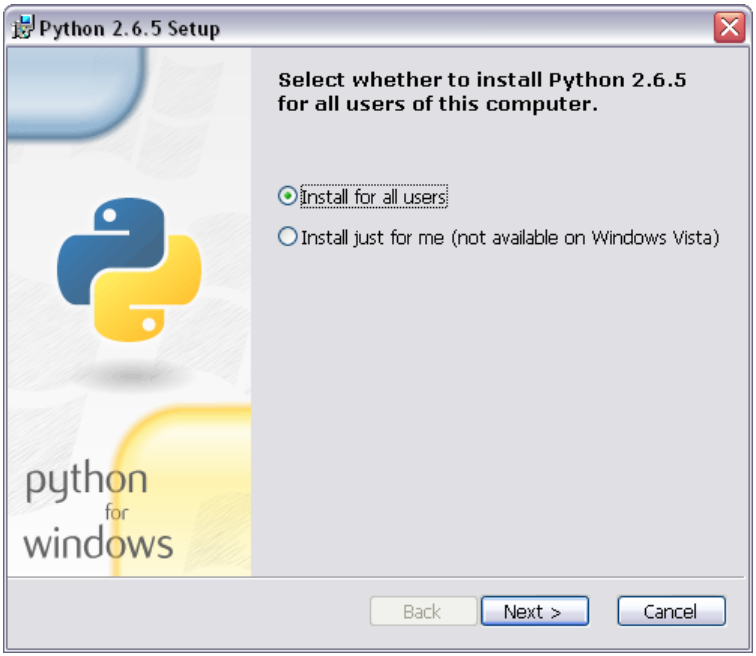


Рис. 1.1. Установка Python. Шаг 1

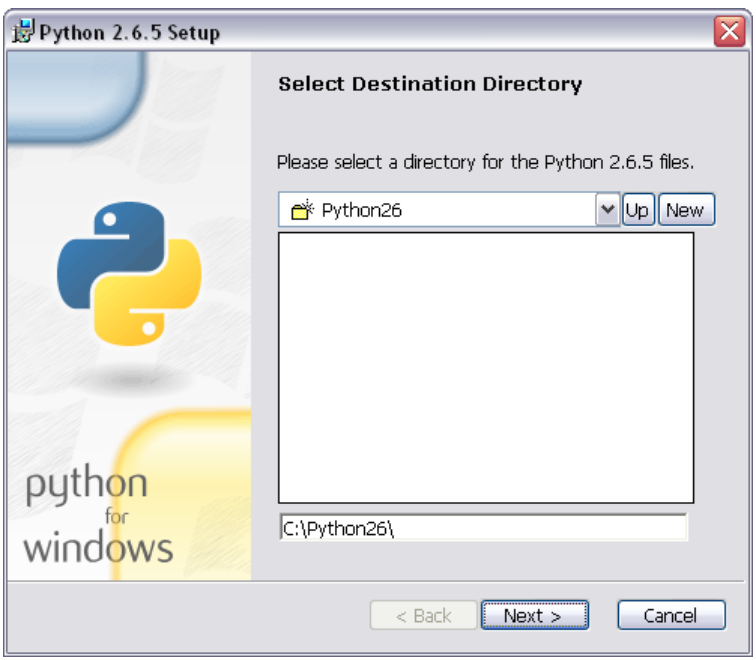


Рис. 1.2. Установка Python. Шаг 2

Рис. 1.3. Установка Python. Шаг 3

Рис. 1.4. Установка Python. Шаг 4

Таким образом, это окно можно использовать в качестве калькулятора, а также для изучения языка. Открыть такое же окно можно с помощью пункта **Python (command line)** в меню **Пуск | Программы | Python 2.6**.

Вместо интерактивной оболочки для изучения языка, а также создания и редактирования файлов с программой, лучше воспользоваться редактором IDLE, который входит в состав установленных компонентов. Для запуска редактора в меню **Пуск | Программы | Python 2.6** выбираем пункт **IDLE (Python GUI)**. В результате откроется окно **Python Shell** (рис. 1.6), которое выполняет все функции интерактивной оболочки, но дополнительно производит подсветку синтаксиса, выводит подсказки и др. Именно этим редактором мы будем пользоваться на протяжении всей книги. Более подробно редактор IDLE мы рассмотрим немного позже.

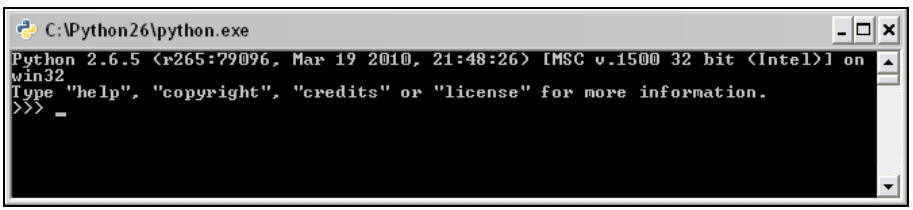


Рис. 1.5. Интерактивная оболочка

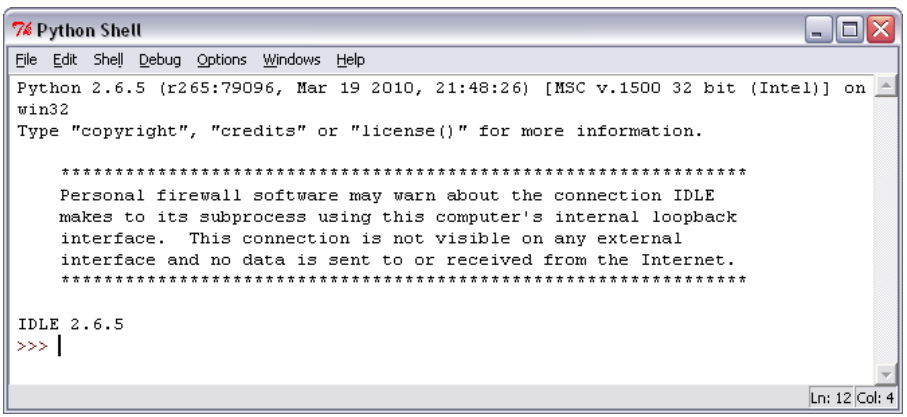


Рис. 1.6. Окно **Python Shell** редактора IDLE

Версии языка Python выпускаются с завидной регулярностью, но, к сожалению, сторонние разработчики не успевают за такой скоростью и не так часто обновляют свои модули. Поэтому приходится при наличии версии Python 3 использовать на практике версию Python 2.6. В некоторых случаях сторонние модули даже не поддерживают версию 2.6. Как же быть, если установлена версия 2.6, а необходимо запустить модуль для версии 2.5 или хочется попробовать возможности версии 3.1? В этом случае удалять версию 2.6 с компьютера не нужно. Все программы установки позволяют выбрать устанавливаемые компоненты. Так, например, существу-

ет возможность задать ассоциацию с файловым расширением. И вот этот компонент необходимо просто отключить при установке.

В качестве примера мы установим на компьютер еще две версии: 2.5.5 и 3.1.2, но вместо программ установки с сайта **http://python.org/** выберем альтернативный дистрибутив от компании ActiveState. Переходим на страницу **http://www.activestate.com/activepython/downloads/** и скачиваем файлы ActivePython-2.5.5.7-win32-x86.msi и ActivePython-3.1.2.3-win32-x86.msi. Последовательность запуска этих программ имеет значение, т. к. при установке в контекстное меню добавляется пункт **Edit with Pythonwin**. С помощью этого пункта запускается редактор PythonWin, который можно использовать вместо IDLE. Так вот из контекстного меню будет открываться версия PythonWin, которая была установлена последней. Поэтому сначала лучше установить версию 2.5, а затем версию 3.1. Установку программ производим в каталоги по умолчанию (C:\Python25\ и C:\Python31\). Обратите особое внимание: при установке в окне **Custom Setup** (рис. 1.7) необходимо отключить компонент **Register as Default Python** (рис. 1.8). Не забудьте это сделать при установке каждой версии, иначе текущей версией будет не Python 2.6.

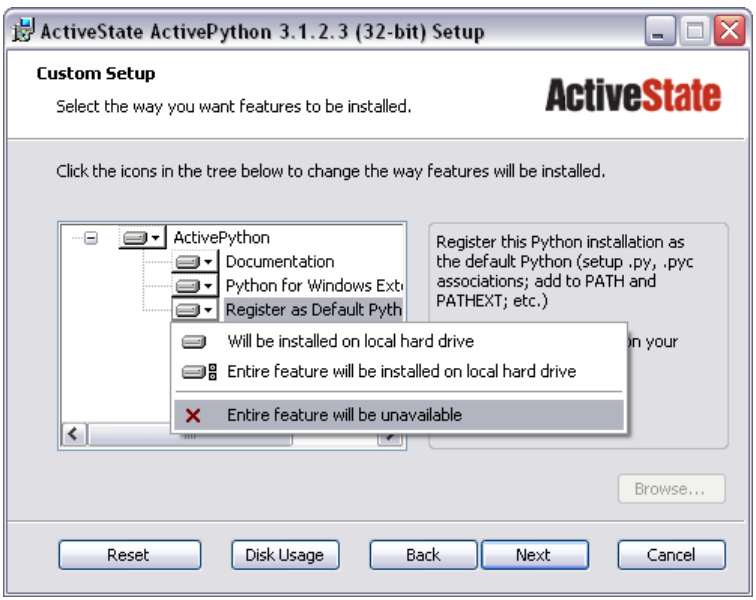


Рис. 1.7. Окно **Custom Setup**

В состав ActivePython кроме редактора PythonWin входит также редактор IDLE. Однако ни в одном меню нет пункта, с помощью которого можно запустить редактор IDLE. Чтобы это исправить, создадим два файла: IDLE25.bat и IDLE31.bat. Содержимое файла IDLE25.bat:

```
@echo off
start C:\Python25\pythonw.exe C:\Python25\Lib\idlelib\idle.pyw
```

Содержимое файла IDLE31.bat:

```
@echo off
start C:\Python31\pythonw.exe C:\Python31\Lib\idlelib\idle.pyw
```

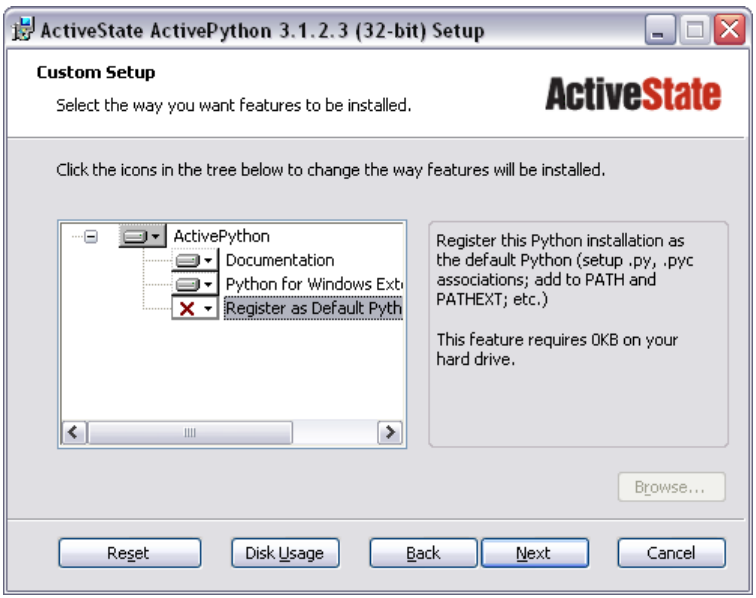


Рис. 1.8. Компонент **Register as Default Python** отключен

С помощью двойного щелчка на этих файлах можно запускать IDLE для версий Python 2.5 и Python 3.1. Чтобы запустить редактор в версии Python 2.6, в меню **Пуск | Программы | Python 2.6** выбираем пункт **IDLE (Python GUI)**.

Теперь рассмотрим запуск программы с помощью разных версий Python. По умолчанию при двойном щелчке на значке файла запускается Python 2.6. Чтобы запустить с помощью другой версии, щелкаем правой кнопкой мыши на значке файла с программой и в контекстном меню выбираем пункт **Открыть с помощью**. При первом запуске в списке будет только программа python.exe. Чтобы добавить другую версию, щелкаем на пункте **Выбрать программу**, в открывшемся окне нажимаем кнопку **Обзор** и выбираем программу python25.exe из папки C:\Python25. Затем добавляем программу python31.exe из папки C:\Python31. В результате список в контекстном меню будет выглядеть так, как показано на рис. 1.9. Выбирая нужную версию из списка, можно производить запуск программы с помощью разных версий Python.

Для проверки установки создайте файл test.py с помощью любого текстового редактора, например Блокнота. Содержимое файла приведено в листинге 1.1.

Листинг 1.1. Проверка установки

```
import sys
print (tuple(sys.version_info))
```

```
try:
    raw_input()         # Python 2
except NameError:
    input()             # Python 3
```

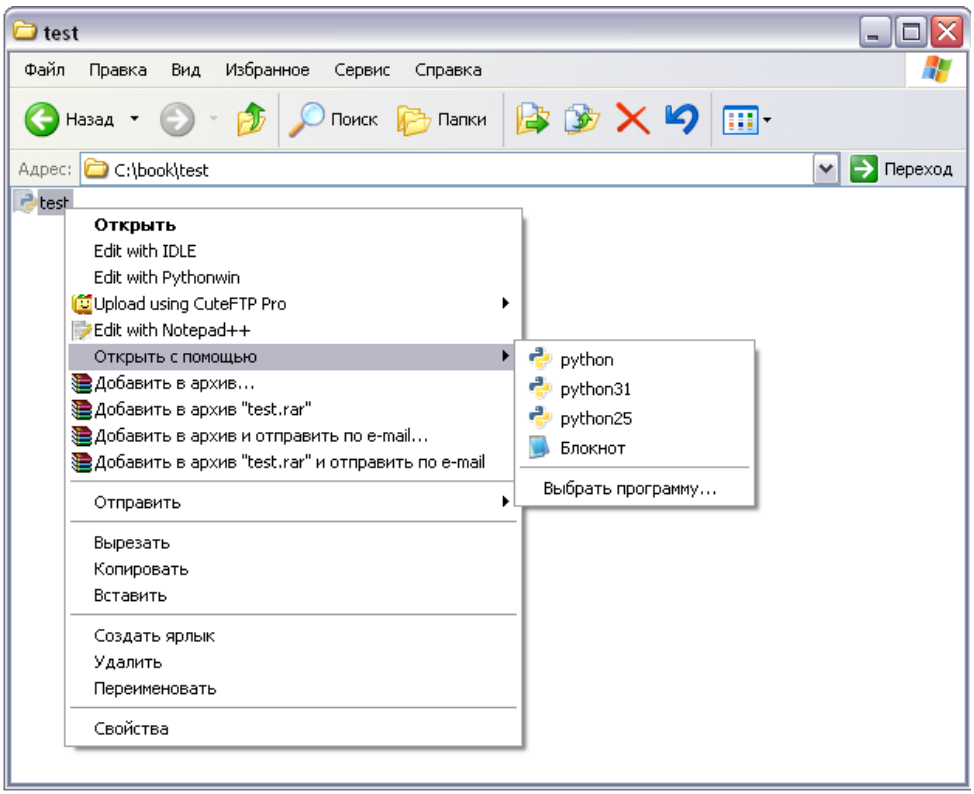


Рис. 1.9. Варианты запуска программы разными версиями Python

Затем запустите программу с помощью двойного щелчка на значке файла. Если результат выполнения — `(2, 6, 5, 'final', 0)`, то установка прошла нормально, а если `(2, 5, 5, 'final', 0)` или `(3, 1, 2, 'final', 0)`, то вы не отключили компонент **Register as Default Python**. Для изучения материала этой книги по умолчанию должна быть версия Python 2.6.5.

1.2. Первая программа на Python

При изучении языков программирования принято начинать с программы, выводящей надпись "Привет, мир!" Не будем нарушать традицию и продемонстрируем, как это будет выглядеть на Python (листинг 1.2).

Листинг 1.2. Первая программа на Python

```
# Выводим надпись с помощью оператора print
print "Привет, мир!"
```

Для запуска программы в меню **Пуск** выбираем пункт **Программы | Python 2.6 | IDLE (Python GUI)**. В результате откроется окно **Python Shell**, в котором символы >>> означают приглашение ввести команду. После ввода команды нажимаем клавишу <Enter>. На следующей строке сразу отобразится результат, а далее приглашение для ввода новой команды. Последовательность выполнения нашей программы показана в листинге 1.3.

Листинг 1.3. Последовательность выполнения программы в окне *Python Shell*

```
>>> # Выводим надпись с помощью оператора print
>>> print "Привет, мир!"
Привет, мир!
>>>
```

Примечание

Символы >>> вводить не нужно, они вставляются автоматически.

Для создания файла с программой в меню **File** выбираем пункт **New Window**. В открывшемся окне набираем код из листинга 1.2, а затем сохраняем его под именем test.py, выбрав пункт меню **File | Save As**. При этом редактор выведет запрос, в какой кодировке сохранить файл, и предложит вставить строку:

```
# -*- coding: cp1251 -*-
```

Соглашаемся и нажимаем кнопку **Edit my file**. В результате строка будет вставлена в самое начало программы, а файл будет сохранен в кодировке Windows-1251.

Запустить программу на выполнение можно, выбрав пункт меню **Run | Run Module** или нажав клавишу <F5>. Результат выполнения программы будет отображен в окне **Python Shell**. По умолчанию в этом окне используется кодировка Windows-1251, поэтому русские буквы отображаются корректно.

Запустить программу можно также с помощью двойного щелчка мыши на значке файла. В этом случае результат выполнения будет отображен в консоли Windows. В консоли используется кодировка cp866, по этой причине файл необходимо сохранить именно в этой кодировке, иначе русские буквы будут искажены. Кроме того, следует учитывать, что после вывода результата окно консоли сразу закрывается. Чтобы предотвратить закрытие окна, необходимо добавить вызов функции `raw_input()`, которая будет ожидать нажатия клавиши <Enter> и не позволит окну сразу закрыться. С учетом всего вышесказанного наша программа будет выглядеть так, как показано в листинге 1.4.

Листинг 1.4. Программа для запуска с помощью двойного щелчка мыши

```
# -*- coding: cp866 -*-
print "Привет, мир!"          # Выводим строку
raw_input()                   # Ожидаем нажатия клавиши <Enter>
```

ПРИМЕЧАНИЕ

Если до функции `raw_input()` возникнет ошибка, то сообщение о ней будет выведено в консоль, но сама консоль после этого сразу будет закрыта, и вы не сможете прочитать сообщение об ошибке.

Если необходимо, чтобы русские буквы правильно отображались в консоли и в окне **Python Shell**, то вместо обычных строк следует использовать Unicode-строки. В этом случае при выводе Unicode-строка будет автоматически преобразована в обычную строку в кодировке терминала. Для создания Unicode-строки можно воспользоваться модификатором `u`, который указывается перед открывающей кавычкой, или функцией `unicode()`. Пример использования Unicode-строк приведен в листинге 1.5.

Листинг 1.5. Пример использования Unicode-строк

```
# -*- coding: cp1251 -*-
print u"Привет, мир!"                    # Модификатор u
print unicode("Привет, мир!", "cp1251") # Функция unicode()
raw_input()
```

Чтобы отредактировать уже созданный файл, щелкаем правой кнопкой мыши на значке файла и из контекстного меню выбираем пункт **Edit with IDLE**. Так как программа на языке Python представляет собой обычный текстовый файл, сохраненный с расширением py, его можно редактировать с помощью других программ, например, Notepad++. Можно также воспользоваться специализированными редакторами, скажем, PyScripter.

При открытии файла с помощью пункта **Edit with IDLE** по умолчанию открываются сразу два окна — окно с текстом программы и окно **Python Shell**. На мой взгляд, это не очень удобно. Чтобы открывалось только одно окно с текстом программы в меню **Options**, выбираем пункт **Configure IDLE**. В открывшемся окне на вкладке **General** устанавливаем флажок **Open Edit Window**.

1.3. Структура программы

Как вы уже знаете, программа на языке Python представляет собой обычный текстовый файл с выражениями. Каждое выражение располагается на отдельной строке. Если выражение не является вложенным, то оно должно начинаться с начала строки, иначе будет выведено сообщение об ошибке (листинг 1.6).

Листинг 1.6. Исключение IndentationError

```
>>>  import sys

  File "<pyshell#0>", line 1
    import sys
   ^
IndentationError: unexpected indent
>>>
```

В этом случае перед оператором `import` расположен один лишний пробел, который привел к выводу сообщения об ошибке.

Если программа предназначена для исполнения в операционной системе UNIX, то в первой строке необходимо дополнительно указать путь к интерпретатору Python:

```
#!/usr/bin/python
```

На некоторых серверах путь к интерпретатору выглядит по-другому:

```
#!/usr/local/bin/python
```

Иногда можно не указывать точный путь к интерпретатору, а передать название языка программе `env`:

```
#!/usr/bin/env python
```

В этом случае программа `env` произведет поиск интерпретатора Python в соответствии с настройками путей поиска.

Помимо указания пути к интерпретатору Python для CGI-программ необходимо, чтобы был установлен бит на выполнение в правах доступа к файлу. Кроме того, следует помнить, что перевод строки в операционной системе Windows состоит из последовательности двух символов — `\r` (перевод каретки) и `\n` (перевод строки). В операционной системе UNIX перевод строки осуществляется только одним символом `\n`. Если загрузить файл программы по протоколу FTP в бинарном режиме, то символ `\r` вызовет фатальную ошибку. По этой причине файлы по протоколу FTP следует загружать только в текстовом режиме (режим `ASCII`). В этом режиме символ `\r` будет удален автоматически. После загрузки файла следует установить права на выполнение. Для исполнения скриптов на Python устанавливаем права в 755 (`-rwxr-xr-x`). Изменить права доступа позволяет практически любой FTP-клиент.

Во второй строке (для ОС Windows в первой строке) следует указать кодировку. Для кодировки Windows-1251 строка будет выглядеть так:

```
# -*- coding: cp1251 -*-
```

Редактор IDLE учитывает указанную кодировку и автоматически производит перекодирование при сохранении файла. Получить полный список поддерживаемых кодировок и их псевдонимы позволяет код, приведенный в листинге 1.7.

Листинг 1.7. Вывод списка поддерживаемых кодировок

```
# -*- coding: cp1251 -*-
import encodings.aliases
arr = encodings.aliases.aliases
keys = arr.keys()
keys.sort()
for key in keys:
    print "%s => %s" % (key, arr[key])
```

Во многих языках программирования (например, в PHP, Perl и др.) каждое выражение должно завершаться точкой с запятой. В языке Python в конце выражения также можно поставить точку с запятой, но это не обязательно. В отличие от языка JavaScript, где рекомендуется завершать выражения точкой с запятой, в языке Python точку с запятой ставить *не рекомендуется*. Концом выражения является конец строки. Тем не менее, если необходимо разместить несколько выражений на одной строке, то точку с запятой *следует указать* (листинг 1.8).

Листинг 1.8. Несколько выражений на одной строке

```
# -*- coding: cp1251 -*-
x = 5; y = 10; z = x + y # Три выражения на одной строке
print z
```

Еще одной отличительной особенностью языка Python является отсутствие ограничительных символов для выделения выражений внутри блока. Например, в языке PHP выражения внутри цикла `while` выделяются фигурными скобками:

```
$i = 1;
while ($i<11) {
   echo $i . "\n";
   $i++;
}
echo "Конец программы";
```

В языке Python тот же код будет выглядеть по-другому (листинг 1.9).

Листинг 1.9. Выделение выражений внутри блока

```
# -*- coding: cp1251 -*-
i = 1
while i<11:
    print i
    i += 1
print "Конец программы"
```

Обратите внимание, что перед всеми выражениями внутри блока расположено одинаковое количество пробелов. Таким образом в языке Python выделяются блоки. Выражения, перед которыми расположено одинаковое количество пробелов, являются телом блока. В нашем примере два выражения выполняются десять раз. Концом блока является выражение, перед которым расположено меньшее количество пробелов. В нашем случае это оператор `print`, который выводит строку "Конец программы". Если количество пробелов внутри блока будет разным, то интерпретатор выведет сообщение о фатальной ошибке и программа будет остановлена. Таким образом, язык Python приучает программистов писать красивый и понятный код.

Примечание

В языке Python принято использовать четыре пробела для выделения выражений внутри блока.

Если блок состоит из одного выражения, то допустимо разместить его на одной строке с основной инструкцией. Например, код

```
for i in range(1, 11):
    print i
print "Конец программы"
```

можно записать так:

```
for i in range(1, 11): print i
print "Конец программы"
```

Если выражение является слишком длинным, то его можно перенести на следующую строку следующими способами:

- в конце строки разместить символ `\`. После этого символа должен следовать символ перевода строки. Другие символы (в том числе и комментарии) недопустимы. Пример:

```
x = 15 + 20 \
    + 30
print x
```

- поместить выражение внутри круглых скобок. Это лучший способ, т. к. любое выражение можно разместить внутри круглых скобок. Пример:

```
x = (15 + 20 # Это комментарий
    + 30)
print x
```

- определение списка и словаря можно разместить на нескольких строках, т. к. используются квадратные и фигурные скобки соответственно. Пример определения списка:

```
arr = [15, 20, # Это комментарий
       30]
print arr
```

Пример определения словаря:

```
arr = {"x": 15, "y": 20, # Это комментарий
       "z": 30}
print arr
```

1.4. Комментарии

Комментарии предназначены для вставки пояснений в текст скрипта, и интерпретатор полностью их игнорирует. Внутри комментария может располагаться любой текст, включая выражения, которые выполнять не следует. Помните, комментарии нужны программисту, а не интерпретатору Python. Вставка комментариев в код позволит через некоторое время быстро вспомнить предназначение фрагмента кода.

В языке Python присутствует только *однострочный комментарий*. Он начинается с символа `#`:

```
# Это комментарий
```

Однострочный комментарий может начинаться не только с начала строки, но и располагаться после выражения. Например, после инструкции вывести надпись "Привет, мир!":

```
print "Привет, мир!" # Выводим надпись с помощью оператора print
```

Если символ комментария разместить перед выражением, то оно не будет выполнено:

```
# print "Привет, мир!" Это выражение выполнено не будет
```

Если символ `#` расположен внутри кавычек или апострофов, то он не является символом комментария:

```
print "# Это НЕ комментарий"
```

Так как в языке Python нет многострочного комментария, то часто комментируемый фрагмент размещают внутри утроенных кавычек (или утроенных апострофов):

```
"""
Это выражение не будет выполнено
print "Привет, мир!"
"""
```

Следует заметить, что этот фрагмент кода не игнорируется интерпретатором, т. к. он не является комментарием. В результате выполнения фрагмента будет создан объект строкового типа. Тем не менее, выражения внутри утроенных кавычек выполнены не будут, т. к. все выражения будут считаться простым текстом. Такие строки являются строками документирования, а не комментариями.

1.5. Скрытые возможности IDLE

На всем протяжении этой книги в качестве редактора мы будем использовать IDLE. По этой причине рассмотрим некоторые возможности этой среды разработки.

Как вы уже знаете, в окне **Python Shell** символы >>> означают приглашение ввести команду. После ввода команды нажимаем клавишу <Enter>. На следующей строке сразу отобразится результат, а далее — приглашение для ввода новой команды. При вводе многострочной команды после нажатия клавиши <Enter> редактор автоматически вставит отступ и будет ожидать дальнейшего ввода. Чтобы сообщить редактору о конце ввода команды необходимо дважды нажать клавишу <Enter>. Пример:

```
>>> for n in range(1, 3):
        print n

1
2
>>>
```

В предыдущем разделе мы выводили строку "Привет, мир!" с помощью оператора `print`. В окне **Python Shell** это делать не обязательно. Например, мы можем просто ввести строку и нажать клавишу <Enter> для получения результата:

```
>>> "Hello, world!"
'Hello, world!'
>>>
```

Однако русские буквы таким образом отображаться не будут. Пример:

```
>>> "Привет, мир!"
'\xcf\xf0\xe8\xe2\xe5\xf2, \xec\xe8\xf0!'
>>>
```

Обратите также внимание, что строки выводятся в апострофах. Этого не произойдет, если выводить строку с помощью оператора `print`:

```
>>> print "Привет, мир!"
Привет, мир!
>>>
```

Учитывая возможность получить результат сразу после ввода команды, окно **Python Shell** можно использовать для изучения команд, а также в качестве многофункционального калькулятора. Пример:

```
>>> 12 * 32 + 54
438
>>>
```

Результат вычисления последнего выражения сохраняется в переменной _ (одно подчеркивание). Это позволяет производить дальнейшие расчеты без ввода предыдущего результата. Вместо него достаточно ввести символ подчеркивания.

Пример:

```
>>> 125 * 3            # Умножение
375
>>> _ + 50             # Сложение. Эквивалентно 375 + 50
425
>>> _ / 5              # Деление. Эквивалентно 425 / 5
85
>>>
```

При вводе команды можно воспользоваться комбинацией клавиш <Ctrl>+<Пробел>. В результате будет отображен список, из которого можно выбрать нужный идентификатор. Если при открытом списке вводить буквы, то показываться будут идентификаторы, начинающиеся с этих букв. Выбирать идентификатор необходимо с помощью клавиш <↑> и <↓>. После выбора не следует нажимать клавишу <Enter>, иначе это приведет к выполнению выражения. Просто вводите выражение дальше, а список закроется. Такой же список будет автоматически появляться (с некоторой задержкой) при обращении к свойствам объекта или атрибутам модулей после ввода точки. Для автоматического завершения идентификатора после ввода первых букв можно воспользоваться комбинацией клавиш <Alt>+</>. При каждом последующем нажатии этой комбинации будет вставляться следующий идентификатор. Эти две комбинации клавиш очень удобны, если вы забыли, как пишется слово, или хотите, чтобы редактор закончил его за вас.

При необходимости повторно выполнить ранее введенное выражение его приходится набирать заново. Можно, конечно, скопировать выражение, а затем вставить, но как вы можете сами убедиться, в контекстном меню нет пунктов **Copy** (Копировать) и **Paste** (Вставить). Они расположены в меню **Edit**. Постоянно выбирать пункты из этого меню очень неудобно. Одним из решений проблемы является использование комбинации клавиш быстрого доступа — <Ctrl>+<C> (копировать) и <Ctrl>+<V> (вставить). Комбинации стандартны для Windows, и вы наверняка их уже использовали ранее. Но опять-таки, прежде чем скопировать выражение, его предварительно необходимо выделить. Редактор IDLE избавляет нас от лишних действий и предоставляет комбинацию клавиш <Alt>+<N> для вставки первого введенного выражения, а также комбинацию <Alt>+<P> для вставки последнего выражения. Каждое последующее нажатие этих клавиш будет вставлять следующее (или предыдущее) выражение. Для еще более быстрого повторного ввода выражения следует предварительно ввести его первые буквы. В этом случае перебираться будут только выражения, начинающиеся в этих букв.

1.6. Вывод результатов работы программы

Вывести результаты работы программы можно с помощью оператора `print`. Мы уже использовали его ранее для вывода строки "Привет, мир!":

```
print "Привет, мир!"
```

Оператор `print` преобразует любой объект в строку и посылает ее в стандартный вывод `stdout`. После строки оператор автоматически добавляет символ перевода строки:

```
print "Строка 1"
print "Строка 2"
```

Результат:

```
Строка 1
Строка 2
```

Если необходимо вывести результат на той же строке, то в операторе `print` данные указываются через запятую:

```
print "Строка 1", "Строка 2"
```

Результат:

```
Строка 1 Строка 2
```

Как видно из примера, между выводимыми строками автоматически вставляется пробел. Чтобы этого избежать следует использовать конкатенацию строк или лучше всего форматирование:

```
print "Строка 1" + "Строка 2"              # Конкатенация строк
# Выведет: Строка 1Строка 2
print '%s%s' % ("Строка 1", "Строка 2")    # Форматирование
# Выведет: Строка 1Строка 2
```

Чтобы избежать добавления символа перевода строки следует последним символом указать запятую. Пример:

```
print "Строка 1", "Строка 2",
print "Строка 3"
# Выведет: Строка 1 Строка 2 Строка 3
```

Если, наоборот, необходимо вставить символ перевода строки, то оператор `print` указывается без параметров. Пример:

```
for n in range(1, 5):
    print n,
print
print "Это текст на новой строке"
```

Результат выполнения:

```
1 2 3 4
Это текст на новой строке
```

В этом примере мы использовали цикл `for`, который позволяет последовательно перебирать элементы. На каждой итерации цикла переменной `n` присваивается новое число, которое мы выводим с помощью оператора `print`, расположенного на следующей строке. Обратите внимание, что перед оператором мы добавили четыре пробела. Таким образом в языке Python выделяются *блоки*. Выражения, перед которыми расположено одинаковое количество пробелов, являются *телом цикла*. Все эти выражения выполняются определенное количество раз. Концом блока является выражение, перед которым расположено меньшее количество пробелов. В нашем

случае это оператор `print` без параметров, который вставляет символ переноса строки.

Если необходимо вывести большой блок текста, то его следует разместить между утроенными кавычками или утроенными апострофами. В этом случае текст сохраняет свое форматирование. Пример:

```
print """Строка 1
Строка 2
Строка 3"""
```

В результате выполнения этого примера мы получим три строки:

```
Строка 1
Строка 2
Строка 3
```

В окне **Python Shell** редактора IDLE необязательно использовать оператор `print`. Однако следует учитывать, что результат вывода будет отличаться:

```
>>> print "Строка"
Строка
>>> "Строка"
'\xd1\xf2\xf0\xee\xea\xe0'
```

Как видно из примера, все русские буквы были заменены соответствующими кодами, а сама строка расположена внутри апострофов. Это происходит потому, что результат автоматически обрабатывается с помощью функции `repr()`. Функция `repr()` возвращает строковое представление объекта. Выведем строку, используя оператор `print` и функцию `repr()`:

```
>>> print repr("Строка")
'\xd1\xf2\xf0\xee\xea\xe0'
```

Для вывода результатов работы программы вместо оператора `print` можно использовать метод `write()` объекта `sys.stdout`:

```
import sys                      # Подключаем модуль sys
sys.stdout.write("Строка")      # Выводим строку
```

В первой строке, с помощью инструкции `import`, мы подключаем модуль `sys`, в котором объявлен объект. Далее с помощью метода `write()` выводим строку. Следует заметить, что метод не вставляет символ перевода строки. Поэтому при необходимости следует добавить его самим с помощью символа `\n`:

```
import sys
sys.stdout.write("Строка 1\n")
sys.stdout.write("Строка 2")
```

1.7. Ввод данных

Для ввода данных предназначены две функции:

- `raw_input([<Сообщение>])` — получает данные со стандартного ввода `stdin`.

Для примера переделаем нашу первую программу так, чтобы она здоровалась не со всем миром, а только с нами:

```
# -*- coding: cp866 -*-
n = raw_input("Введите ваше имя: ")
print "Привет,", n
raw_input("Нажмите <Enter> для закрытия окна")
```

Вводим код и сохраняем файл, например, под названием test.py, а затем запускаем программу на выполнение с помощью двойного щелчка на значке файла. Откроется черное окно, в котором будет надпись "Введите ваше имя: ". Вводим свое имя, например "Николай", и нажимаем клавишу <Enter>. В результате будет выведено приветствие "Привет, Николай". Чтобы окно сразу не закрылось, повторно вызываем функцию `raw_input()`. В этом случае окно не закроется, пока не будет нажата клавиша <Enter>;

- `input([<Сообщение>])` — получает данные со стандартного ввода `stdin` и обрабатывает их с помощью функции `eval()`. Вызов функции `input()` эквивалентен следующему коду:

```
eval(raw_input([<Сообщение>]))
```

Внимание!

Функция `eval()` выполнит любую введенную инструкцию. Никогда не используйте функцию `input()`, если не доверяете пользователю.

Передать данные можно в командной строке после названия файла. Такие данные доступны через список `argv` модуля `sys`. Первый элемент списка `argv` будет содержать название файла, а последующие элементы — переданные данные. В качестве примера создадим файл test.py в папке C:\book. Содержимое файла приведено в листинге 1.10.

Листинг 1.10. Получение данных из командной строки

```
# -*- coding: cp866 -*-
import sys
arr = sys.argv[:]
for n in arr:
    print n
```

Теперь запустим программу на выполнение с помощью командной строки и передадим данные. Запускаем командную строку. Для этого в меню **Пуск** выбираем пункт **Выполнить**. В открывшемся окне набираем команду `cmd` и нажимаем кнопку **OK**. Откроется черное окно, в котором будет приглашение для ввода команд. Переходим в папку C:\book. Для этого набираем команду:

```
cd C:\book
```

В командной строке должно быть приглашение:

```
C:\book>
```

Для запуска нашей программы вводим команду:

```
C:\Python26\python.exe test.py -uNik -p123
```

В этой команде мы передаем название файла (test.py) и некоторые данные (-uNik -p123). Результат выполнения программы будет выглядеть так:

```
test.py
-uNik
-p123
```

1.8. Доступ к документации

Вместе с установкой интерпретатора Python на компьютер автоматически устанавливается документация в формате CHM. Чтобы отобразить документацию, в меню **Пуск** выбираем пункт **Программы | Python 2.6 | Python Manuals**.

Если в меню **Пуск** выбрать пункт **Программы | Python 2.6 | Module Docs**, то откроется окно **pydoc** (рис. 1.10). С помощью этого инструмента можно получить дополнительную информацию, которая расположена внутри модулей. Чтобы получить список всех модулей, установленных на компьютере, оставляем текстовое поле пустым и нажимаем кнопку **open browser**. В результате в окне Web-браузера, используемого в системе по умолчанию, отобразится список всех модулей. Каждое название модуля является ссылкой, при переходе по которой доступна документация по конкретному модулю. Если в окне **pydoc** в текстовом поле ввести какое-либо название и нажать клавишу <Enter>, то будет отображен список совпадений. Выделяем пункт в списке и нажимаем кнопку **go to selected**. Результат будет отображен в окне Web-браузера.

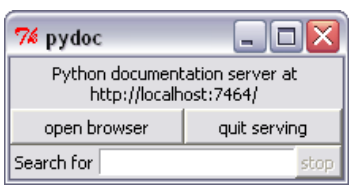


Рис. 1.10. Окно **pydoc**

В окне **Python Shell** редактора IDLE также можно отобразить документацию. Для этого предназначена функция `help()`. В качестве примера отобразим документацию по встроенной функции `raw_input()`:

```
>>> help(raw_input)
```

Результат выполнения:

```
Help on built-in function raw_input in module __builtin__:

raw_input(...)
    raw_input([prompt]) -> string
```

```
    Read a string from standard input.  The trailing newline is stripped.
    If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
    On Unix, GNU readline is used if enabled.  The prompt string, if given,
    is printed without a trailing newline before reading.
```

С помощью функции `help()` можно получить документацию не только по конкретной функции, но и по всему модулю сразу. Для этого предварительно необходимо подключить модуль. Например, подключим модуль `__builtin__`, содержащий определения всех встроенных функций и классов, а затем выведем документацию по модулю:

```
>>> import __builtin__
>>> help(__builtin__)
```

При рассмотрении комментариев мы говорили, что часто для комментирования большого фрагмента кода используются утроенные кавычки или утроенные апострофы. Такие строки не являются комментариями в полном смысле этого слова. Вместо комментирования фрагмента создается объект строкового типа, который сохраняется в атрибуте `__doc__`. Функция `help()` при составлении документации получает информацию из этого атрибута. Таким образом, такие строки называются *строками документирования*.

В качестве примера создадим два файла в одной папке. Содержимое файла test.py:

```
# -*- coding: cp1251 -*-
""" Это описание нашего модуля """
def myFunc():
    """ Это описание функции"""
    pass
```

Теперь подключим этот модуль и выведем содержимое строк документирования:

```
# -*- coding: cp1251 -*-
import test                        # Подключаем файл test.py
print help(test)
```

Результат выполнения:

```
Help on module test:

NAME
    test - Это описание нашего модуля

FILE
    c:\documents and settings\unicross\рабочий стол\test.py

FUNCTIONS
    myFunc()
        Это описание функции
```

Теперь получим содержимое строк документирования с помощью атрибута `__doc__`:

```
# -*- coding: cp1251 -*-
import test                          # Подключаем файл test.py
print test.__doc__
print test.myFunc.__doc__
```

Результат выполнения:

```
Это описание нашего модуля
Это описание функции
```

Атрибут `__doc__` можно использовать вместо функции `help()`. В качестве примера получим документацию по функции `raw_input()`:

```
>>> print raw_input.__doc__
```

Результат выполнения:

```
raw_input([prompt]) -> string

Read a string from standard input.  The trailing newline is stripped.
If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
On Unix, GNU readline is used if enabled.  The prompt string, if given,
is printed without a trailing newline before reading.
```

Получить список всех идентификаторов внутри модуля позволяет функция `dir()`:

```
# -*- coding: cp1251 -*-
import test                          # Подключаем файл test.py
print dir(test)
```

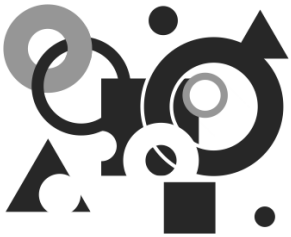
Результат выполнения:

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'myFunc']
```

Теперь получим список всех встроенных идентификаторов:

```
>>> import __builtin__
>>> dir(__builtin__)
```

Глава 2



Переменные

Все данные в языке Python представлены объектами. Каждый объект имеет тип данных и значение. Для доступа к объекту предназначены *переменные*. При инициализации в переменной сохраняется ссылка (адрес объекта в памяти компьютера) на объект. Благодаря этой ссылке можно в дальнейшем изменять объект из программы.

2.1. Именование переменных

Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знаков подчеркивания, причем имя переменной не может начинаться с цифры. Кроме того, следует избегать указания символа подчеркивания в начале имени, т. к. идентификаторы с таким символом имеют специальное значение. Например, имена, начинающиеся с символа подчеркивания, не импортируются из модуля с помощью инструкции `from module import *`, а имена, имеющие по два символа подчеркивания в начале и конце, для интерпретатора имеют особый смысл.

В качестве имени переменной нельзя использовать ключевые слова. Получить список всех ключевых слов позволяет код, приведенный в листинге 2.1.

Листинг 2.1. Список всех ключевых слов

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global',
'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

Помимо ключевых слов следует избегать совпадений со встроенными идентификаторами. В отличие от ключевых слов, встроенные идентификаторы можно переопределять, но дальнейший результат может стать для вас неожиданным (листинг 2.2).

Листинг 2.2. Ошибочное переопределение встроенных идентификаторов

```
>>> help(input)
Help on built-in function input in module __builtin__:

input(...)
    input([prompt]) -> value

    Equivalent to eval(raw_input(prompt)).

>>> help = 10
>>> help
10
>>> help(input)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    help(input)
TypeError: 'int' object is not callable
```

В этом примере мы с помощью встроенной функции `help()` получаем справку по функции `input()`. Далее переменной `help` присваиваем число 10. После переопределения идентификатора мы больше не можем пользоваться функцией `help()`, т. к. это приведет к выводу сообщения об ошибке. По этой причине лучше избегать имен, совпадающих со встроенными идентификаторами. Получить полный список встроенных идентификаторов позволяет код, приведенный в листинге 2.3.

Листинг 2.3. Получение списка встроенных идентификаторов

```
>>> import __builtin__
>>> dir(__builtin__)
```

Правильные имена переменных: `x`, `y1`, `strName`.

Неправильные имена переменных: `1y`, `ИмяПеременной`.

Последнее имя неправильное, т. к. в нем используются русские буквы. Хотя на самом деле такой вариант также будет работать, но лучше русские буквы все же не применять:

```
>>> ИмяПеременной = 10              # Лучше так не делать!!!
>>> ИмяПеременной
10
```

При указании имени переменной важно учитывать регистр букв: `x` и `X` — разные переменные:

```
>>> x = 10; X = 20
>>> x, X
(10, 20)
```

2.2. Типы данных

В языке Python объекты могут иметь следующие типы данных:

- `int` — целые числа в диапазоне от –2 147 483 648 до 2 147 483 647:

```
>>> print type(2147483647)
<type 'int'>
```

Если число выходит за рамки диапазона, то тип `int` автоматически преобразуется в тип `long`:

```
>>> x = 2147483647; print type(x)
<type 'int'>
>>> x += 1; print type(x)
<type 'long'>
```

- `long` — длинные целые числа. Размер числа ограничен лишь объемом оперативной памяти:

```
>>> print type(2147483648)
<type 'long'>
```

В окне **Python Shell** редактора IDLE число, имеющее тип `long`, выводится с буквой `L` в конце:

```
>>> 2147483648
2147483648L
```

- `float` — вещественные числа:

```
>>> print type(5.1)
<type 'float'>
```

- `complex` — комплексные числа:

```
>>> print type(2+2j)
<type 'complex'>
```

- `str` — обычная строка. Может содержать как однобайтовые символы, так и многобайтовые:

```
>>> print type("Строка")
<type 'str'>
```

- `unicode` — Unicode-строка:

```
>>> print type(u"Строка")
<type 'unicode'>
```

- `bool` — логический тип данных. Может содержать значения `True` или `False`, которые соответствуют числовым значениям `1` и `0` соответственно:

```
>>> print type(True), type(False)
<type 'bool'> <type 'bool'>
```

- `list` — списки. Тип данных `list` аналогичен массивам в других языках программирования:

```
>>> print type([1, 2, 3])
<type 'list'>
```

- `tuple` — кортежи:

```
>>> print type((4, 5, 6))
<type 'tuple'>
```

- `dict` — словари. Тип данных `dict` аналогичен ассоциативным массивам в других языках программирования:

```
>>> print type({"x": 5, "y": 20})
<type 'dict'>
```

- `set` — множества:

```
>>> print type(set(["a", "b", "c"]))
<type 'set'>
```

- `frozenset` — неизменяемые множества:

```
>>> print type(frozenset(["a", "b", "c"]))
<type 'frozenset'>
```

- `NoneType` — объект со значением `None`:

```
>>> print type(None)
<type 'NoneType'>
```

  В логическом контексте значение `None` интерпретируется как `False`:

```
>>> print bool(None)
False
```

- `function` — функции:

```
>>> def myFunc(): pass

>>> print type(myFunc)
<type 'function'>
```

- `classobj` — классические классы:

```
>>> class C1: pass

>>> print type(C1)
<type 'classobj'>
```

- `instance` — экземпляры классических классов:

```
>>> class C1: pass

>>> c1 = C1()          # Создание экземпляра класса
>>> print type(c1)
<type 'instance'>
```

- `module` — модули:

```
>>> import sys
>>> print type(sys)
<type 'module'>
```

- `file` — файлы:

```
>>> f = open("file1.txt", "w")
>>> print type(f)
<type 'file'>
```

- `type` — типы данных. Не удивляйтесь! Все данные в языке Python являются объектами, даже сами типы данных!

```
>>> print type(type(""))
<type 'type'>
```

Основные типы данных делятся на *изменяемые* и *неизменяемые*. К изменяемым типам относятся списки и словари. Пример изменения элемента списка:

```
>>> arr = [1, 2, 3]
>>> arr[0] = 0                  # Изменяем первый элемент списка
>>> arr
[0, 2, 3]
```

К неизменяемым типам относятся числа, строки и кортежи. Например, чтобы получить строку из двух других строк, необходимо использовать операцию *конкатенации*, а ссылку на новый объект присвоить переменной:

```
>>> str1 = "авто"
>>> str2 = "транспорт"
>>> str3 = str1 + str2          # Конкатенация
>>> print str3
автотранспорт
```

Кроме того, типы данных делятся на *последовательности* и *отображения*. К последовательностям относятся строки, списки и кортежи, а к отображениям — словари. Последовательности поддерживают механизм итераторов, позволяющий произвести обход всех элементов с помощью метода `next()`. Например, вывести элементы списка можно так:

```
>>> arr = [1, 2]
>>> i = iter(arr)
>>> i.next()
1
>>> i.next()
2
```

На практике подобным способом не пользуются. Вместо него применяется цикл `for`, который использует механизм итераторов незаметно для нас. Например, вывести элементы списка можно так:

```
>>> for i in [1, 2]:
        print i
```

Перебрать слово по буквам можно точно также. Для примера вставим тире после каждой буквы:

```
>>> for i in "Строка":
        print i + " -",
```

Результат:

```
С - т - р - о - к - а -
```

Кроме того, последовательности поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`). Все эти операции мы будем подробно рассматривать по мере изучения языка.

2.3. Инициализация переменных

В языке Python используется *динамическая типизация*. Это означает, что при инициализации переменной интерпретатор автоматически относит переменную к одному из типов данных. Значение переменной присваивается с помощью оператора = таким образом:

```
>>> x = 7           # Тип int
>>> y = 7.8         # Тип float
>>> s1 = "Строка"   # Переменной s1 присвоено значение Строка
>>> s2 = 'Строка'   # Переменной s2 также присвоено значение Строка
>>> b = True        # Переменной b присвоено логическое значение True
```

В одной строке можно присвоить значение сразу нескольким переменным:

```
>>> x = y = 10
```

При инициализации в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел и строк, но для других объектов этого делать нельзя. Пример:

```
>>> x = y = [1, 2]          # Якобы создали два объекта
>>> x, y
([1, 2], [1, 2])
```

В этом примере мы создали список из двух элементов и присвоили значение переменным `x` и `y`. Теперь попробуем изменить значение в переменной `y`:

```
>>> y[1] = 100              # Изменяем второй элемент
>>> x, y
([1, 100], [1, 100])
```

Как видно из примера, изменение значения в переменной `y` привело также к изменению значения в переменной `x`. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> x = [1, 2]
>>> y = [1, 2]
>>> y[1] = 100              # Изменяем второй элемент
>>> x, y
([1, 2], [1, 100])
```

Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор `is`. Если переменные ссылаются на один и тот же объект, то оператор `is` возвращает значение `True`:

```
>>> x = y = [1, 2]          # Один объект
>>> x is y
True
>>> x = [1, 2]              # Разные объекты
>>> y = [1, 2]              # Разные объекты
>>> x is y
False
```

Следует заметить, что в целях эффективности кода интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что если ста переменным присвоено число 2, то в этих переменных будет сохранена ссылка на один и тот же объект. Пример:

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
(True, True)
```

Посмотреть количество ссылок на объект позволяет метод `getrefcount()` из модуля `sys`:

```
>>> import sys                          # Подключаем модуль sys
>>> sys.getrefcount(2)
375
```

Когда число ссылок на объект становится равно нулю, объект автоматически удаляется из оперативной памяти. Исключением являются объекты, которые подлежат кэшированию.

Помимо группового присваивания язык Python поддерживает позиционное присваивание. В этом случае переменные указываются через запятую слева от оператора =, а значения — через запятую справа. Пример позиционного присваивания:

```
>>> x, y, z = 1, 2, 3
>>> x, y, z
(1, 2, 3)
```

С помощью позиционного присваивания можно поменять значения переменных местами. Пример:

```
>>> x, y = 1, 2; x, y
(1, 2)
>>> x, y = y, x; x, y
(2, 1)
```

По обе стороны оператора = могут быть указаны последовательности. Напомню, что к последовательностям относятся строки, списки и кортежи. Пример:

```
>>> x, y, z = "123"               # Строка
>>> x, y, z
('1', '2', '3')
>>> x, y, z = [1, 2, 3]           # Список
>>> x, y, z
(1, 2, 3)
>>> x, y, z = (1, 2, 3)           # Кортеж
>>> x, y, z
(1, 2, 3)
```

Количество элементов справа и слева от оператора = должно совпадать, иначе будет выведено сообщение об ошибке:

```
>>> x, y, z = (1, 2, 3, 4)
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    x, y, z = (1, 2, 3, 4)
ValueError: too many values to unpack
```

2.4. Проверка типа данных

Python в любой момент времени изменяет тип переменной в соответствии с данными, хранящимися в ней. Пример:

```
>>> a = "Строка"            # Тип str
>>> a = 7                   # Теперь переменная имеет тип int
```

Функция `type(<Имя переменной>)` возвращает тип данных переменной:

```
>>> print type(a)
<type 'int'>
```

Проверить тип данных переменной можно следующими способами:

- сравнить значение, возвращаемое функцией `type()`, с названием типа данных:

```
>>> x = 10
>>> if type(x) == int:
        print "Это тип int"
```

- проверить тип с помощью функции `isinstance()`:

```
>>> s = "Строка"
>>> if isinstance(s, str):
        print "Это тип str"
```

2.5. Преобразование типов данных

Как вы уже знаете, в языке Python используется *динамическая типизация*. После присваивания значения в переменной сохраняется ссылка на объект определенного типа, а не сам объект. Если затем переменной присвоить значение другого типа, то переменная будет ссылаться на другой объект, и тип данных соответственно изменится. Таким образом, тип данных в языке Python — это характеристика объекта, а не переменной. Переменная всегда содержит только ссылку на объект.

После присваивания переменной значения над объектом можно производить операции, предназначенные для этого типа данных. Например, строку нельзя сложить с числом, т. к. это приведет к выводу сообщения об ошибке:

```
>>> 2 + "25"
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    2 + "25"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Для преобразования типов данных предназначены следующие функции:

- `int([<Объект>[, <Система счисления>]])` — преобразует объект в число. Во втором параметре можно указать систему счисления (значение по умолчанию — 10). Примеры:

```
>>> int(7.5), int("71")
(7, 71)
>>> int("71", 10), int("71", 8), int("A", 16)
(71, 57, 10)
```

Если преобразование невозможно, то возвращается сообщение об ошибке:

```
>>> int("71s")
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    int("71s")
ValueError: invalid literal for int() with base 10: '71s'
```

- `float([<Число или строка>])` — преобразует целое число или строку в вещественное число. Примеры:

```
>>> float(7), float("7.1")
(7.0, 7.0999999999999996)
```

- `str([<Объект>])` — преобразует объект в строку. Примеры:

```
>>> str(125), str([1, 2, 3])
('125', '[1, 2, 3]')
>>> str((1, 2, 3)), str({"x": 5, "y": 10})
('(1, 2, 3)', "{'y': 10, 'x': 5}")
```

- `unicode(<Строка>[, <Кодировка>[, <Обработка ошибок>]])` — преобразует обычную строку в Unicode-строку. В третьем параметре могут быть указаны значения "strict" (значение по умолчанию), "replace" или "ignore". Примеры:

```
>>> unicode("Строка в кодировке windows-1251", "cp1251")
u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432
\u043a\u043e\u0434\u0438\u0440\u043e\u0432\u043a\u0435
windows-1251'
>>> unicode("Строка", "utf-8", "strict")
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    unicode("Строка", "utf-8", "strict")
UnicodeDecodeError: 'utf8' codec can't decode bytes in position
0-1: invalid data
>>> unicode("Строка", "utf-8", "replace")
u'\ufffd\ufffd'
>>> unicode("Строка", "utf-8", "ignore")
u''
```

- `bool(<Объект>)` — преобразует объект в логический тип данных. Примеры:

```
>>> bool(0), bool(1), bool(""), bool("Строка")
(False, True, False, True)
```

- `list(<Последовательность>)` — преобразует элементы последовательности в список. Примеры:

```
>>> list("12345")              # Преобразование строки
['1', '2', '3', '4', '5']
>>> list((1, 2, 3, 4, 5))      # Преобразование кортежа
[1, 2, 3, 4, 5]
```

- `tuple(<Последовательность>)` — преобразует элементы последовательности в кортеж:

```
>>> tuple("123456")            # Преобразование строки
('1', '2', '3', '4', '5', '6')
>>> tuple([1, 2, 3, 4, 5])     # Преобразование списка
(1, 2, 3, 4, 5)
```

В качестве примера рассмотрим возможность сложения двух чисел, введенных пользователем. Как вы уже знаете, вводить данные позволяет функция `raw_input()`. Воспользуемся этой функцией для получения чисел от пользователя (листинг 2.4).

Листинг 2.4. Получение данных от пользователя

```
# -*- coding: cp1251 -*-
x = raw_input("x = ")            # Вводим 5
y = raw_input("y = ")            # Вводим 12
print x + y
```

Результатом выполнения этого скрипта будет не число, а строка "512". Таким образом, следует запомнить, что функция `raw_input()` возвращает результат в виде строки. Чтобы просуммировать два числа, необходимо преобразовать строку в число (листинг 2.5).

Листинг 2.5. Преобразование строки в число

```
# -*- coding: cp1251 -*-
x = int(raw_input("x = "))       # Вводим 5
y = int(raw_input("y = "))       # Вводим 12
print x + y
```

В этом случае мы получим число 17, как и должно быть. Однако если пользователь вместо числа введет строку, то программа завершится с фатальной ошибкой. Как обработать ошибку, мы будем рассматривать по мере изучения языка.

2.6. Удаление переменной

Удалить переменную можно с помощью инструкции `del`:

```
del <Переменная1>[, ..., <ПеременнаяN>]
```

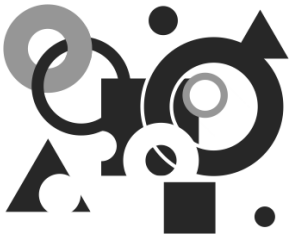
Пример удаления одной переменной:

```
>>> x = 10; x
10
>>> del x; x
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x
NameError: name 'x' is not defined
```

Пример удаления нескольких переменных:

```
>>> x, y = 10, 20
    >>> del x, y
```

Глава 3



Операторы Python

Операторы позволяют произвести определенные действия с данными. Например, операторы присваивания служат для сохранения данных в переменной, математические операторы позволяют произвести арифметические вычисления, а оператор конкатенации строк используется для соединения двух строк в одну. Рассмотрим операторы, доступные в Python, более подробно.

3.1. Математические операторы

Производить операции над числами позволяют следующие операторы:

- `+` — сложение:

```
>>> 10 + 5              # Целые числа
15
>>> 12.4 + 5.2          # Вещественные числа
17.600000000000001
>>> 10 + 12.4           # Целые и вещественные числа
22.399999999999999
```

- `-` — вычитание:

```
>>> 10 - 5              # Целые числа
5
>>> 12.4 - 5.2          # Вещественные числа
7.2000000000000002
>>> 12 - 5.2            # Целые и вещественные числа
6.7999999999999998
```

- `*` — умножение:

```
>>> 10 * 5              # Целые числа
50
>>> 12.4 * 5.2          # Вещественные числа
64.480000000000004
>>> 10 * 5.2            # Целые и вещественные числа
52.0
```

- `/` — деление. Если производится деление целых чисел, то остаток отбрасывается и возвращается целое число. Деление вещественных чисел производится классическим способом. Примеры:

```
>>> 10 / 5                # Деление целых чисел без остатка
2
>>> 10 / 3                # Деление целых чисел с остатком
3
>>> 10.0 / 5.0            # Деление вещественных чисел
2.0
>>> 10.0 / 3.0            # Деление вещественных чисел
3.3333333333333335
>>> 10 / 5.0              # Деление целого числа на вещественное
2.0
>>> 10.0 / 5              # Деление вещественного числа на целое
2.0
```

- `//` — деление с округлением вниз. Вне зависимости от типа чисел остаток отбрасывается. Примеры:

```
>>> 10 // 5            # Деление целых чисел без остатка
2
>>> 10 // 3            # Деление целых чисел с остатком
3
>>> 10.0 // 5.0        # Деление вещественных чисел
2.0
>>> 10.0 // 3.0        # Деление вещественных чисел
3.0
>>> 10 // 5.0          # Деление целого числа на вещественное
2.0
>>> 10 // 3.0          # Деление целого числа на вещественное
3.0
>>> 10.0 // 5          # Деление вещественного числа на целое
2.0
>>> 10.0 // 3          # Деление вещественного числа на целое
3.0
```

- `%` — остаток от деления:

```
>>> 10 % 5             # Деление целых чисел без остатка
0
>>> 10 % 3             # Деление целых чисел с остатком
1
>>> 10.0 % 5.0         # Операция над вещественными числами
0.0
>>> 10.0 % 3.0         # Операция над вещественными числами
1.0
>>> 10 % 5.0  # Операция над целыми и вещественными числами
0.0
>>> 10 % 3.0  # Операция над целыми и вещественными числами
```

```
1.0
>>> 10.0 % 5  # Операция над целыми и вещественными числами
0.0
>>> 10.0 % 3  # Операция над целыми и вещественными числами
1.0
```

- `**` — возведение в степень:

```
>>> 10 ** 2, 10.0 ** 2
(100, 100.0)
```

- унарный – (минус) и унарный + (плюс):

```
>>> +10, +10.0, -10, -10.0, -(-10), -(-10.0)
(10, 10.0, -10, -10.0, 10, 10.0)
```

Как видно из примеров, операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут длинные целые числа, вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое и вещественное числа, то целое число будет автоматически преобразовано в вещественное число, а затем произведена операция над вещественными числами. Результатом этой операции будет вещественное число.

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 - 0.1 - 0.1 - 0.1
-2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат. Если необходимо производить операции с фиксированной точностью, то следует использовать модуль `decimal`:

```
>>> from decimal import Decimal
>>> Decimal("0.3") - Decimal("0.1") - Decimal("0.1") - Decimal("0.1")
Decimal('0.0')
```

3.2. Двоичные операторы

- `~` — двоичная инверсия;
- `&` — двоичное И;
- `|` — двоичное ИЛИ;
- `^` — двоичное исключающее ИЛИ;
- `<<` — сдвиг влево;
- `>>` — сдвиг вправо.

3.3. Операторы для работы с последовательностями

Для работы с последовательностями предназначены следующие операторы:

- `+` — конкатенация:

```
>>> print "Строка1" + "Строка2",   # Конкатенация строк
Строка1Строка2
>>> [1, 2, 3] + [4, 5, 6]          # Списки
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)          # Кортежи
(1, 2, 3, 4, 5, 6)
```

- `*` — повторение:

```
>>> "s" * 20                       # Строки
'ssssssssssssssssssss'
>>> [1, 2] * 3                     # Списки
[1, 2, 1, 2, 1, 2]
>>> (1, 2) * 3                     # Кортежи
(1, 2, 1, 2, 1, 2)
```

- `in` — проверка на вхождение. Если элемент входит в последовательность, то возвращается логическое значение `True`:

```
>>> "Строка" in "Строка для поиска"   # Строки
True
>>> "Строка2" in "Строка для поиска"  # Строки
False
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]    # Списки
(True, False)
>>> 2 in (1, 2, 3), 6 in (1, 2, 3)    # Кортежи
(True, False)
```

3.4. Операторы присваивания

- `=` — присваивает переменной значение:

```
>>> x = 5; x
5
```

- `+=` — увеличивает значение переменной на указанную величину:

```
>>> x = 5; x += 10            # Эквивалентно x = x + 10
>>> x
15
```

Для последовательностей оператор `+=` производит конкатенацию:

```
>>> s = "Стр"; s += "ока"
>>> print s
Строка
```

- `-=` — уменьшает значение переменной на указанную величину:

```
>>> x = 10; x -= 5            # Эквивалентно x = x - 5
>>> x
5
```

- `*=` — умножает значение переменной на указанную величину:

```
>>> x = 10; x *= 5            # Эквивалентно x = x * 5
>>> x
50
```

Для последовательностей оператор `*=` производит повторение:

```
>>> s = "*"; s *= 20
>>> s
'********************'
```

- `/=` — делит значение переменной на указанную величину:

```
>>> x = 10; x /= 3            # Эквивалентно x = x / 3
>>> x
3
>>> y = 10.0; y /= 3.0        # Эквивалентно y = y / 3.0
>>> y
3.3333333333333335
```

- `//=` — деление с округлением вниз и присваиванием:

```
>>> x = 10; x //= 3           # Эквивалентно x = x // 3
>>> x
3
>>> y = 10.0; y //= 3.0       # Эквивалентно y = y // 3.0
>>> y
3.0
```

- `%=` — деление по модулю и присваивание:

```
>>> x = 10; x %= 2            # Эквивалентно x = x % 2
>>> x
0
>>> y = 10; y %= 3            # Эквивалентно y = y % 3
>>> y
1
```

- `**=` — возведение в степень и присваивание:

```
>>> x = 10; x **= 2           # Эквивалентно x = x ** 2
>>> x
100
```

3.5. Приоритет выполнения операторов

В какой последовательности будет вычисляться приведенное ниже выражение?

```
x = 5 + 10 * 3 / 2
```

Это зависит от приоритета выполнения операторов.

В данном случае последовательность вычисления выражения будет такой:

1. Число 10 будет умножено на 3, т. к. приоритет оператора умножения выше приоритета оператора сложения.
2. Полученное значение будет поделено на 2, т. к. приоритет оператора деления равен приоритету оператора умножения (а операторы с равными приоритетами выполняются слева направо), но выше чем у оператора сложения.
3. К полученному значению будет прибавлено число 5, т. к. оператор присваивания = имеет наименьший приоритет.
4. Значение будет присвоено переменной x.

```
>>> x = 5 + 10 * 3 / 2
>>> x
20
```

С помощью скобок можно изменить последовательность вычисления выражения:

```
x = (5 + 10) * 3 / 2
```

Теперь порядок вычислений будет другим:

1. К числу 5 будет прибавлено 10.
2. Полученное значение будет умножено на 3.
3. Полученное значение будет поделено на 2.
4. Значение будет присвоено переменной x.

```
>>> x = (5 + 10) * 3 / 2
>>> x
22
```

Перечислим операторы в порядке убывания приоритета:

1. `-x`, `+x`, `~x`, `**` — унарный минус, унарный плюс, двоичная инверсия, возведение в степень. Если унарные операторы расположены слева от оператора `**`, то возведение в степень имеет больший приоритет, а если справа — то меньший. Например, выражение

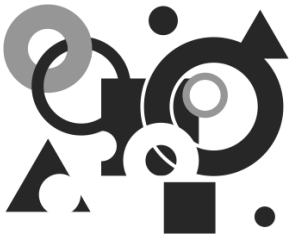
```
-10 ** -2
```

эквивалентно следующей расстановке скобок:

```
-(10 ** (-2))
```

2. `*`, `%`, `/`, `//` — умножение (повторение), остаток от деления, деление, деление с округлением вниз.
3. `+`, `-` — сложение (конкатенация), вычитание.
4. `<<`, `>>` — двоичные сдвиги.
5. `&` — двоичное И.
6. `^` — двоичное исключающее ИЛИ.
7. `|` — двоичное ИЛИ.
8. `=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=` — присваивание.

Глава 4



Условные операторы и циклы

Условные операторы позволяют в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнять его. Логические выражения возвращают только два значения: `True` (истина) или `False` (ложь), которые соответствуют числовым значениям `1` и `0` соответственно:

```
>>> True + 2                    # Эквивалентно 1 + 2
3
>>> False + 2                   # Эквивалентно 0 + 2
2
```

Логическое значение можно сохранить в переменной:

```
>>> x = True; y = False
>>> x, y
(True, False)
```

Любой объект в логическом контексте может интерпретироваться как истина (`True`) или как ложь (`False`). Для определения логического значения можно использовать функцию `bool()`.

Значение `True` возвращают следующие объекты:

- любое число, не равное нулю:

```
>>> bool(1), bool(20), bool(-20)
(True, True, True)
>>> bool(1.0), bool(0.1), bool(-20.0)
(True, True, True)
```

- не пустой объект:

```
>>> bool("0"), bool([0, None]), bool((None,)), bool({"x": 5})
(True, True, True, True)
```

Следующие объекты интерпретируются как `False`:

- число, равное нулю:

```
>>> bool(0), bool(0.0)
(False, False)
```

- пустой объект:

```
>>> bool(""), bool([]), bool(())
(False, False, False)
```

- значение `None`:

```
>>> bool(None)
False
```

4.1. Операторы сравнения

Операторы сравнения используются в логических выражениях. Перечислим их:

- `==` — равно:

```
>>> 1 == 1, 1 == 5
(True, False)
```

- `!=` и `<>` — не равно:

```
>>> 1 != 5, 1 <> 5, 1 != 1
(True, True, False)
```

Примечание

Оператор `<>` признан устаревшим и не рекомендуется к использованию.

- `<` — меньше:

```
>>> 1 < 5, 1 < 0
(True, False)
```

- `>` — больше:

```
>>> 1 > 0, 1 > 5
(True, False)
```

- `<=` — меньше или равно:

```
>>> 1 <= 5, 1 <= 0, 1 <= 1
(True, False, True)
```

- `>=` — больше или равно:

```
>>> 1 >= 0, 1 >= 5, 1 >= 1
(True, False, True)
```

- `in` — проверка на вхождение в последовательность:

```
>>> "Строка" in "Строка для поиска" # Строки
True
>>> 2 in [1, 2, 3], 4 in [1, 2, 3]  # Списки
(True, False)
>>> 2 in (1, 2, 3), 4 in (1, 2, 3)  # Кортежи
(True, False)
```

- `is` — проверяет, ссылаются ли две переменные на один и тот же объект. Если переменные ссылаются на один и тот же объект, то оператор `is` возвращает значение `True`:

```
>>> x = y = [1, 2]
>>> x is y
True
>>> x = [1, 2]; y = [1, 2]
>>> x is y
False
```

  Следует заметить, что в целях эффективности интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что если ста переменным присвоено число 2, то в этих переменных будет сохранена ссылка на один и тот же объект. Пример:

```
>>> x = 2; y = 2; z = 2
>>> x is y, y is z
(True, True)
```

Значение логического выражения можно инвертировать с помощью оператора `not`:

```
>>> x = 1; y = 1
>>> x == y
True
>>> not (x == y), not x == y
(False, False)
```

Если переменные `x` и `y` равны, то возвращается значение `True`, но т. к. перед выражением стоит оператор `not`, выражение вернет `False`. Круглые скобки можно не указывать, т. к. оператор `not` имеет более низкий приоритет выполнения, чем операторы сравнения.

При необходимости инвертировать значение оператора `in` оператор `not` указывается непосредственно перед этим оператором:

```
>>> 2 in [1, 2, 3], 2 not in [1, 2, 3]
(True, False)
```

Чтобы инвертировать значение оператора `is`, оператор `not` указывается непосредственно после этого оператора:

```
>>> x = y = [1, 2]
>>> x is y, x is not y
(True, False)
```

В логическом выражении можно указывать сразу несколько условий:

```
>>> x = 10
>>> 1 < x < 20, 11 < x < 20
(True, False)
```

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

- `and` — `x and y` — логическое И. Если `x` интерпретируется как `False`, то возвращается `x`, в противном случае возвращается `y`:

```
>>> 1 < 5 and 2 < 5              # True and True == True
True
>>> 1 < 5 and 2 > 5              # True and False == False
False
>>> 1 > 5 and 2 < 5              # False and True == False
False
>>> 10 and 20, 0 and 20, 10 and 0
(20, 0, 0)
```

- `or` — `x or y` — логическое ИЛИ. Если `x` интерпретируется как `False`, то возвращается `y`, в противном случае возвращается `x`:

```
>>> 1 < 5 or 2 < 5               # True or True == True
True
>>> 1 < 5 or 2 > 5               # True or False == True
True
>>> 1 > 5 or 2 < 5               # False or True == True
True
>>> 1 > 5 or 2 > 5               # False or False == False
False
>>> 10 or 20, 0 or 20, 10 or 0
(10, 20, 10)
>>> 0 or "" or None or [] or "s"
's'
```

Это выражение вернет `True` только в случае, если оба выражения вернут `True`:

```
x1 == x2 and x2 != x3
```

А это выражение вернет `True`, если хотя бы одно из выражений вернет `True`:

```
x1 == x2 or x3 == x4
```

Перечислим операторы сравнения в порядке убывания приоритета:

1. `<`, `>`, `<=`, `>=`, `==`, `!=`, `<>`, `is`, `is not`, `in`, `not in`.
2. `not` — логическое отрицание.
3. `and` — логическое И.
4. `or` — логическое ИЛИ.

4.2. Оператор ветвления *if...else*

Оператор ветвления `if...else` позволяет в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнять его.

Оператор имеет следующий формат:

```
if <Логическое выражение>:
    <Блок, выполняемый если условие истинно>
[elif <Логическое выражение>:
    <Блок, выполняемый если условие истинно>
]
[else:
    <Блок, выполняемый если все условия ложны>
]
```

Как вы уже знаете, блоки внутри составной инструкции выделяются одинаковым количеством пробелов (обычно четыре пробела). Концом блока является выражение, перед которым расположено меньшее количество пробелов. В некоторых языках программирования логическое выражение заключается в круглые скобки. В языке Python это делать необязательно, но можно, т. к. любое выражение может быть расположено внутри круглых скобок. Круглые скобки следует использовать только при необходимости разместить условие на нескольких строках.

Для примера напишем программу, которая проверяет, является ли введенное пользователем число четным или нет (листинг 4.1). После проверки выводится соответствующее сообщение.

Листинг 4.1. Проверка числа на четность

```
# -*- coding: cp1251 -*-
x = int(raw_input("Введите число: "))
if x % 2 == 0:
    print x, "- Четное число"
else:
    print x, "- Нечетное число"
```

Если блок состоит из одной инструкции, то эту инструкцию можно разместить на одной строке с заголовком:

```
# -*- coding: cp1251 -*-
x = int(raw_input("Введите число: "))
if x % 2 == 0: print x, "- Четное число"
else: print x, "- Нечетное число"
```

В этом случае концом блока является конец строки. Это означает, что можно разместить сразу несколько инструкций на одной строке, разделяя их точкой с запятой:

```
# -*- coding: cp1251 -*-
x = int(raw_input("Введите число: "))
if x % 2 == 0: print x,; print "- Четное число"
else: print x,; print "- Нечетное число"
```

Знайте, что так можно сделать, но никогда на практике не пользуйтесь этим способом, т. к. подобная конструкция нарушает стройность кода и ухудшает его сопровождение в дальнейшем. Всегда размещайте инструкцию на отдельной строке, даже если блок содержит только одну инструкцию. Согласитесь, что этот код читается намного проще, чем предыдущий:

```
# -*- coding: cp1251 -*-
x = int(raw_input("Введите число: "))
if x % 2 == 0:
    print x,
    print "- Четное число"
else:
    print x,
    print "- Нечетное число"
```

Оператор `if...else` позволяет проверить сразу несколько условий. Рассмотрим это на примере (листинг 4.2).

Листинг 4.2. Проверка нескольких условий

```
# -*- coding: cp1251 -*-
print """Какой операционной системой вы пользуетесь?
1 - Windows 98
2 - Windows XP
3 - Windows Vista
4 - Другая"""
os = raw_input("Введите число, соответствующее ответу: ")
if   os == "1":
    print "Вы выбрали - Windows 98"
elif os == "2":
    print "Вы выбрали - Windows XP"
elif os == "3":
    print "Вы выбрали - Windows Vista"
elif os == "4":
    print "Вы выбрали — Другая"
elif not os:
    print "Вы не ввели число"
else:
    print "Мы не смогли определить вашу операционную систему"
```

С помощью инструкции `elif` мы можем определить выбранное значение и вывести соответствующее сообщение. Обратите внимание на то, что логическое выражение не содержит операторов сравнения:

```
elif not os:
```

Такая запись эквивалентна следующей:

```
elif os == "":
```

Проверка на равенство выражения значению `True` выполняется по умолчанию. Так как пустая строка интерпретируется как `False`, мы инвертируем возвращаемое значение с помощью оператора `not`.

Один условный оператор можно вложить в другой. В этом случае отступ вложенной инструкции должен быть в два раза больше (листинг 4.3).

Листинг 4.3. Вложенные инструкции

```
# -*- coding: cp1251 -*-
print """Какой операционной системой вы пользуетесь?
1 - Windows 98
2 - Windows XP
3 - Windows Vista
4 - Другая"""
os = raw_input("Введите число, соответствующее ответу: ")
if os != "":
    if   os == "1":
        print "Вы выбрали - Windows 98"
    elif os == "2":
        print "Вы выбрали - Windows XP"
    elif os == "3":
        print "Вы выбрали - Windows Vista"
    elif os == "4":
        print "Вы выбрали — Другая"
    else:
        print "Мы не смогли определить вашу операционную систему"
else:
    print "Вы не ввели число"
```

Начиная с версии Python 2.5, оператор `if...else` имеет еще один формат:

```
<Переменная> = <Если истина> if <Условие> else <Если ложь>
```

Пример:

```
>>> print "Yes" if 10 % 2 == 0 else "No"
Yes
>>> s = "Yes" if 10 % 2 == 0 else "No"
>>> s
'Yes'
```

4.3. Цикл *for*

Предположим, нужно вывести все числа от 1 до 100 по одному на строке. Обычным способом пришлось бы писать 100 строк кода:

```
print 1
print 2
...
print 100
```

При помощи циклов то же действие можно выполнить одной строкой кода:

```
for x in xrange(1, 101): print x
```

Иными словами, циклы позволяют выполнить одни и те же выражения многократно.

В языке Python используются два цикла: `for` и `while`. Цикл `for` применяется для перебора элементов последовательности. Имеет следующий формат:

```
for <Текущий элемент> in <Последовательность>:
    <Выражения внутри цикла>
[else:
    <Блок, выполняемый если не использовался оператор break>
]
```

Здесь присутствуют следующие конструкции:

- `<Последовательность>` — объект, поддерживающий механизм итерации. Например, строка, список, кортеж и др.;
- `<Текущий элемент>` — на каждой итерации через этот параметр доступен текущий элемент последовательности;
- `<Выражения внутри цикла>` — блок, который будет многократно выполняться;
- если внутри цикла не использовался оператор `break`, то после завершения выполнения цикла будет выполнен блок в инструкции `else`. Данный блок не является обязательным.

Пример перебора букв в слове приведен в листинге 4.4.

Листинг 4.4. Перебор букв в слове

```
for s in "str":
    print s,
else:
    print "\nЦикл выполнен"
```

Результат выполнения:

```
s t r
Цикл выполнен
```

Теперь выведем каждый элемент списка и кортежа на отдельной строке (листинг 4.5).

Листинг 4.5. Перебор списка и кортежа

```
for x in [1, 2, 3]:
    print x
for y in (1, 2, 3):
    print y
```

Цикл `for` позволяет также перебрать элементы словарей, хотя словари и не являются последовательностями. В качестве примера выведем элементы словаря

двумя способами. Первый способ использует метод `keys()`, возвращающий список всех ключей словаря. Второй способ доступен в последних версиях Python. В этом случае мы просто указываем словарь в качестве параметра. На каждой итерации цикла будет возвращаться ключ, с помощью которого внутри цикла можно получить значение, соответствующее этому ключу (листинг 4.6).

Листинг 4.6. Перебор элементов словаря

```
arr = {"x": 1, "y": 2, "z": 3}
for key in arr.keys():          # Использование метода keys()
    print key, arr[key]
for key in arr:                 # Словари также поддерживают итерации
    print key, arr[key]
```

С помощью цикла `for` можно перебирать сложные структуры данных. В качестве примера выведем элементы списка кортежей (листинг 4.7).

Листинг 4.7. Перебор элементов списка кортежей

```
arr = [(1, 2), (3, 4)]          # Список кортежей
for a, b in arr:
    print a, b
```

Результат выполнения:

```
1 2
3 4
```

4.4. Функции *range()*, *xrange()* и *enumerate()*

До сих пор мы только выводили элементы последовательностей. Теперь попробуем умножить каждый элемент списка на 2:

```
arr = [1, 2, 3]
for i in arr:
    i = i * 2
print arr          # Результат выполнения: [1, 2, 3]
```

Как видно из примера, список не изменился. Переменная `i` на каждой итерации цикла содержит лишь копию значения текущего элемента списка. Изменить таким способом элементы списка нельзя. Чтобы получить доступ к каждому элементу, можно, например, воспользоваться функцией `range()` для генерации списка с индексами. Функция `range()` имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если параметр `<Начало>` не указан, то по умолчанию используется значение `0`. Во втором параметре указывается

конечное значение. Следует заметить, что это значение не входит в возвращаемый список значений. Если параметр <Шаг> не указан, то используется значение 1. В качестве примера умножим каждый элемент списка на 2 (листинг 4.8).

Листинг 4.8. Пример использования функции `range()`

```
arr = [1, 2, 3]
for i in range(len(arr)):
    arr[i] *= 2
print arr          # Результат выполнения: [2, 4, 6]
```

В этом примере мы получаем количество элементов списка с помощью функции `len()` и передаем результат в функцию `range()`. В итоге функция `range()` возвращает список значений от `0` до `len(arr) - 1`:

```
>>> range(len([1, 2, 3]))
[0, 1, 2]
```

На каждой итерации цикла через переменную `i` доступен текущий элемент из списка индексов. Чтобы получить доступ к элементу списка, указываем индекс внутри квадратных скобок. Умножаем каждый элемент списка на 2, а затем выводим результат с помощью оператора `print`.

Функция `range()` возвращает список, который занимает в памяти компьютера определенное место. Вместо этой функции лучше использовать функцию `xrange()`, которая возвращает не список значений, а итератор. С помощью этого итератора внутри цикла `for` можно получить значение текущего элемента списка. В итоге мы экономим память, т. к. сам список в память не помещается. Функция `xrange()` имеет следующий формат:

```
xrange([<Начало>, ]<Конец>[, <Шаг>])
```

Предназначение параметров такое же, как и в функции `range()`. Переделаем наш пример и используем функцию `xrange()` вместо функции `range()` (листинг 4.9).

Листинг 4.9. Пример использования функции `xrange()`

```
arr = [1, 2, 3]
for i in xrange(len(arr)):
    arr[i] *= 2
print arr
```

Как видно из примера, мы изменили только название функции, а остальной код остался прежним. Функцию `xrange()` удобно использовать и в других случаях. Например, выведем числа от 1 до 100:

```
for i in xrange(1, 101): print i
```

Можно не только увеличивать значение, но и уменьшать его. Выведем все числа от 100 до 1:

```
for i in xrange(100, 0, -1): print i
```

Можно также изменять значение не только на единицу. Выведем все четные числа от 1 до 100:

```
for i in range(2, 101, 2): print i
```

Функция `enumerate(<Объект>)` на каждой итерации цикла `for` возвращает кортеж из индекса и значения текущего элемента. В качестве примера умножим на 2 каждый элемент списка, который содержит четное число (листинг 4.10).

Листинг 4.10. Пример использования функции `enumerate()`

```
arr = [1, 2, 3, 4, 5, 6]
for i, elem in enumerate(arr):
    if elem % 2 == 0:
        arr[i] *= 2
print arr                # Результат выполнения: [1, 4, 3, 8, 5, 12]
```

Функция `enumerate()`, также как и функция `xrange()`, не создает список, а возвращает итератор. С помощью метода `next()` можно обойти всю последовательность. Когда перебор будет закончен, возвращается исключение `StopIteration`:

```
>>> i = enumerate([1, 2])
>>> i
<enumerate object at 0x018CD8A0>
>>> i.next()
(0, 1)
>>> i.next()
(1, 2)
>>> i.next()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    i.next()
StopIteration
```

Весь этот процесс цикл `for` выполняет незаметно для нас.

4.5. Цикл *while*

Выполнение выражений в цикле `while` продолжается до тех пор, пока логическое выражение истинно. Цикл `while` имеет следующий формат:

```
<Начальное значение>
while <Условие>:
    <Выражения>
    <Приращение>
[else:
    <Блок, выполняемый если не использовался оператор break>
]
```

Последовательность работы цикла `while`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие; если оно истинно, выполняются выражения внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре `<Приращение>`.
4. Переход к пункту 2.
5. Если внутри цикла не использовался оператор `break`, то после завершения выполнения цикла будет выполнен блок в инструкции `else`. Данный блок не является обязательным.

Выведем все числа от 1 до 100, используя цикл `while` (листинг 4.11).

Листинг 4.11. Вывод чисел от 1 до 100

```
i = 1                    # <Начальное значение>
while i < 101:           # <Условие>
    print i              # <Выражения>
    i += 1               # <Приращение>
```

Внимание!

Если `<Приращение>` не указано, то цикл будет бесконечным. Чтобы прервать бесконечный цикл, следует нажать комбинацию клавиш <Ctrl>+<C>. В результате генерируется исключение `KeyboardInterrupt`, и выполнение программы будет остановлено. Следует учитывать, что прервать таким образом можно только цикл, который выводит данные.

Выведем все числа от 100 до 1 (листинг 4.12).

Листинг 4.12. Вывод чисел от 100 до 1

```
i = 100
while i:
    print i
    i -= 1
```

Обратите внимание на условие, оно не содержит операторов сравнения. На каждой итерации цикла мы вычитаем единицу из значения переменной-счетчика. Как только значение будет равно `0`, цикл остановится. Как вы уже знаете, число `0` в логическом контексте эквивалентно значению `False`, а проверка на равенство выражения значению `True` выполняется по умолчанию.

С помощью цикла `while` можно перебирать и элементы различных структур. Но в этом случае следует помнить, что цикл `while` работает медленнее цикла `for`. В качестве примера умножим каждый элемент списка на 2 (листинг 4.13).

Листинг 4.13. Перебор элементов списка

```
arr = [1, 2, 3]
i, count = 0, len(arr)
```

```
while i < count:
    arr[i] *= 2
    i += 1
print arr               # Результат выполнения: [2, 4, 6]
```

4.6. Оператор *continue*. Переход на следующую итерацию цикла

Оператор `continue` позволяет перейти к следующей итерации цикла до завершения выполнения всех выражений внутри цикла. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно (листинг 4.14).

Листинг 4.14. Оператор `continue`

```
for i in xrange(1, 101):
    if 4 < i < 11:
        continue            # Переходим на следующую итерацию цикла
    print i
```

4.7. Оператор *break*. Прерывание цикла

Оператор `break` позволяет прервать выполнение цикла досрочно. Для примера выведем все числа от 1 до 100 еще одним способом (листинг 4.15).

Листинг 4.15. Оператор `break`

```
i = 1
while True:
    if i > 100: break       # Прерываем цикл
    print i
    i += 1
```

Здесь мы в условии указали значение `True`. В этом случае выражения внутри цикла будут выполняться бесконечно. Однако использование оператора `break` прерывает его выполнение, как только 100 строк уже напечатано.

Внимание!

Оператор `break` прерывает выполнение цикла, а не программы, т. е. далее будет выполнено выражение, следующее сразу за циклом.

Цикл `while` совместно с оператором `break` удобно использовать для получения неопределенного заранее количества данных от пользователя. В качестве примера просуммируем неопределенное количество чисел (листинг 4.16).

Листинг 4.16. Суммирование неопределенного количества чисел

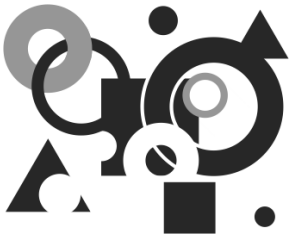
```
# -*- coding: cp1251 -*-
print "Введите слово 'stop' для получения результата"
summa = 0
while True:
    x = raw_input("Введите число: ")
    if x == "stop":
        break                          # Выход из цикла
    x = int(x)                         # Преобразуем строку в число
    summa += x
print "Сумма чисел равна:", summa
```

Процесс ввода трех чисел и получения суммы выглядит так:

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: 20
Введите число: 30
Введите число: stop
Сумма чисел равна: 60
```

Значения, введенные пользователем, выделены полужирным шрифтом.

Глава 5



Числа

Язык Python поддерживает следующие числовые типы:

- `int` — целые числа в диапазоне от –2 147 483 648 до 2 147 483 647. Если число выходит за рамки диапазона, то тип `int` автоматически преобразуется в тип `long`;
- `long` — длинные целые числа. Размер числа ограничен лишь объемом оперативной памяти. В окне **Python Shell** редактора IDLE число, имеющее тип `long`, выводится с буквой `L` в конце;
- `float` — вещественные числа;
- `complex` — комплексные числа.

Операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Целые числа имеют самый простой тип, далее идут длинные целые числа, вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое и вещественное числа, то целое число будет автоматически преобразовано в вещественное число, а затем произведена операция над вещественными числами. Результатом этой операции будет вещественное число.

Создать объект целочисленного типа можно обычным способом:

```
>>> x = 0; y = 10; z = -80
>>> x, y, z
(0, 10, -80)
```

Кроме того, можно указать число в восьмеричной или шестнадцатеричной форме. Такие числа будут автоматически преобразованы в десятичные целые числа. Восьмеричные числа начинаются с нуля и содержат цифры от `0` до `7`:

```
>>> 07, 012, 0777
(7, 10, 511)
```

Начиная с Python 2.6, при вводе восьмеричных чисел после нуля можно указать латинскую букву `o` (регистр не имеет значения):

```
>>> 0o7, 0o12, 0o777, 0O7, 0O12, 0O777
(7, 10, 511, 7, 10, 511)
```

Шестнадцатеричные числа начинаются с комбинации символов `0x` (или `0X`) и могут содержать цифры от `0` до `9` и буквы от `A` до `F` (регистр букв не имеет значения):

```
>>> 0X9,0xA, 0x10, 0xFFF, 0xfff
(9, 10, 16, 4095, 4095)
```

Длинные целые числа создавать специально не нужно. Если целое число выходит за рамки допустимого диапазона, то тип `int` автоматически преобразуется в тип `long`. Чтобы явно создать число типа `long`, после числа следует указать букву `L` (регистр не имеет значения):

```
>>> 10L, 20l
(10L, 20L)
```

Вещественное число может содержать точку и (или) быть представлено в экспоненциальной форме с буквой `E` (регистр не имеет значения):

```
>>> 10., .14, 3.14
(10.0, 0.14000000000000001, 3.1400000000000001)
>>> 11E20, 2.5e-12
(1.1e+21, 2.4999999999999998e-12)
```

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей операции может показаться странным:

```
>>> 0.3 - 0.1 - 0.1 - 0.1
-2.7755575615628914e-17
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат. Если необходимо производить операции с фиксированной точностью, то следует использовать модуль `decimal`.

Комплексные числа записываются в формате:

```
<Вещественная часть>+<Мнимая часть>J
```

Примеры комплексных чисел:

```
>>> 2+5J, 8j
((2+5j), 8j)
```

5.1. Встроенные функции для работы с числами

Для работы с числами предназначены следующие встроенные функции:

- `int([<Объект>[, <Система счисления>]])` — преобразует объект в целое число. Во втором параметре можно указать систему счисления (значение по умолчанию 10). Пример:

  ```
  >>> int(7.5), int("71", 10), int("071", 8), int("0xA", 16)
  (7, 71, 57, 10)
  >>> int()
  0
  ```

- `long([<Объект>[, <Система счисления>]])` — преобразует объект в длинное целое число. Во втором параметре можно указать систему счисления (значение по умолчанию 10). Пример:

```
>>> long(7.5), long("71", 10), long("071", 8), long("0xA", 16)
(7L, 71L, 57L, 10L)
>>> long()
0L
```

- `float([<Число или строка>])` — преобразует целое число или строку в вещественное число:

```
>>> float(7), float("7.1"), float("12.")
(7.0, 7.0999999999999996, 12.0)
>>> float()
0.0
```

- `oct(<Число>)` — преобразует десятичное число в восьмеричное. Возвращает строковое представление числа. Пример:

```
>>> oct(7), oct(8), oct(64)
('07', '010', '0100')
```

- `hex(<Число>)` — преобразует десятичное число в шестнадцатеричное. Возвращает строковое представление числа. Пример:

```
>>> hex(10), hex(16), hex(255)
('0xa', '0x10', '0xff')
```

- `round(<Число>[, <Количество знаков после точки>])` — возвращает вещественное число, округленное до ближайшего меньшего целого для чисел с дробной частью меньше `0.5`, или значение, округленное до ближайшего большего целого для чисел с дробной частью больше или равной `0.5`. Пример:

```
>>> round(1), round(1.49), round(1.50)
(1.0, 1.0, 2.0)
```

Во втором параметре можно указать количество знаков в дробной части. Если параметр не указан, то используется значение `0`:

```
>>> round(1.524, 2), round(1.525, 2), round(1.5555, 3)
(1.52, 1.53, 1.556)
```

- `abs(<Число>)` — возвращает абсолютное значение:

```
>>> abs(-10), abs(10), abs(-12.5)
(10, 10, 12.5)
```

- `pow(<Число>, <Степень>[, <Остаток от деления>])` — возводит `<Число>` в `<Степень>`:

```
>>> pow(10, 2), 10 ** 2, pow(3, 3), 3 ** 3
(100, 100, 27, 27)
```

Если указан третий параметр, то возвращается остаток от деления:

```
>>> pow(10, 2, 2), (10 ** 2) % 2, pow(3, 3, 2), (3 ** 3) % 2
(0, 0, 1, 1)
```

- `max(<Список чисел через запятую>)` — максимальное значение из списка:
```
>>> max(1, 2, 3), max(3, 2, 3, 1), max(1, 1.0), max(1.0, 1)
(3, 3, 1, 1.0)
```
- `min(<Список чисел через запятую>)` — минимальное значение из списка:
```
>>> min(1, 2, 3), min(3, 2, 3, 1), min(1, 1.0), min(1.0, 1)
(1, 1, 1, 1.0)
```
- `cmp(<Объект1>, <Объект2>)` — сравнивает два объекта и возвращает следующие значения:
  - `1` — если `<Объект1>` больше `<Объект2>`;
  - `-1` — если `<Объект1>` меньше `<Объект2>`;
  - `0` — если значения равны.

Пример сравнения чисел:
```
>>> cmp(10, 5), cmp(5, 10), cmp(5, 5), cmp(5, 5.0)
(1, -1, 0, 0)
```
- `sum(<Последовательность>[, <Начальное значение>])` — возвращает сумму значений элементов последовательности (например, списка, кортежа) плюс `<Начальное значение>`. Если второй параметр не указан, то значение параметра равно `0`. Если последовательность пустая, то возвращается значение второго параметра. Примеры:
```
>>> sum((10, 20, 30, 40)), sum([10, 20, 30, 40])
(100, 100)
>>> sum([10, 20, 30, 40], 2), sum([], 2)
(102, 2)
```
- `divmod(x, y)` — возвращает кортеж из двух значений `(x // y, x % y)`:
```
>>> divmod(13, 2)              # 13 == 6 * 2 + 1
(6, 1)
>>> 13 // 2, 13 % 2
(6, 1)
>>> divmod(13.5, 2.0)          # 13.5 == 6.0 * 2.0 + 1.5
(6.0, 1.5)
>>> 13.5 // 2.0, 13.5 % 2.0
(6.0, 1.5)
```

5.2. Модуль *math*. Математические функции

Модуль `math` предоставляет дополнительные функции для работы с числами, а также стандартные константы. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:
```
import math
```

Примечание

Для работы с комплексными числами необходимо использовать модуль `cmath`.

Модуль `math` предоставляет следующие стандартные константы:

- `pi` — возвращает число π:

```
>>> import math
>>> math.pi
3.1415926535897931
```

- `e` — возвращает значение константы *e*:

```
>>> math.e
2.7182818284590451
```

Перечислим основные функции для работы с числами:

- `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус, косинус, тангенс). Значение указывается в радианах;
- `asin()`, `acos()`, `atan()` — обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах;
- `degrees()` — преобразует радианы в градусы:

```
>>> math.degrees(math.pi)
180.0
```

- `radians()` — преобразует градусы в радианы:

```
>>> math.radians(180.0)
3.1415926535897931
```

- `exp()` — экспонента;
- `log()` — логарифм;
- `sqrt()` — квадратный корень:

```
>>> math.sqrt(100), math.sqrt(25)
(10.0, 5.0)
```

- `ceil()` — значение, округленное до ближайшего большего целого:

```
>>> math.ceil(5.49), math.ceil(5.50), math.ceil(5.51)
(6.0, 6.0, 6.0)
```

- `floor()` — значение, округленное до ближайшего меньшего целого:

```
>>> math.floor(5.49), math.floor(5.50), math.floor(5.51)
(5.0, 5.0, 5.0)
```

- `pow(<Число>, <Степень>)` — возводит `<Число>` в `<Степень>`:

```
>>> math.pow(10, 2), 10 ** 2, math.pow(3, 3), 3 ** 3
(100.0, 100, 27.0, 27)
```

- `fabs()` — абсолютное значение:

```
>>> math.fabs(10), math.fabs(-10), math.fabs(-12.5)
(10.0, 10.0, 12.5)
```

- `fmod()` — остаток от деления:

```
>>> math.fmod(10, 5), 10 % 5, math.fmod(10, 3), 10 % 3
(0.0, 0, 1.0, 1)
```

- `factorial()` — факториал числа. Функция доступна, начиная с версии 2.6. Пример:

```
>>> math.factorial(5), math.factorial(6)
(120, 720)
```

Примечание

В этом разделе мы рассмотрели только основные функции. Чтобы получить полный список функций, обращайтесь к документации по модулю `math`.

5.3. Модуль *random*. Генерация случайных чисел

Модуль `random` позволяет генерировать случайные числа. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:

```
import random
```

Перечислим основные функции:

- `random()` — возвращает псевдослучайное число от `0.0` до `1.0`:

```
>>> import random
>>> random.random()
0.42888905467511462
>>> random.random()
0.57809130113447038
>>> random.random()
0.20609823213950174
```

- `seed(<Параметр>)` — настраивает генератор случайных чисел на новую последовательность. По умолчанию используется системное время. Если значение параметра будет одинаковым, то генерируется одинаковое число:

```
>>> random.seed(10)
>>> random.random()
0.5714025946899135
>>> random.seed(10)
>>> random.random()
0.5714025946899135
```

- `uniform(<Начало>, <Конец>)` — возвращает псевдослучайное вещественное число в диапазоне от `<Начало>` до `<Конец>`:

```
>>> random.uniform(0, 10)
1.6022955651881965
>>> random.uniform(0, 10)
5.206693596399246
```

- `randint(<Начало>, <Конец>)` — возвращает псевдослучайное целое число в диапазоне от `<Начало>` до `<Конец>`:

```
>>> random.randint(0, 10)
10
>>> random.randint(0, 10)
6
```

- `randrange([<Начало>, ]<Конец>[, <Шаг>])` — возвращает случайный элемент из числовой последовательности. Параметры аналогичны параметрам функции `range()`:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Именно из списка, возвращаемого функцией `range()`, и выбирается случайный элемент:

```
>>> random.randrange(10)
9
>>> random.randrange(0, 10)
1
>>> random.randrange(0, 10, 2)
8
```

- `choice(<Последовательность>)` — возвращает случайный элемент из любой последовательности (строки, списка, кортежа):

```
>>> random.choice("string")          # Строка
't'
>>> random.choice(["s", "t", "r"])   # Список
's'
>>> random.choice(("s", "t", "r"))   # Кортеж
'r'
```

- `shuffle(<Список>[, <Число от 0.0 до 1.0>])` — перемешивает элементы списка случайным образом. Функция перемешивает сам список и ничего не возвращает. Если второй параметр не указан, то используется значение, возвращаемое функцией `random()`. Пример:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)
>>> arr
[10, 7, 8, 6, 1, 4, 3, 5, 2, 9]
```

- `sample(<Последовательность>, <Количество элементов>)` — возвращает список из указанного количества элементов. В этот список попадут элементы из последовательности, выбранные случайным образом. В качестве последовательности можно указать любые объекты, поддерживающие итерации. Примеры:

```
>>> random.sample("string", 2)
['i', 'r']
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 2)
```

```
[7, 10]
>>> arr # Сам список не изменяется
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample((1, 2, 3, 4, 5, 6, 7), 3)
[6, 3, 5]
>>> random.sample(xrange(300), 5)
[126, 194, 272, 46, 71]
```

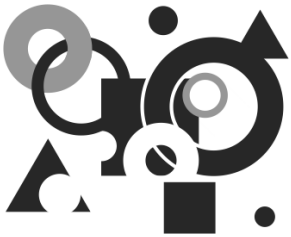
Для примера создадим генератор паролей произвольной длины (листинг 5.1). Для этого добавляем в список `arr` все разрешенные символы, а далее в цикле получаем случайный элемент с помощью функции `choice()`. По умолчанию будет выдаваться пароль из 8 символов.

Листинг 5.1. Генератор паролей

```
# -*- coding: cp1251 -*-
import random # Подключаем модуль random
def passw_generator(count_char=8):
    arr = ['a','b','c','d','e','f','g','h','i','j','k','l','m',
    'n','o','p','q','r','s','t','u','v','w','x','y','z',
    'A','B','C','D','E','F','G','H','I','J','K','L',
    'M','N','O','P','Q','R','S','T','U','V', 'W',
    'X','Y','Z','1','2','3','4','5','6','7','8','9','0']
    passw = []
    for i in xrange(count_char):
        passw.append(random.choice(arr))
    return "".join(passw)

# Вызываем функцию
print passw_generator(10)  # Выведет что-то вроде rPiK6lvemm
print passw_generator()    # Выведет что-то вроде f04CURGA
```

Глава 6



Строки

Строки являются упорядоченными последовательностями символов. Длина строки ограничена лишь объемом оперативной памяти компьютера. Как и все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`).

Кроме того, строки относятся к неизменяемым типам данных. Поэтому практически все строковые методы в качестве значения возвращают новую строку. При использовании небольших строк это не приводит к каким-либо проблемам, но при работе с большими строками можно столкнуться с проблемой нехватки памяти. Иными словами, можно получить символ по индексу, но изменить его нельзя (листинг 6.1).

Листинг 6.1. Попытка изменить символ по индексу

```
>>> s = "Python"
>>> s[0]                    # Можно получить символ по индексу
'P'
>>> s[0] = "J"              # Изменить строку нельзя
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    s[0] = "J"              # Изменить строку нельзя
TypeError: 'str' object does not support item assignment
```

В некоторых языках программирования концом строки является нулевой символ. В языке Python нулевой символ может быть расположен внутри строки:

```
>>> "string\x00string" # Нулевой символ — это НЕ конец строки
'string\x00string'
```

Язык Python поддерживает следующие строковые типы:

- `str` — обычная строка. Может содержать как однобайтовые символы, так и многобайтовые. Обратите внимание на то, что функции и методы обычных строк некорректно работают с многобайтовыми кодировками, например, функ-

ция `len()` вернет количество байтов, а не символов. При работе с многобайтовыми кодировками следует использовать Unicode-строки. В этом случае функция `len()` вернет количество символов, а не байтов;

- `unicode` — Unicode-строка. Необходимо заметить, что Unicode-строки не имеют никакого отношения к кодировке UTF-8. Рассматривайте такие строки, как строки в некой абстрактной кодировке, позволяющие хранить символы Unicode и производить манипуляции с ними. При выводе Unicode-строку необходимо преобразовать в обычную строку в какой-либо кодировке.

6.1. Создание строки

Создать обычную строку можно следующими способами:

- с помощью функции `str([<Объект>])`. Функция позволяет преобразовать любой объект в строку. Если параметр не указан, то возвращается пустая строка. Пример:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
```

- указав строку между апострофами или двойными кавычками:

```
>>> 'string', "string", '"x": 5', "'x': 5"
('string', 'string', '"x": 5', "'x': 5")
>>> print 'Строка1\nСтрока2'
Строка1
Строка2
>>> print "Строка1\nСтрока2"
Строка1
Строка2
```

В некоторых языках программирования (например, в PHP) строка в апострофах отличается от строки в кавычках тем, что внутри апострофов специальные символы выводятся как есть, а внутри кавычек спецсимволы интерпретируются. В языке Python никакого отличия между строкой в апострофах и строкой в кавычках нет. Это одно и то же. Если строка содержит кавычки, то ее лучше заключить в апострофы и наоборот. Все специальные символы в таких строках интерпретируются. Например, последовательность символов `\n` преобразуется в символ новой строки. Чтобы специальный символ выводился как есть, его необходимо экранировать с помощью слеша:

```
>>> print "Строка1\\nСтрока2"
Строка1\nСтрока2
>>> print 'Строка1\\nСтрока2'
Строка1\nСтрока2
```

Кавычку внутри строки в кавычках и апостроф внутри строки в апострофах также необходимо экранировать с помощью защитного слеша:

```
>>> "\"x\": 5", '\'x\': 5'
('"x": 5', "'x': 5")
```

Следует также заметить, что заключить объект в одинарные кавычки (или апострофы) на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку:

```
>>> "string
SyntaxError: EOL while scanning single-quoted string
```

Чтобы расположить объект на нескольких строках, следует перед символом перевода строки указать символ \, поместить две строки внутри скобок или использовать конкатенацию внутри скобок:

```
>>> "string1\
string2" # После символа \ не должно быть никаких символов
'string1string2'
>>> ("string1"
"string2") # Неявная конкатенация строк
'string1string2'
>>> ("string1" +
"string2") # Явная конкатенация строк
'string1string2'
```

Кроме того, если в конце строки расположен символ \, то его необходимо экранировать, иначе будет выведено сообщение об ошибке:

```
>>> print "string\"
SyntaxError: EOL while scanning single-quoted string
>>> print "string\\"
string\
```

- указав строку между утроенными апострофами или утроенными кавычками. Такие объекты можно разместить на нескольких строках, а также одновременно использовать кавычки и апострофы без необходимости их экранировать. В остальном такие объекты эквивалентны строкам в апострофах и кавычках. Все специальные символы в таких строках интерпретируются. Примеры:

```
>>> print '''Строка1
Строка2'''
Строка1
Строка2
>>> print """Строка1
Строка2"""
Строка1
Строка2
```

Если строка не присваивается переменной, то она считается строкой документирования. Такая строка сохраняется в свойстве __doc__ того объекта, в котором расположена. В качестве примера создадим функцию со строкой документирования, а затем выведем содержимое строки:

```
>>> def test():
        """Это описание функции"""
        pass
>>> print test.__doc__
Это описание функции
```

Так как выражения внутри таких строк не выполняются, то утроенные кавычки (или утроенные апострофы) очень часто используются для комментирования больших фрагментов кода на этапе отладки программы.

Если перед строкой разместить модификатор `r`, то специальные символы внутри строки выводятся как есть. Например, символ `\n` не будет преобразован в символ перевода строки. Иными словами, он будет считаться последовательностью двух символов: `\` и `n`:

```
>>> print "Строка1\nСтрока2"
Строка1
Строка2
>>> print r"Строка1\nСтрока2"
Строка1\nСтрока2
>>> print r"""Строка1\nСтрока2"""
Строка1\nСтрока2
```

Такие неформатированные строки удобно использовать в шаблонах регулярных выражений, а также при указании пути к файлу или каталогу:

```
>>> print r"C:\Python26\lib\site-packages"
C:\Python26\lib\site-packages
```

Если модификатор не указать, то все слеши в пути необходимо экранировать:

```
>>> print "C:\\Python26\\lib\\site-packages"
C:\Python26\lib\site-packages
```

Если в конце неформатированной строки расположен слеш, то его необходимо экранировать. Однако следует учитывать, что этот слеш будет добавлен в исходную строку. Пример:

```
>>> print r"C:\Python26\lib\site-packages\"
SyntaxError: EOL while scanning string literal
>>> print r"C:\Python26\lib\site-packages\\"
C:\Python26\lib\site-packages\\
```

Создать Unicode-строку можно с помощью функции `unicode()`. Формат функции:

```
unicode([<Строка>[, <Кодировка>[, <Обработка ошибок>]]])
```

Если параметры не указаны, то возвращается пустая Unicode-строка. В третьем параметре могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`. Примеры:

```
>>> unicode()
u''
>>> unicode("Строка в кодировке windows-1251", "cp1251")
u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432
\u043a\u043e\u0434\u0438\u0440\u043e\u0432\u043a\u0435 windows-1251'
>>> unicode("Строка", "utf-8", "strict")
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    unicode("Строка", "utf-8", "strict")
```

```
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 0-1:
invalid data
>>> unicode("Строка", "utf-8", "replace")
u'\ufffd\ufffd'
>>> unicode("Строка", "utf-8", "ignore")
u''
```

Если перед кавычками (или апострофами) разместить модификатор `u`, то также будет создана Unicode-строка. Чтобы создать неформатированную строку, необходимо после модификатора `u` указать модификатор `r`:

```
# -*- coding: utf-8 -*-
print u"Строка1\nСтрока2"        # Unicode-строка
print ur"Строка1\nСтрока2"       # Неформатированная Unicode-строка
```

Результат выполнения:

```
Строка1
Строка2
Строка1\nСтрока2
```

Необходимо заметить, что в окне **Python Shell** редактора IDLE модификатор `u` обрабатывается некорректно. Поэтому в этом окне лучше использовать функцию `unicode()`. Пример:

```
>>> str1 = u"Строка"
>>> str1                        # Строка преобразована некорректно!
u'\xd1\xf2\xf0\xee\xea\xe0'
>>> str2 = unicode("Строка", "cp1251")
>>> str2                        # Строка преобразована правильно
u'\u0421\u0442\u0440\u043e\u043a\u0430'
>>> str1.encode("cp1251")       # Ошибка при преобразовании
... Фрагмент опущен ...
UnicodeEncodeError: 'charmap' codec can't encode characters in
position 0-5: character maps to <undefined>
>>> str2.encode("cp1251")       # Преобразование без ошибок
'\xd1\xf2\xf0\xee\xea\xe0'
```

6.2. Специальные символы

Специальные символы — это комбинации знаков, обозначающих служебные или непечатаемые символы, которые невозможно вставить обычным способом. Перечислим специальные символы, допустимые внутри строки, перед которой нет модификатора `r`:

- `\n` — перевод строки;
- `\r` — возврат каретки;
- `\t` — знак табуляции;
- `\v` — вертикальная табуляция;

- `\a` — звонок;
- `\b` — забой;
- `\f` — перевод формата;
- `\0` — нулевой символ (не является концом строки);
- `\"` — кавычка;
- `\'` — апостроф;
- `\0xx` — восьмеричное значение символа. Например, `\074` соответствует символу <;
- `\xhh` — шестнадцатеричное значение символа. Например, `\x6a` соответствует символу `j`;
- `\\` — обратный слеш.

В Unicode-строках можно указать следующие специальные символы:

- `\uxxxx` — 16-битный символ Unicode. Например, `\u043a` соответствует русской букве к;
- `\Uxxxxxxxx` — 32-битный символ Unicode.

Если после слеша не стоит символ, который вместе со слешем интерпретируется как спецсимвол, то слеш сохраняется в составе строки:

```
>>> print "Этот символ \не специальный"
Этот символ \не специальный
```

Тем не менее, лучше экранировать слеш явным образом:

```
>>> print "Этот символ \\не специальный"
Этот символ \не специальный
```

6.3. Операции над строками

Как вы уже знаете, строки относятся к последовательностями. Как и все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение. Рассмотрим эти операции подробно.

К любому символу строки можно обратиться как к элементу списка. Достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля:

```
>>> s = "Python"
>>> s[0], s[1], s[2], s[3], s[4], s[5]
('P', 'y', 't', 'h', 'o', 'n')
```

Если символ, соответствующий указанному индексу, отсутствует в строке, то возбуждается исключение `IndexError`:

```
>>> s = "Python"
>>> s[10]
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    s[10]
IndexError: string index out of range
```

В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца строки, а точнее, значение вычитается из длины строки, чтобы получить положительный индекс:

```
>>> s = "Python"
>>> s[-1], s[len(s)-1]
('n', 'n')
```

Так как строки относятся к неизменяемым типам данных, то изменить символ по индексу нельзя:

```
>>> s = "Python"
>>> s[0] = "J"                          # Изменить строку нельзя
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    s[0] = "J"                         # Изменить строку нельзя
TypeError: 'str' object does not support item assignment
```

Чтобы выполнить изменение, можно воспользоваться операцией извлечения среза, которая возвращает указанный фрагмент строки. Формат операции:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры являются необязательными. Если параметр `<Начало>` не указан, то используется значение `0`. Если параметр `<Конец>` не указан, то возвращается фрагмент до конца строки. Следует также заметить, что символ с индексом, указанным в этом параметре, не входит в возвращаемый фрагмент. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве значения параметров можно указать отрицательные значения.

Теперь рассмотрим несколько примеров. Сначала получим копию строки:

```
>>> s = "Python"
>>> s[:] # Возвращается фрагмент от позиции 0 до конца строки
'Python'
```

Теперь выведем символы в обратном порядке:

```
>>> s[::-1] # Указываем отрицательное значение в параметре <Шаг>
'nohtyP'
```

Заменим первый символ в строке:

```
>>> "J" + s[1:] # Извлекаем фрагмент от символа 1 до конца строки
'Jython'
```

Удалим последний символ:

```
>>> s[:-1] # Возвращается фрагмент от 0 до len(s)-1
'Pytho'
```

Получим первый символ в строке:

```
>>> s[0:1] # Символ с индексом 1 не входит в диапазон
'P'
```

А теперь получим последний символ:

```
>>> s[-1:] # Получаем фрагмент от len(s)-1 до конца строки
'n'
```

И, наконец, выведем символы с индексами 2, 3 и 4:

```
>>> s[2:5] # Возвращаются символы с индексами 2, 3 и 4
'tho'
```

Узнать количество символов в строке позволяет функция `len()`:

```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
(6, 3, 6)
```

Необходимо заметить, что для многобайтовых кодировок функция `len()` возвращает количество байтов, а не количество символов. Чтобы получить длину строки в символах, следует использовать Unicode-строки. Пример:

```
>>> len("Строка")                                    # Правильно
6
>>> len(unicode("Строка", "cp1251").encode("utf-8")) # Неправильно
12
>>> len(unicode("Строка", "cp1251"))                 # Правильно
6
```

Теперь, когда мы знаем количество символов, можно перебрать все символы с помощью цикла `for`:

```
>>> s = "Python"
>>> for i in xrange(len(s)): print s[i],
```

Результат выполнения:

```
P y t h o n
```

Так как строки поддерживают итерации, мы можем просто указать строку в качестве параметра цикла:

```
>>> s = "Python"
>>> for i in s: print i,
```

Результат выполнения будет таким же:

```
P y t h o n
```

Соединить две строки в одну строку позволяет оператор `+`:

```
>>> print "Строка1" + "Строка2"
Строка1Строка2
```

Кроме того, можно выполнить неявную конкатенацию строк. В этом случае две строки указываются рядом без оператора между ними:

```
>>> print "Строка1" "Строка2"
Строка1Строка2
```

Обратите внимание на то, что если между строками указать запятую, то мы получим кортеж, а не строку:

```
>>> s = "Строка1", "Строка2"
>>> type(s)                        # Получаем кортеж, а не строку
<type 'tuple'>
```

Если соединяются, например, переменная и строка, то следует обязательно указывать символ конкатенации строк, иначе будет выведено сообщение об ошибке:

```
>>> s = "Строка1"
>>> print s + "Строка2"            # Нормально
Строка1Строка2
>>> print s "Строка2"              # Ошибка
SyntaxError: invalid syntax
```

При необходимости соединить строку с другим типом данных (например, с числом) следует произвести явное преобразование типов с помощью функции `str()`:

```
>>> "string" + str(10)
'string10'
```

Кроме рассмотренных операций, строки поддерживают операцию повторения и проверку на вхождение. Повторить строку указанное количество раз можно с помощью оператора `*`, а выполнить проверку на вхождение фрагмента в строку позволяет оператор `in`:

```
>>> "-" * 20
'--------------------'
>>> "yt" in "Python"               # Найдено
True
>>> "yt" in "Perl"                 # Не найдено
False
```

6.4. Форматирование строк

Вместо соединения строк с помощью оператора `+` лучше использовать форматирование. Данная операция позволяет соединять строку с любым другим типом данных и выполняется быстрее конкатенации. Форматирование имеет следующий синтаксис:

```
<Строка специального формата> % <Значения>
```

Внутри параметра `<Строка специального формата>` могут быть указаны спецификаторы, имеющие следующий синтаксис:

```
%[(<Ключ>)][<Флаг>][<Ширина>][.<Точность>]<Тип преобразования>
```

Количество спецификаторов внутри строки должно быть равно количеству элементов в параметре `<Значения>`. Если используется только один спецификатор, то параметр `<Значения>` может содержать одно значение, в противном случае необходимо перечислить значения через запятую внутри круглых скобок, создавая тем самым кортеж. Пример:

```
>>> "%s" % 10                          # Один элемент
'10'
>>> "%s - %s - %s" % (10, 20, 30)      # Несколько элементов
'10 - 20 - 30'
```

Параметры внутри спецификатора имеют следующий смысл:

- `<Ключ>` — ключ словаря. Если задан ключ, то в параметре `<Значения>` необходимо указать словарь, а не кортеж. Пример:

```
>>> "%(name)s - %(year)s" % {"year": 1978, "name": "Nik"}
'Nik - 1978'
```

- `<Флаг>` — флаг преобразования. Может содержать следующие значения:
  - `#` — для восьмеричных значений добавляет в начало символ `0`, для шестнадцатеричных значений добавляет комбинацию символов `0x` (если используется тип `x`) или `0X` (если используется тип `X`), для вещественных чисел указывает всегда выводить дробную точку, даже если задано значение `0` в параметре `<Точность>`:

```
>>> print "%#o %#o %#o" % (077, 10, 10.5)
077 012 012
>>> print "%#x %#x %#x" % (0xff, 10, 10.5)
0xff 0xa 0xa
>>> print "%#X %#X %#X" % (0xff, 10, 10.5)
0XFF 0XA 0XA
>>> print "%#.0F %.0F" % (300, 300)
300. 300
```

  - `0` — задает наличие ведущих нулей для числового значения:

```
>>> "'%d' - '%05d'" % (3, 3) # 5 — ширина поля
"'3' - '00003'"
```

  - `-` — задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе. Если флаг `-` указан одновременно с флагом `0`, то действие флага `0` будет отменено. Пример:

```
>>> "'%5d' - '%-5d'" % (3, 3) # 5 — ширина поля
"'    3' - '3    '"
>>> "'%05d' - '%-05d'" % (3, 3)
"'00003' - '3    '"
```

  - `пробел` — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Пример:

```
>>> "'% d' - '% d'" % (-3, 3)
"'-3' - ' 3'"
```

  - `+` — задает обязательный вывод знака, как для отрицательных, так и для положительных чисел. Если флаг `+` указан одновременно с флагом `пробел`, то действие флага `пробел` будет отменено. Пример:

```
>>> "'%+d' - '%+d'" % (-3, 3)
"'-3' - '+3'"
```

- `<Ширина>` — минимальная ширина поля. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

```
>>> "'%10d' - '%-10d'" % (3, 3)
"'         3' - '3         '"
>>> "'%3s''%10s'" % ("string", "string")
"'string''    string'"
```

Вместо значения можно указать символ "*". В этом случае значение следует задать внутри кортежа:

```
>>> "'%*s''%10s'" % (10, "string", "str")
"'    string''       str'"
```

- <Точность> — количество знаков после точки для вещественных чисел. Перед этим параметром обязательно должна стоять точка. Пример:

```
>>> import math
>>> "%s %f %.2f" % (math.pi, math.pi, math.pi)
'3.14159265359 3.141593 3.14'
```

Вместо значения можно указать символ "*". В этом случае значение следует задать внутри кортежа:

```
>>> "'%*.*f'" % (8, 5, math.pi)
"' 3.14159'"
```

- <Тип преобразования> — задает тип преобразования. Параметр является обязательным.

В параметре <Тип преобразования> могут быть указаны следующие символы:

- s — преобразует любой объект в строку с помощью функции str():

```
>>> print "%s" % ("Обычная строка")
Обычная строка
>>> print "%s %s %s" % (10, 10.52, [1, 2, 3])
10 10.52 [1, 2, 3]
```

- r — преобразует любой объект в строку с помощью функции repr():

```
>>> print "%r" % ("Обычная строка")
'\xce\xe1\xfb\xf7\xed\xe0\xff \xf1\xf2\xf0\xee\xea\xe0'
```

- c — выводит одиночный символ или преобразует числовое значение в символ. В качестве примера выведем числовое значение и соответствующий этому значению символ:

```
>>> for i in xrange(33, 127): print "%s => %c" % (i, i)
```

- d и i — возвращают целую часть числа:

```
>>> print "%d %d %d" % (10, 25.6, -80)
10 25 -80
>>> print "%i %i %i" % (10, 25.6, -80)
10 25 -80
```

- o — восьмеричное значение:

```
>>> print "%o %o %o" % (077, 10, 10.5)
77 12 12
>>> print "%#o %#o %#o" % (077, 10, 10.5)
077 012 012
```

- x — шестнадцатеричное значение в нижнем регистре:

```
>>> print "%x %x %x" % (0xff, 10, 10.5)
ff a a
>>> print "%#x %#x %#x" % (0xff, 10, 10.5)
0xff 0xa 0xa
```

- `X` — шестнадцатеричное значение в верхнем регистре:

```
>>> print "%X %X %X" % (0xff, 10, 10.5)
FF A A
>>> print "%#X %#X %#X" % (0xff, 10, 10.5)
0XFF 0XA 0XA
```

- `f` и `F` — вещественное число в десятичном представлении:

```
>>> print "%f %f %f" % (300, 18.65781452, -12.5)
300.000000 18.657815 -12.500000
>>> print "%F %F %F" % (300, 18.65781452, -12.5)
300.000000 18.657815 -12.500000
>>> print "%#.0F %.0F" % (300, 300)
300. 300
```

- `e` — вещественное число в экспоненциальной форме (буква "e" в нижнем регистре):

```
>>> print "%e %e" % (3000, 18657.81452)
3.000000e+03 1.865781e+04
```

- `E` — вещественное число в экспоненциальной форме (буква "e" в верхнем регистре):

```
>>> print "%E %E" % (3000, 18657.81452)
3.000000E+03 1.865781E+04
```

- `g` — эквивалентно `e`, если экспонента меньше `-4`. В противном случае используется десятичное представление вещественного числа:

```
>>> print "%g %g %g" % (0.086578, 0.000086578, 1.865E-005)
0.086578 8.6578e-05 1.865e-05
```

- `G` — эквивалентно `E`, если экспонента меньше `-4`. В противном случае используется десятичное представление вещественного числа:

```
>>> print "%G %G %G" % (0.086578, 0.000086578, 1.865E-005)
0.086578 8.6578E-05 1.865E-05
```

Если внутри строки необходимо использовать символ процента, то этот символ следует удвоить, иначе будет выведено сообщение об ошибке:

```
>>> print "% %s" % ("- это символ процента") # Ошибка
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    print "% %s" % ("- это символ процента") # Ошибка
TypeError: not all arguments converted during string formatting
>>> print "%% %s" % ("- это символ процента") # Нормально
% - это символ процента
```

Форматирование строк очень удобно использовать при передаче данных в шаблон HTML-страницы. Для этого заполняем словарь данными и указываем его справа от символа `%`, а сам шаблон — слева. Продемонстрируем это на примере (листинг 6.2).

Листинг 6.2. Пример использования форматирования строк

```
# -*- coding: cp1251 -*-
html = """<html>
<head><title>%(title)s</title>
</head>
<body>
<h1>%(h1)s</h1>
<div>%(content)s</div>
</body>
</html>"""
arr = {"title": "Это название документа",
       "h1": "Это заголовок первого уровня",
       "content": "Это основное содержание страницы"}
print html % arr # Подставляем значения и выводим шаблон
```

Результат выполнения:

```
<html>
<head><title>Это название документа</title>
</head>
<body>
<h1>Это заголовок первого уровня</h1>
<div>Это основное содержание страницы</div>
</body>
</html>
```

При использовании операции форматирования следует учитывать, что если в операции участвуют обычная строка и Unicode-строка, то результатом будет Unicode-строка:

```
>>> type("%s %s" % (unicode("str", "cp1251"), "str"))
<type 'unicode'>
```

При использовании смешанных строк производится попытка преобразовать обычную строку в Unicode-строку. Так как по умолчанию в системе используется кодировка ASCII, попытка преобразовать обычную строку (содержащую русские буквы) в Unicode-строку приведет к исключению `UnicodeDecodeError`:

```
>>> "%s %s" % (unicode("str", "cp1251"), "Строка")
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    "%s %s" % (unicode("str", "cp1251"), "Строка")
UnicodeDecodeError: 'ascii' codec can't decode byte 0xd1 in position 0:
ordinal not in range(128)
```

Преобразованию подвергаются обычные строки, переданные в качестве параметров, а также сама строка со специальными символами. Если в строке со специ-

альными символами содержатся русские буквы, то это также приведет к исключению `UnicodeDecodeError`:

```
>>> "%s %s строка" % (unicode("str", "cp1251"), "str")
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    "%s %s строка" % (unicode("str", "cp1251"), "str")
UnicodeDecodeError: 'ascii' codec can't decode byte 0xf1 in position 6:
ordinal not in range(128)
```

Чтобы исключение не возбуждалось, необходимо явно преобразовывать строки к одному типу данных. Например, преобразовать Unicode-строку в обычную строку можно так:

```
>>> s = unicode("Строка1", "cp1251")
>>> print "%s %s" % (s.encode("cp1251"), "Строка2")
Строка1 Строка2
```

Более подробно преобразование кодировок мы рассмотрим далее в этой главе.

Для форматирования строк можно также использовать следующие методы:

- `expandtabs([<Ширина поля>])` — заменяет символ табуляции пробелами таким образом, чтобы общая ширина фрагмента вместе с текстом (расположенным перед символом табуляции) была равна указанной величине. Если параметр не указан, то ширина поля предполагается равной 8 символам. Пример:

  ```
  >>> s = "1\t12\t123\t"
  >>> "'%s'" % s.expandtabs(4)
  "'1   12  123 '"
  ```

  В этом примере ширина задана равной четырем символам. Поэтому во фрагменте `"1\t"` табуляция будет заменена тремя пробелами, во фрагменте `"12\t"` — двумя пробелами, а во фрагменте `"123\t"` — одним пробелом. Во всех трех фрагментах ширина будет равна четырем символам.

  Если перед символом табуляции нет текста или количество символов перед табуляцией равно ширине, то табуляция заменяется указанным количеством пробелов:

  ```
  >>> s = "\t"
  >>> "'%s' - '%s'" % (s.expandtabs(), s.expandtabs(4))
  "'        ' - '    '"
  >>> s = "1234\t"
  >>> "'%s'" % s.expandtabs(4)
  "'1234    '"
  ```

  Если количество символов перед табуляцией больше ширины, то табуляция заменяется пробелами таким образом, чтобы ширина фрагмента вместе с текстом делилась без остатка на указанную ширину:

  ```
  >>> s = "12345\t123456\t1234567\t1234567890\t"
  >>> "'%s'" % s.expandtabs(4)
  "'12345   123456  1234567 1234567890  '"
  ```

Таким образом, если количество символов перед табуляцией больше 4, но менее 8, то фрагмент дополняется пробелами до 8 символов. Если количество символов больше 8, но менее 12, то фрагмент дополняется пробелами до 12 символов и т. д. Все это справедливо при указании в качестве параметра числа 4;

- `center(<Ширина>[, <Символ>])` — производит выравнивание строки по центру внутри поля указанной ширины. Если второй параметр не указан, то справа и слева от исходной строки будут добавлены пробелы. Пример:

```
>>> s = "str"
>>> s.center(15), s.center(11, "-")
('      str      ', '----str----')
```

Теперь произведем выравнивание трех фрагментов шириной 15 символов. Первый фрагмент будет выровнен по правому краю, второй — по левому, а третий — по центру:

```
>>> s = "str"
>>> "'%15s' '%-15s' '%s'" % (s, s, s.center(15))
"'            str' 'str            ' '      str      '"
```

Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью:

```
>>> s = "string"
>>> s.center(6), s.center(5)
('string', 'string')
```

- `ljust(<Ширина>[, <Символ>])` — производит выравнивание строки по левому краю внутри поля указанной ширины. Если второй параметр не указан, то справа от исходной строки будут добавлены пробелы. Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Пример:

```
>>> s = "string"
>>> s.ljust(15), s.ljust(15, "-")
('string         ', 'string---------')
>>> s.ljust(6), s.ljust(5)
('string', 'string')
```

- `rjust(<Ширина>[, <Символ>])` — производит выравнивание строки по правому краю внутри поля указанной ширины. Если второй параметр не указан, то слева от исходной строки будут добавлены пробелы. Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Пример:

```
>>> s = "string"
>>> s.rjust(15), s.rjust(15, "-")
('         string', '---------string')
>>> s.rjust(6), s.rjust(5)
('string', 'string')
>>> print unicode("строка", "cp1251").rjust(20, "-")
--------------строка
```

- `zfill(<Ширина>)` — производит выравнивание фрагмента по правому краю внутри поля указанной ширины. Слева от фрагмента будут добавлены нули. Если количество символов в строке превышает ширину поля, то значение ширины игнорируется и строка возвращается полностью. Пример:

```
>>> "5".zfill(20), "123546".zfill(5)
('00000000000000000005', '123546')
```

6.5. Метод *format()*

Начиная с Python 2.6, помимо операции форматирования, строки поддерживают метод `format()`. Метод имеет следующий синтаксис:

```
<Строка специального формата>.format(*args, **kwargs)
```

В параметре `<Строка специального формата>` внутри символов `{` и `}` указываются спецификаторы, имеющие следующий синтаксис:

```
{<Поле>[!<Функция>][:<Формат>]}
```

Все символы, расположенные вне фигурных скобок, выводятся без преобразований. Если внутри строки необходимо использовать символы `{` и `}`, то эти символы следует удвоить, иначе возбуждается исключение `ValueError`. Пример:

```
>>> print "Символы {{ и }} - {0}".format("специальные")
Символы { и } - специальные
```

В параметре `<Поле>` можно указать индекс позиции (нумерация начинается с нуля) или ключ. Допустимо комбинировать позиционные и именованные параметры. В этом случае в методе `format()` именованные параметры указываются в самом конце. Пример:

```
>>> "{0} - {1} - {2}".format(10, 12.3, "string")      # Индексы
'10 - 12.3 - string'
>>> arr = [10, 12.3, "string"]
>>> "{0} - {1} - {2}".format(*arr)                    # Индексы
'10 - 12.3 - string'
>>> "{color} - {model}".format(color="red", model="BMW") # Ключи
'red - BMW'
>>> d = {"color": "red", "model": "BMW"}
>>> "{color} - {model}".format(**d)                   # Ключи
'red - BMW'
>>> "{color} - {0}".format(2010, color="red")         # Комбинация
'red - 2010'
```

В качестве параметра в методе `format()` можно указать объект. Для доступа к элементам по индексу внутри строки формата применяются квадратные скобки, а для доступа к атрибутам объекта используется точечная нотация:

```
>>> arr = [10, [12.3, "string"] ]
>>> "{0[0]} - {0[1][0]} - {0[1][1]}".format(arr)      # Индексы
'10 - 12.3 - string'
```

```
>>> "{arr[0]} - {arr[1][1]}".format(arr=arr)          # Индексы
'10 - string'
>>> class Car: color, model = "red", "BMW"

>>> car = Car()
>>> "{0.color} - {0.model}".format(car)               # Атрибуты
'red - BMW'
```

Параметр <Функция> задает функцию, с помощью которой обрабатываются данные перед вставкой в строку. Если указано значение "s", то данные обрабатываются функцией str(), а если значение "r", то функцией repr(). Если параметр не указан, то для преобразования данных в строку используется функция str(). Пример:

```
>>> print "{0!s}".format("строка")                    # str()
строка
>>> print "{0!r}".format("строка")                    # repr()
'\xf1\xf2\xf0\xee\xea\xe0'
```

В параметре <Формат> указывается значение, имеющее следующий синтаксис:

```
[<Выравнивание>][<Знак>][#][0][<Ширина>][.<Точность>][<Преобразование>]
```

Параметр <Ширина> задает минимальную ширину поля. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

```
>>> "'{0:10}' '{1:3}'".format(3, "string")
"'         3' 'string'"
```

Ширину поля можно передать в качестве параметра в методе format(). В этом случае вместо числа указывается индекс параметра внутри фигурных скобок:

```
>>> "'{0:{1}}'".format(3, 10) # 10 - это ширина поля
"'         3'"
```

По умолчанию значение внутри поля выравнивается по правому краю. Управлять выравниванием позволяет параметр <Выравнивание>. Можно указать следующие значения:

- < — по левому краю;
- > — по правому краю;
- ^ — по центру поля. Пример:

```
>>> "'{0:<10}' '{1:>10}' '{2:^10}'".format(3, 3, 3)
"'3         ' '         3' '    3     '"
```

- = — знак числа выравнивается по левому краю, а число по правому краю:

```
>>> "'{0:=10}' '{1:=10}'".format(-3, 3)
"'-        3' '         3'"
```

Как видно из предыдущего примера, пространство между знаком и числом по умолчанию заполняется пробелами, а знак положительного числа не указывается.

Чтобы вместо пробелов пространство заполнялось нулями, необходимо указать ноль перед шириной поля:

```
>>> "'{0:=010}' '{1:=010}'".format(-3, 3)
"'-000000003' '0000000003'"
```

Управлять выводом знака числа позволяет параметр `<Знак>`. Допустимые значения:

- `+` — задает обязательный вывод знака как для отрицательных, так и для положительных чисел;
- `-` — вывод знака только для отрицательных чисел (значение по умолчанию);
- `пробел` — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Пример:

```
>>> "'{0:+}' '{1:+}' '{0:-}' '{1:-}'".format(3, -3)
"'+3' '-3' '3' '-3'"
>>> "'{0: }' '{1: }'".format(3, -3)        # Пробел
"' 3' '-3'"
```

Для целых чисел в параметре `<Преобразование>` могут быть указаны следующие опции:

- `b` — двоичное значение:

```
>>> "'{0:b}' '{0:#b}'".format(3)
"'11' '0b11'"
```

- `c` — преобразует целое число в соответствующий символ:

```
>>> "'{0:c}'".format(100)
"'d'"
```

- `d` — десятичное значение;
- `n` — аналогично опции `d`, но учитывает настройки локали;
- `o` — восьмеричное значение:

```
>>> "'{0:d}' '{0:o}' '{0:#o}'".format(511)
"'511' '777' '0o777'"
```

- `x` — шестнадцатеричное значение в нижнем регистре:

```
>>> "'{0:x}' '{0:#x}'".format(255)
"'ff' '0xff'"
```

- `X` — шестнадцатеричное значение в верхнем регистре:

```
>>> "'{0:X}' '{0:#X}'".format(255)
"'FF' '0XFF'"
```

Для вещественных чисел в параметре `<Преобразование>` могут быть указаны следующие опции:

- `f` и `F` — вещественное число в десятичном представлении:

```
>>> "'{0:f}' '{1:f}' '{2:f}'".format(30, 18.6578145, -2.5)
"'30.000000' '18.657815' '-2.500000'"
```

Задать количество знаков после запятой позволяет параметр `<Точность>`:

```
>>> "'{0:.7f}' '{1:.2f}'".format(18.6578145, -2.5)
"'18.6578145' '-2.50'"
```

- `e` — вещественное число в экспоненциальной форме (буква "e" в нижнем регистре):
```
>>> "'{0:e}' '{1:e}'".format(3000, 18657.81452)
"'3.000000e+03' '1.865781e+04'"
```
- `E` — вещественное число в экспоненциальной форме (буква "e" в верхнем регистре):
```
>>> "'{0:E}' '{1:E}'".format(3000, 18657.81452)
"'3.000000E+03' '1.865781E+04'"
```
- `g` — эквивалентно `e`, если экспонента меньше `-4`. В противном случае используется десятичное представление вещественного числа:
```
>>> "'{0:g}' '{1:g}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578e-05'"
```
- `n` — аналогично опции `g`, но учитывает настройки локали;
- `G` — эквивалентно `E`, если экспонента меньше `-4`. В противном случае используется десятичное представление вещественного числа:
```
>>> "'{0:G}' '{1:G}'".format(0.086578, 0.000086578)
"'0.086578' '8.6578E-05'"
```
- `%` — умножает число на 100 и добавляет символ процента в конец. Значение отображается в соответствии с опцией `f`. Пример:
```
>>> "'{0:%}' '{1:.4%}'".format(0.086578, 0.000086578)
"'8.657800%' '0.0087%'"
```

6.6. Функции и методы для работы со строками

Рассмотрим основные функции для работы со строками:

- `str([<Объект>])` — преобразует любой объект в строку. Если параметр не указан, то возвращается пустая строка. Используется оператором `print` для вывода объектов. Пример:
```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
```
- `repr(<Объект>)` — возвращает строковое представление объекта. Используется при выводе объектов в окне **Python Shell** редактора IDLE. Пример:
```
>>> repr("Строка"), repr([1, 2, 3]), repr({"x": 5})
("'\\xd1\\xf2\\xf0\\xee\\xea\\xe0'", '[1, 2, 3]', "{'x': 5}")
>>> repr(unicode("Строка", "cp1251"))
"u'\\u0421\\u0442\\u0440\\u043e\\u043a\\u0430'"
```
- `len(<Строка>)` — для строк в однобайтовых кодировках и Unicode-строк возвращает количество символов, а для строк в многобайтовых кодировках — количество байтов:
```
>>> len("Python"), len("\r\n\t"), len(r"\r\n\t")
(6, 3, 6)
```

```
>>> s = "Строка"
>>> len(s) # Строка в кодировке windows-1251
6
>>> # Преобразуем кодировку из windows-1251 в utf-8
>>> s = s.decode("cp1251").encode("utf-8")
>>> len(s) # Строка в кодировке utf-8
12
>>> len(unicode("Строка", "cp1251"))
6
```

- `cmp(<Объект1>, <Объект2>)` — сравнивает два объекта и возвращает следующие значения:
  - `1` — если `<Объект1>` больше `<Объект2>`;
  - `-1` — если `<Объект1>` меньше `<Объект2>`;
  - `0` — если значения равны.

  Пример сравнения строк:

```
>>> s1, s2 = "строка", "строки"
>>> cmp(s1, s2), cmp(s2, s1), cmp(s1, "строка")
(-1, 1, 0)
```

Перечислим основные методы:

- `strip([<Символы>])` — удаляет пробельные символы в начале и конце строки. Пробельными символами считаются: пробел, символ перевода строки (`\n`), символ возврата каретки (`\r`), символы горизонтальной (`\t`) и вертикальной (`\v`) табуляции:

```
>>> s1, s2 = "     str\n\r\v\t", "strstrstrokstrstrstr"
>>> "'%s' - '%s'" % (s1.strip(), s2.strip("tsr"))
"'str' - 'ok'"
>>> print unicode("\tСтрока\r\n ", "cp1251").strip()
Строка
```

- `lstrip([<Символы>])` — удаляет пробельные символы в начале строки:

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' - '%s'" % (s1.lstrip(), s2.lstrip("tsr"))
"'str     ' - 'okstrstrstr'"
>>> "'%s'" % unicode("\tСтрока\r\n ", "cp1251").lstrip()
u"'\u0421\u0442\u0440\u043e\u043a\u0430\r\n '"
```

- `rstrip([<Символы>])` — удаляет пробельные символы в конце строки:

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' - '%s'" % (s1.rstrip(), s2.rstrip("tsr"))
"'     str' - 'strstrstrok'"
>>> "'%s'" % unicode("\tСтрока\r\n ", "cp1251").rstrip()
u"'\t\u0421\u0442\u0440\u043e\u043a\u0430'"
```

- `split([<Разделитель>[, <Лимит>]])` — разделяет строку на подстроки по указанному разделителю и добавляет их в список. Если первый параметр не указан

или имеет значение `None`, то в качестве разделителя используется символ пробела. Если во втором параметре задано число, то в списке будет указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент, в котором будет остаток строки. Примеры:

```
>>> s = "word1 word2 word3"
>>> s.split(), s.split(None, 1)
(['word1', 'word2', 'word3'], ['word1', 'word2 word3'])
>>> s = "word1\nword2\nword3"
>>> s.split("\n")
['word1', 'word2', 'word3']
```

Если в строке содержатся несколько пробелов подряд и разделитель не указан, то пустые элементы не будут добавлены в список:

```
>>> s = "word1           word2 word3     "
>>> s.split()
['word1', 'word2', 'word3']
```

При использовании другого разделителя могут быть пустые элементы:

```
>>> s = ",,word1,,word2,,word3,,"
>>> s.split(",")
['', '', 'word1', '', 'word2', '', 'word3', '', '']
>>> "1,,2,,3".split(",")
['1', '', '2', '', '3']
```

Если разделитель не найден в строке, то список будет состоять из одного элемента, представляющего исходную строку:

```
>>> "word1 word2 word3".split("\n")
['word1 word2 word3']
```

- `rsplit([<Разделитель>[, <Лимит>]])` — метод аналогичен методу `split()`, но поиск символа-разделителя производится не слева направо, а, наоборот, справа налево. Примеры:

```
>>> s = "word1 word2 word3"
>>> s.rsplit(), s.rsplit(None, 1)
(['word1', 'word2', 'word3'], ['word1 word2', 'word3'])
>>> "word1\nword2\nword3".rsplit("\n")
['word1', 'word2', 'word3']
```

- `splitlines([True])` — разделяет строку на подстроки по символу переноса строки (`"\n"`) и добавляет их в список. Символы новой строки включаются в результат, только если необязательный параметр имеет значение `True`. Если разделитель не найден в строке, то список будет содержать только один элемент. Примеры:

```
>>> "word1\nword2\nword3".splitlines()
['word1', 'word2', 'word3']
>>> "word1\nword2\nword3".splitlines(True)
['word1\n', 'word2\n', 'word3']
```

```
>>> "word1\nword2\nword3".splitlines(False)
['word1', 'word2', 'word3']
>>> "word1 word2 word3".splitlines()
['word1 word2 word3']
```

- `partition(<Разделитель>)` — находит первое вхождение символа-разделителя в строку и возвращает кортеж из трех элементов. Первый элемент будет содержать фрагмент, расположенный перед разделителем, второй элемент — символ-разделитель, а третий элемент — фрагмент, расположенный после символа-разделителя. Поиск производится слева направо. Если символ-разделитель не найден, то первый элемент кортежа будет содержать всю строку, а остальные элементы будут пустыми. Пример:

```
>>> "word1 word2 word3".partition(" ")
('word1', ' ', 'word2 word3')
>>> "word1 word2 word3".partition("\n")
('word1 word2 word3', '', '')
```

- `rpartition(<Разделитель>)` — метод аналогичен методу `partition()`, но поиск символа-разделителя производится не слева направо, а, наоборот, справа налево. Если символ-разделитель не найден, то первые два элемента кортежа будут пустыми, а третий элемент будет содержать всю строку. Пример:

```
>>> "word1 word2 word3".rpartition(" ")
('word1 word2', ' ', 'word3')
>>> "word1 word2 word3".rpartition("\n")
('', '', 'word1 word2 word3')
```

- `join()` — преобразует последовательность в строку. Элементы добавляются через указанный разделитель. Формат метода:

```
<Строка> = <Разделитель>.join(<Последовательность>)
```

  В качестве примера преобразуем список и кортеж в строку:

```
>>> " => ".join(["word1", "word2", "word3"])
'word1 => word2 => word3'
>>> " ".join(("word1", "word2", "word3"))
'word1 word2 word3'
```

  Обратите внимание на то, что элементы последовательностей должны быть строками, иначе возбуждается исключение `TypeError`:

```
>>> " ".join(("word1", "word2", 5))
Traceback (most recent call last):
  File "<pyshell#106>", line 1, in <module>
    " ".join(("word1", "word2", 5))
TypeError: sequence item 2: expected string, int found
```

  Как вы уже знаете, строки относятся к неизменяемым типам данных. Если попытаться изменить символ по индексу, то возникнет ошибка. Чтобы изменить символ по индексу, можно преобразовать строку в список с помощью функции

list(), произвести изменения, а затем с помощью метода join() преобразовать список обратно в строку. Пример:

```
>>> s = "Python"
>>> arr = list(s); arr        # Преобразуем строку в список
['P', 'y', 't', 'h', 'o', 'n']
>>> arr[0] = "J"; arr         # Изменяем элемент по индексу
['J', 'y', 't', 'h', 'o', 'n']
>>> s = "".join(arr); s       # Преобразуем список в строку
'Jython'
```

6.7. Настройка локали и изменение регистра символов

При изменении регистра русских букв может возникнуть проблема. Чтобы ее избежать, необходимо правильно настроить локаль. *Локалью* называют совокупность локальных настроек системы.

Для установки локали используется функция setlocale() из модуля locale. Прежде чем использовать функцию, необходимо подключить модуль с помощью выражения:

```
import locale
```

Функция setlocale() имеет следующий формат:

```
setlocale(<Категория>[, <Локаль>]);
```

Параметр <Категория> может принимать следующие значения:

- locale.LC_ALL — устанавливает локаль для всех режимов;
- locale.LC_COLLATE — для сравнения строк;
- locale.LC_CTYPE — для перевода символов в нижний или верхний регистр;
- locale.LC_MONETARY — для отображения денежных единиц;
- locale.LC_NUMERIC — для форматирования дробных чисел;
- locale.LC_TIME — для форматирования вывода даты и времени.

Получить текущее значение локали позволяет функция getlocale([<Категория>]). В качестве примера настроим локаль под Windows вначале на кодировку Windows-1251, потом на кодировку UTF-8, а затем на кодировку по умолчанию. Далее выведем текущее значение локали для всех категорий и только для locale.LC_COLLATE (листинг 6.3).

Листинг 6.3. Настройка локали

```
>>> import locale
>>> # Для кодировки windows-1251
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> # Устанавливаем локаль по умолчанию
>>> locale.setlocale(locale.LC_ALL, "")
```

```
'Russian_Russia.1251'
>>> # Получаем текущее значение локали для всех категорий
>>> locale.getlocale()
('Russian_Russia', '1251')
>>> # Получаем текущее значение категории locale.LC_COLLATE
>>> locale.getlocale(locale.LC_COLLATE)
('Russian_Russia', '1251')
```

Для изменения регистра символов предназначены следующие методы:

- `upper()` — заменяет все символы строки соответствующими прописными буквами:

```
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> s = "строка"
>>> print s.upper()
СТРОКА
>>> print unicode("строка", "cp1251").upper()
СТРОКА
```

- `lower()` — заменяет все символы строки соответствующими строчными буквами:

```
>>> s = "СТРОКА"
>>> print s.lower()
строка
>>> print unicode("СТРОКА", "cp1251").lower()
строка
```

- `swapcase()` — заменяет все строчные символы соответствующими прописными буквами, а все прописные символы — строчными:

```
>>> s = "СТРОКА строка"
>>> print s.swapcase()
строка СТРОКА
>>> print unicode("СТРОКА строка", "cp1251").swapcase()
строка СТРОКА
```

- `capitalize()` — делает первую букву прописной:

```
>>> s = "строка"
>>> print s.capitalize()
Строка
>>> print unicode("строка", "cp1251").capitalize()
Строка
```

- `title()` — делает первую букву каждого слова прописной:

```
>>> s = "первая буква каждого слова станет прописной"
>>> print s.title()
Первая Буква Каждого Слова Станет Прописной
>>> print unicode("строка строка", "cp1251").title()
Строка Строка
```

6.8. Функции для работы с символами

Для работы с отдельными символами предназначены следующие функции:

- `chr(<ASCII код символа>)` — возвращает символ по указанному ASCII-коду:

```
>>> print chr(207)
П
```

- `ord(<Символ>)` — возвращает ASCII- или Unicode-код указанного символа:

```
>>> print ord("П")
207
>>> ord(unicode("П", "cp1251"))
1055
```

- `unichr(<Unicode код символа>)` — возвращает символ по указанному Unicode-коду:

```
>>> print unichr(1055)
П
>>> unichr(1055)
u'\u041f'
```

6.9. Поиск и замена в строке

Для поиска и замены в строке используются следующие методы:

- `find()` — ищет подстроку в строке. Возвращает номер позиции, с которой начинается вхождение подстроки в строку. Если подстрока в строку не входит, то возвращается значение `-1`. Метод зависит от регистра символов. Имеет следующий формат:

```
<Строка>.find(<Подстрока>[, <Начало>[, <Конец>]])
```

Если начальная позиция не указана, то поиск будет производиться с начала строки. Если параметры `<Начало>` и `<Конец>` указаны, то производится операция извлечения среза

```
<Строка>[<Начало>:<Конец>]
```

и поиск подстроки будет производиться в этом фрагменте. Примеры:

```
>>> s = "пример пример Пример"
>>> s.find("при"), s.find("При"), s.find("тест")
(0, 14, -1)
>>> s.find("при", 9), s.find("при", 0, 6), s.find("при", 7, 12)
(-1, 0, 7)
>>> u = unicode("это пример", "cp1251")
>>> u.find(unicode("при", "cp1251"))
4
```

- `index()` — метод аналогичен методу `find()`, но если подстрока в строку не входит, то возбуждается исключение `ValueError`.

Метод имеет следующий формат:

```
<Строка>.index(<Подстрока>[, <Начало>[, <Конец>]])
```

Примеры:

```
>>> s = "пример пример Пример"
>>> s.index("при"), s.index("при", 7, 12), s.index("При", 1)
(0, 7, 14)
>>> s.index("тест")
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    s.index("тест")
ValueError: substring not found
>>> u = unicode("это пример", "cp1251")
>>> u.index(unicode("при", "cp1251"))
4
```

- `rfind()` — ищет подстроку в строке. Возвращает позицию последнего вхождения подстроки в строку. Если подстрока в строку не входит, то возвращается значение `-1`. Метод зависит от регистра символов. Имеет следующий формат:

  ```
  <Строка>.rfind(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Если начальная позиция не указана, то поиск будет производиться с начала строки. Если параметры `<Начало>` и `<Конец>` указаны, то выполняется операция извлечения среза и поиск подстроки будет производиться в этом фрагменте. Примеры:

  ```
  >>> s = "пример пример Пример Пример"
  >>> s.rfind("при"), s.rfind("При"), s.rfind("тест")
  (7, 21, -1)
  >>> s.find("при", 0, 6), s.find("При", 10, 20)
  (0, 14)
  ```

- `rindex()` — метод аналогичен методу `rfind()`, но если подстрока в строку не входит, то возбуждается исключение `ValueError`. Метод имеет следующий формат:

  ```
  <Строка>.rindex(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Примеры:

  ```
  >>> s = "пример пример Пример Пример"
  >>> s.rindex("при"), s.rindex("При"), s.rindex("при", 0, 6)
  (7, 21, 0)
  >>> s.rindex("тест")
  Traceback (most recent call last):
    File "<pyshell#80>", line 1, in <module>
      s.rindex("тест")
  ValueError: substring not found
  ```

- `count()` — возвращает число вхождений подстроки в строку. Если подстрока в строку не входит, то возвращается значение `0`. Метод зависит от регистра символов.

Имеет следующий формат:

```
<Строка>.count(<Подстрока>[, <Начало>[, <Конец>]])
```

Примеры:

```
>>> s = "пример пример Пример Пример"
>>> s.count("при"), s.count("при", 6), s.count("При")
(2, 1, 2)
>>> s.count("тест")
0
```

- `startswith()` — проверяет, начинается ли строка с указанной подстроки. Если начинается, то возвращается значение `True` , в противном случае — `False`. Метод зависит от регистра символов. Имеет следующий формат:

  ```
  <Строка>.startswith(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Если начальная позиция не указана, сравнение будет производиться с началом строки. Если параметры `<Начало>` и `<Конец>` указаны, то выполняется операция извлечения среза и сравнение будет производиться с началом фрагмента. Примеры:

  ```
  >>> s = "пример пример Пример Пример"
  >>> s.startswith("при"), s.startswith("При")
  (True, False)
  >>> s.startswith("при", 6), s.startswith("При", 14)
  (False, True)
  >>> u = unicode("пример", "cp1251")
  >>> u.startswith(unicode("при", "cp1251"))
  True
  ```

  Начиная с версии 2.5, параметр `<Подстрока>` может быть кортежем:

  ```
  >>> s = "пример пример Пример Пример"
  >>> s.startswith(("при", "При"))
  True
  ```

- `endswith()` — проверяет, заканчивается ли строка указанной подстрокой. Если заканчивается, то возвращается значение `True`, в противном случае — `False`. Метод зависит от регистра символов. Имеет следующий формат:

  ```
  <Строка>.endswith(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Если начальная позиция не указана, то сравнение будет производиться с концом строки. Если параметры `<Начало>` и `<Конец>` указаны, то выполняется операция извлечения среза и сравнение будет производиться с концом фрагмента. Примеры:

  ```
  >>> s = "подстрока ПОДСТРОКА"
  >>> s.endswith("ока"), s.endswith("ОКА")
  (False, True)
  >>> s.endswith("ока", 0, 9)
  True
  ```

Начиная с версии 2.5, параметр <Подстрока> может быть кортежем:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith(("ока", "ОКА"))
True
```

- `replace()` — производит замену всех вхождений подстроки в строку на другую подстроку и возвращает результат в виде новой строки. Метод зависит от регистра символов. Имеет следующий формат:

```
<Строка>.replace(<Подстрока для замены>, <Новая подстрока>[,
                 <Максимальное количество замен>])
```

Пример:

```
>>> s = "Привет, Петя"
>>> print s.replace("Петя", "Вася")
Привет, Вася
>>> print s.replace("петя", "вася") # Зависит от регистра
Привет, Петя
>>> s = "strstrstrstrstr"
>>> s.replace("str", ""), s.replace("str", "", 3)
('', 'strstr')
```

- `str.translate(<Таблица символов>[, <Удаляемые символы>])` — удаляет символы, указанные во втором параметре, а остальные символы заменяются в соответствии с параметром <Таблица символов>. В параметре <Таблица символов> следует указать строку из 256 символов. Создать эту строку позволяет функция `maketrans()` из модуля `string`:

```
>>> import string            # Подключаем модуль
>>> string.maketrans("", "") # Выводим таблицу символов
```

Функция `maketrans()` имеет следующий формат:

```
maketrans(<Строка1>, <Строка2>)
```

В качестве параметров указываются строки одинаковой длины. Функция `maketrans()` производит замену символов в строке <Строка1> на символы, расположенные в той же позиции в строке <Строка2>. В качестве значения возвращает измененную таблицу символов.

В первом примере заменим все цифры в строке на `0`. Во втором примере удалим все цифры, а для букв `"abcdf"` изменим регистр:

```
>>> import string
>>> t = string.maketrans("123456789", "0" * 9)
>>> "987654321".translate(t)
'000000000'
>>> t = string.maketrans("abcdf", "ABCDF")
>>> "12457478abcdfgh".translate(t, "1234567890")
'ABCDFgh'
```

Для Unicode-строк синтаксис метода `translate()` имеет другой формат:

```
unicode.translate(<Таблица символов>)
```

Параметр <Таблица символов> должен быть отображением, ключами которого являются Unicode-коды заменяемых символов, а значениями — Unicode-коды вставляемых символов. Если в качестве значения указать None, то символ будет удален. Для примера удалим букву "П", а также изменим регистр всех букв "р":

```
>>> u = unicode("Пример", "cp1251")
>>> char1 = unicode("П", "cp1251")
>>> char2 = unicode("р", "cp1251")
>>> char3 = unicode("Р", "cp1251")
>>> d = {ord(char1): None, ord(char2): ord(char3)}
>>> d
{1088: 1056, 1055: None}
>>> print u.translate(d)
РимеР
```

6.10. Проверка типа содержимого строки

Для проверки типа содержимого предназначены следующие методы:

- isdigit() — возвращает True, если строка содержит только цифры, в противном случае — False:

```
>>> "0123".isdigit(), "123abc".isdigit(), "abc123".isdigit()
(True, False, False)
>>> unicode("123456", "cp1251").isdigit()
True
```

- isalpha() — возвращает True, если строка содержит только буквы, в противном случае — False. Если строка пустая, то возвращается значение False. Примеры:

```
>>> "string".isalpha(), "строка".isalpha(), "".isalpha()
(True, True, False)
>>> "123abc".isalpha(), "str str".isalpha(), "st,st".isalpha()
(False, False, False)
>>> unicode("строка", "cp1251").isalpha()
True
```

- isspace() — возвращает True, если строка содержит только пробельные символы, в противном случае — False:

```
>>> "".isspace(), " \n\r\t".isspace(), "str str".isspace()
(False, True, False)
>>> unicode(" \n\t\r\v", "cp1251").isspace()
True
```

- isalnum() — возвращает True, если строка содержит только буквы и (или) цифры, в противном случае — False. Если строка пустая, то возвращается значение False.

Примеры:

```
>>> "0123".isalnum(), "123abc".isalnum(), "abc123".isalnum()
(True, True, True)
>>> "строка".isalnum()
True
>>> "".isalnum(), "123 abc".isalnum(), "abc, 123.".isalnum()
(False, False, False)
>>> unicode("123абв", "cp1251").isalnum()
True
```

- `islower()` — возвращает `True`, если строка содержит буквы и они все в нижнем регистре, в противном случае — `False`. Помимо букв, строка может иметь другие символы, например, цифры. Примеры:

```
>>> "srting".islower(), "строка".islower(), "".islower()
(True, True, False)
>>> "srting1".islower(), "str, 123".islower(), "123".islower()
(True, True, False)
>>> "STRING".islower(), "String".islower()
(False, False)
>>> unicode("абвгде", "cp1251").islower()
True
```

- `isupper()` — возвращает `True`, если строка содержит буквы и они все в верхнем регистре, в противном случае — `False`. Помимо букв, строка может иметь другие символы, например цифры. Примеры:

```
>>> "STRING".isupper(), "СТРОКА".isupper(), "".isupper()
(True, True, False)
>>> "STRING1".isupper(), "СТРОКА, 123".isupper(), "123".isupper()
(True, True, False)
>>> "string".isupper(), "STRing".isupper()
(False, False)
>>> unicode("АБВГДЕ", "cp1251").isupper()
True
```

- `istitle()` — возвращает `True`, если строка содержит буквы и первые буквы всех слов являются заглавными, в противном случае — `False`. Помимо букв, строка может иметь другие символы, например цифры. Примеры:

```
>>> "Str Str".istitle(), "Стр Стр".istitle()
(True, True)
>>> "Str Str 123".istitle(), "Стр Стр 123".istitle()
(True, True)
>>> "Str str".istitle(), "Стр стр".istitle()
(False, False)
>>> "".istitle(), "123".istitle()
(False, False)
>>> unicode("Стр Стр Стр", "cp1251").istitle()
True
```

Для Unicode-строк определены два дополнительных метода:

- `unicode.isdecimal()` — возвращает `True`, если строка содержит только десятичные символы, в противном случае — `False`. Обратите внимание на то, что к десятичным символам относятся не только десятичные цифры в кодировке ASCII, но и надстрочные и подстрочные десятичные цифры в других языках. Пример:

```
>>> unicode("123", "cp1251").isdecimal()
True
>>> unicode("123стр", "cp1251").isdecimal()
False
```

- `unicode.isnumeric()` — возвращает `True`, если строка содержит только числовые символы, в противном случае — `False`. Обратите внимание на то, что к числовым символам относятся не только десятичные цифры в кодировке ASCII, но символы римских чисел, дробные числа и др. Пример:

```
>>> u"\u2155".isnumeric(), u"\u2155".isdigit()
(True, False)
>>> print u"\u2155" # Выведет символ "1/5"
```

Переделаем нашу программу (листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 6.4). Кроме того, предусмотрим возможность ввода отрицательных целых чисел.

Листинг 6.4. Суммирование неопределенного количества чисел

```
# -*- coding: cp1251 -*-
print "Введите слово 'stop' для получения результата"
summa = 0
while True:
    x = raw_input("Введите число: ")
    if x == "stop":
        break # Выход из цикла
    if x == "":
        print "Вы не ввели значение!"
        continue
    if x[0] == "-": # Если первым символом является минус
        if not x[1:].isdigit(): # Если фрагмент не состоит из цифр
            print "Необходимо ввести число, а не строку!"
            continue
    else: # Если минуса нет, то проверяем всю строку
        if not x.isdigit(): # Если строка не состоит из цифр
           print "Необходимо ввести число, а не строку!"
           continue
    x = int(x) # Преобразуем строку в число
    summa += x
print "Сумма чисел равна:", summa
```

Процесс ввода значений и получения результата выглядит так:

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число:
Вы не ввели значение!
Введите число: str
Необходимо ввести число, а не строку!
Введите число: -5
Введите число: -str
Необходимо ввести число, а не строку!
Введите число: stop
Сумма чисел равна: 5
```

Значения, введенные пользователем, выделены полужирным шрифтом.

6.11. Преобразование объекта в строку

Преобразовать объект в строку, а затем восстановить объект из строки позволяет модуль `pickle`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью выражения:

```
import pickle
```

Для преобразования предназначены две функции:

- `dumps(<Объект>[, <Протокол>])` — производит сериализацию объекта и возвращает строку специального формата. Формат этой строки зависит от указанного протокола (число от `0` до `2`). Пример преобразования списка и кортежа в строку:

```
>>> import pickle
>>> obj1 = [1, 2, 3, 4, 5]      # Список
>>> obj2 = (6, 7, 8, 9, 10)     # Кортеж
>>> pickle.dumps(obj1)
'(lp0\nI1\naI2\naI3\naI4\naI5\na.'
>>> pickle.dumps(obj2)
'(I6\nI7\nI8\nI9\nI10\ntp0\n.'
```

- `loads(<Строка>)` — преобразует строку специального формата обратно в объект. Пример восстановления списка и кортежа из строки специального формата:

```
>>> pickle.loads('(lp0\nI1\naI2\naI3\naI4\naI5\na.')
[1, 2, 3, 4, 5]
>>> pickle.loads('(I6\nI7\nI8\nI9\nI10\ntp0\n.')
(6, 7, 8, 9, 10)
```

6.12. Шифрование строк

Для шифрования строк предназначен модуль `hashlib`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью выражения:

```
import hashlib
```

Модуль предоставляет следующие функции: `md5()`, `sha1()`, `sha224()`, `sha256()`, `sha384()` и `sha512()`. В качестве необязательного параметра функциям можно передать шифруемую строку. Пример:

```
>>> h = hashlib.sha1("password")
```

Передать строку можно также с помощью метода `update()`. В этом случае строка присоединяется к предыдущему значению:

```
>>> h = hashlib.sha1()
>>> h.update("password")
```

Получить зашифрованную строку позволяют два метода — `digest()` и `hexdigest()`:

```
>>> h = hashlib.sha1("password")
>>> h.digest()
'[\xaaa\xe4\xc9\xb9??\x06\x82%\x0bl\xf83\x1b~\xe6\x8f\xd8'
>>> h.hexdigest()
'5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8'
```

Наиболее часто применяемой функцией является функция `md5()`, которая шифрует строку с помощью алгоритма MD5. Она используется для шифрования паролей, т. к. не существует алгоритма для дешифровки. Для сравнения введенного пользователем пароля с сохраненным в базе необходимо зашифровать введенный пароль, а затем произвести сравнение (листинг 6.5).

Листинг 6.5. Проверка правильности ввода пароля

```
>>> import hashlib
>>> h = hashlib.md5("password")
>>> p = h.hexdigest()
>>> p                                # Пароль, сохраненный в базе
'5f4dcc3b5aa765d61d8327deb882cf99'
>>> h2 = hashlib.md5("password")     # Пароль, введенный пользователем
>>> if p == h2.hexdigest(): print "Пароль правильный"
Пароль правильный
```

6.13. Преобразование кодировок

Для преобразования кодировок предназначены следующие функции и методы:

- `unicode([<Строка>[, <Кодировка>[, <Обработка ошибок>]]])` — преобразует строку в указанной кодировке в Unicode-строку. В третьем параметре могут

быть указаны значения "strict" (значение по умолчанию), `"replace"` или `"ignore"`. Пример:

```
>>> s = "Кодировка windows-1251"
>>> unicode(s, "cp1251")
u'\u041a\u043e\u0434\u0438\u0440\u043e\u0432\u043a\u0430
windows-1251'
```

- `str.decode([<Кодировка строки>[, <Обработка ошибок>]])` — преобразует строку в указанной кодировке в Unicode-строку. Во втором параметре могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`. Пример:

```
>>> s = "Кодировка windows-1251"
>>> s.decode("cp1251")
u'\u041a\u043e\u0434\u0438\u0440\u043e\u0432\u043a\u0430
windows-1251'
>>> s.decode("utf-8", "strict")
... Фрагмент опущен ...
UnicodeDecodeError: 'utf8' codec can't decode bytes in
position 0-1: invalid data
>>> s.decode("utf-8", "replace")
u'\ufffd\ufffd\ufffd\ufffdindows-1251'
>>> s.decode("utf-8", "ignore")
u'indows-1251'
```

- `unicode.encode([<Нужная кодировка>[, <Обработка ошибок>]])` — преобразует Unicode-строку в обычную строку в указанной кодировке. Во втором параметре могут быть указаны значения `"strict"` (значение по умолчанию), `"replace"`, `"ignore"`, `"xmlcharrefreplace"` и `"backslashreplace"`. Пример:

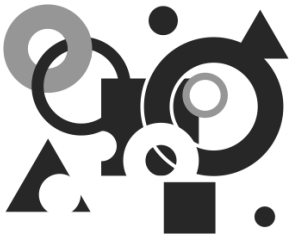
```
>>> u = unicode("Кодировка", "cp1251")
>>> print u.encode("cp1251")
Кодировка
```

Преобразуем строку из кодировки Windows-1251 в кодировку KOI8-R, а затем обратно (листинг 6.6).

Листинг 6.6. Преобразование кодировок

```
>>> w = "Строка"                    # Строка в кодировке windows-1251
>>> k = w.decode("cp1251").encode("koi8-r")
>>> print k                         # Строка в кодировке KOI8-R
уФТПЛБ
>>> w = k.decode("koi8-r").encode("cp1251")
>>> print w
Строка
```

Глава 7



Регулярные выражения

Регулярные выражения позволяют осуществить сложный поиск или замену в строке. В языке Python использовать регулярные выражения позволяет модуль `re`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью выражения:

```
import re
```

7.1. Синтаксис регулярных выражений

Создать откомпилированный шаблон регулярного выражения позволяет функция `compile()`. Функция имеет следующий формат:

```
<Шаблон> = re.compile(<Регулярное выражение>[, <Модификатор>])
```

В параметре `<Модификатор>` могут быть указаны следующие флаги (или их комбинация через символ `|`):

- `I` или `IGNORECASE` — поиск без учета регистра. Для русского языка необходимо дополнительно указать флаг `L` и настроить локаль;
- `L` или `LOCALE` — учитывает настройки текущей локали. Например, сделать регистронезависимый поиск для русских букв можно следующим образом:

```
>>> import re, locale
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> p = re.compile(r"^[а-яё]+$", re.I | re.L)
>>> print "Найдено" if p.search("АБВГДЕЁ") else "Нет"
Найдено
```

- `M` или `MULTILINE` — поиск в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`"\n"`). Символ `^` соответствует привязке к началу каждой подстроки, а символ `$` соответствует позиции перед символом перевода строки;

- `S` или `DOTALL` — метасимвол "точка" будет соответствовать любому символу, включая символ перевода строки (`\n`). По умолчанию метасимвол "точка" не соответствует символу перевода строки. Пример:

```
>>> p = re.compile(r"^.$")
>>> print "Найдено" if p.search("\n") else "Нет"
Нет
>>> p = re.compile(r"^.$", re.S)
>>> print "Найдено" if p.search("\n") else "Нет"
Найдено
```

- `X` или `VERBOSE` — если флаг указан, то пробелы и символы перевода строки будут игнорированы. Кроме того, внутри регулярного выражения можно использовать комментарии. Пример:

```
>>> p = re.compile(r"""^ # Привязка к началу строки
        [0-9]+ # Строка должна содержать одну цифру (или более)
        $      # Привязка к концу строки
        """, re.X)
>>> print "Найдено" if p.search("1234567890") else "Нет"
Найдено
>>> print "Найдено" if p.search("abcd123") else "Нет"
Нет
```

- `U` или `UNICODE` — классы `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` будут соответствовать Unicode-символам. Пример:

```
>>> s = unicode("абв", "cp1251")
>>> p = re.compile(r"^\w+$") # Флаг не указан
>>> print "Найдено" if p.search(s) else "Нет"
Нет
>>> p = re.compile(r"^\w+$", re.U) # Флаг указан
>>> print "Найдено" if p.search(s) else "Нет"
Найдено
```

Как видно из примеров, перед всеми строками, содержащими регулярные выражения, указан модификатор `r`. Иными словами, мы используем неформатированные строки. Если модификатор не указать, то все слеши необходимо будет экранировать. Например, строку

```
p = re.compile(r"^\w+$")
```

нужно было бы записать так

```
p = re.compile("^\\w+$")
```

Внутри регулярного выражения символы `.`, `^`, `$`, `*`, `+`, `?`, `{`, `[`, `]`, `\`, `|`, `(` и `)` имеют специальное значение. Если эти символы должны трактоваться как есть, то их следует экранировать с помощью слеша. Некоторые специальные символы теряют свое специальное значение, если их разместить внутри квадратных скобок. В этом случае экранировать их не нужно. Например, метасимвол "точка" соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед точкой надо указать символ `\` или разместить точку внутри квад-

ратных скобок ([.]). Продемонстрируем это на примере проверки правильности введенной даты (листинг 7.1).

Листинг 7.1. Проверка правильности ввода даты

```
# -*- coding: cp1251 -*-
import re          # Подключаем модуль
d = "29,12.2009"  # Вместо точки указана запятая

p = re.compile(r"^[0-3][0-9].[01][0-9].[12][09][0-9][0-9]$")
# Символ "\" не указан перед точкой
if p.search(d):
    print "Дата введена правильно"
else:
    print "Дата введена неправильно"
# Так как точка означает любой символ,
# выведет: Дата введена правильно

p = re.compile(r"^[0-3][0-9]\.[01][0-9]\.[12][09][0-9][0-9]$")
# Символ "\" указан перед точкой
if p.search(d):
    print "Дата введена правильно"
else:
    print "Дата введена неправильно"
# Так как перед точкой указан символ "\",
# выведет: Дата введена неправильно

p = re.compile(r"^[0-3][0-9][.][01][0-9][.][12][09][0-9][0-9]$")
# Точка внутри квадратных скобок
if p.search(d):
    print "Дата введена правильно"
else:
    print "Дата введена неправильно"
# Выведет: Дата введена неправильно
```

В этом примере мы осуществляли привязку к началу и концу строки с помощью следующих метасимволов:

- ^ — привязка к началу строки (назначение зависит от модификатора);
- $ — привязка к концу строки (назначение зависит от модификатора);
- \A — привязка к началу строки (не зависит от модификатора);
- \Z — привязка к концу строки (не зависит от модификатора).

Если в параметре <Модификатор> указан флаг M (или MULTILINE), то поиск производится в строке, состоящей из нескольких подстрок, разделенных символом новой строки ("\n"). В этом случае символ ^ соответствует привязке к началу каждой подстроки, а символ $ соответствует позиции перед символом перевода строки (листинг 7.2).

Листинг 7.2. Пример использования многострочного режима

```
>>> p = re.compile(r"^.+$")       # Точка не соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Ничего не найдено
[]
>>> p = re.compile(r"^.+$", re.S) # Теперь точка соответствует \n
>>> p.findall("str1\nstr2\nstr3") # Строка полностью соответствует
['str1\nstr2\nstr3']
>>> p = re.compile(r"^.+$", re.M) # Многострочный режим
>>> p.findall("str1\nstr2\nstr3") # Получили каждую подстроку
['str1', 'str2', 'str3']
```

Привязку к началу и концу строки следует использовать, если строка должна полностью соответствовать регулярному выражению. Например, привязку нужно использовать для проверки, содержит ли строка число (листинг 7.3).

Листинг 7.3. Проверка наличия целого числа в строке

```
# -*- coding: cp1251 -*-
import re                       # Подключаем модуль
p = re.compile(r"^[0-9]+$")
if p.search("245"):
    print "Число"              # Выведет: Число
else:
    print "Не число"
if p.search("Строка245"):
    print "Число"
else:
    print "Не число"           # Выведет: Не число
```

Если убрать привязку к началу и концу строки, то любая строка, содержащая хотя бы одну цифру, будет распознана как `"Число"` (листинг 7.4).

Листинг 7.4. Отсутствие привязки к началу или концу строки

```
# -*- coding: cp1251 -*-
import re                       # Подключаем модуль
p = re.compile(r"[0-9]+")
if p.search("Строка245"):
    print "Число"              # Выведет: Число
else:
    print "Не число"
```

Кроме того, можно указать привязку только к началу или только к концу строки (листинг 7.5).

Листинг 7.5. Привязка к началу и концу строки

```
# -*- coding: cp1251 -*-
import re                       # Подключаем модуль
p = re.compile(r"[0-9]+$")
if p.search("Строка245"):
    print "Есть число в конце строки"
else:
    print "Нет числа в конце строки"
# Выведет: Есть число в конце строки
p = re.compile(r"^[0-9]+")
if p.search("Строка245"):
    print "Есть число в начале строки"
else:
    print "Нет числа в начале строки"
# Выведет: Нет числа в начале строки
```

В квадратных скобках `[]` можно указать символы, которые могут встречаться на этом месте в строке. Можно перечислять символы подряд или указать диапазон через тире:

- `[09]` — соответствует числу 0 или 9;
- `[0-9]` — соответствует любому числу от 0 до 9;
- `[абв]` — соответствует буквам "а", "б" и "в";
- `[а-г]` — соответствует буквам "а", "б", "в" и "г";
- `[а-яё]` — соответствует любой букве от "а" до "я";
- `[АБВ]` — соответствует буквам "А", "Б" и "В";
- `[А-ЯЁ]` — соответствует любой букве от "А" до "Я";
- `[а-яА-ЯёЁ]` — соответствует любой русской букве в любом регистре;
- `[0-9а-яА-ЯёЁa-zA-Z]` — любая цифра и любая буква независимо от регистра и языка.

ВНИМАНИЕ!

Буква "ё" не входит в диапазон `[а-я]`.

Значение можно инвертировать, если после первой скобки указать символ `^`. Таким образом можно указать символы, которых не должно быть на этом месте в строке:

- `[^09]` — не цифра 0 или 9;
- `[^0-9]` — не цифра от 0 до 9;
- `[^а-яА-ЯёЁa-zA-Z]` — не буква.

Как вы уже знаете, точка теряет свое специальное значение, если ее заключить в квадратные скобки. Кроме того, внутри квадратных скобок могут встретиться символы, которые имеют специальное значение (например, `^` и `-`). Символ `^` теряет свое специальное значение, если он не расположен сразу после открывающей квадратной скобки. Чтобы отменить специальное значение символа `-`, его необхо-

димо указать после перечисления всех символов перед закрывающей квадратной скобкой. Все специальные символы можно сделать обычными, если перед ними указать символ \.

Вместо перечисления символов можно использовать стандартные классы:

- \d — соответствует любой цифре. Эквивалентно [0-9];
- \w — соответствует любой букве, цифре или символу подчеркивания. Зависит от модификаторов L (LOCALE) и U (UNICODE). По умолчанию эквивалентно [a-zA-Z0-9_]. Настроим класс на работу с русскими буквами:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> p = re.compile(r"^\w+$", re.L)
>>> print "Найдено" if p.search("абвгдеё") else "Нет"
Найдено
```

- \s — любой пробельный символ. Эквивалентно [\t\n\r\f\v];
- \D — не цифра. Эквивалентно [^0-9];
- \W — не буква, не цифра и не символ подчеркивания. Зависит от модификаторов L (или LOCALE) и U (или UNICODE). По умолчанию эквивалентно [^a-zA-Z0-9_];
- \S — не пробельный символ. Эквивалентно [^ \t\n\r\f\v].

Количество вхождений символа в строку задается с помощью *квантификаторов*:

- {n} — n вхождений символа в строку. Например, шаблон r"^[0-9]{2}$" соответствует двум вхождениям любой цифры;
- {n,} — n или более вхождений символа в строку. Например, шаблон r"^[0-9]{2,}$" соответствует двум и более вхождениям любой цифры;
- {n,m} — не менее n и не более m вхождений символа в строку. Числа указываются через запятую без пробела. Например, шаблон r"^[0-9]{2,4}$" соответствует от двух до четырех вхождениям любой цифры;
- * — ноль или большее число вхождений символа в строку. Эквивалентно комбинации {0,};
- + — одно или большее число вхождений символа в строку. Эквивалентно комбинации {1,};
- ? — ни одного или одно вхождение символа в строку. Эквивалентно комбинации {0,1}.

Все квантификаторы являются "жадными". При поиске соответствия ищется самая длинная подстрока, соответствующая шаблону, и не учитываются более короткие соответствия. Рассмотрим это на примере. Получим содержимое всех тегов <b> вместе с тегами:

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*</b>", re.S)
>>> p.findall(s)
['<b>Text1</b>Text2<b>Text3</b>']
```

Вместо желаемого результата мы получили полностью строку. Чтобы ограничить "жадность", необходимо после квантификатора указать символ ? (листинг 7.6).

Листинг 7.6. Ограничение "жадности" квантификаторов

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>.*?</b>", re.S)
>>> p.findall(s)
['<b>Text1</b>', '<b>Text3</b>']
```

Этот код вывел то, что мы искали. Если необходимо получить содержимое без тегов, то нужный фрагмент внутри шаблона следует разместить внутри круглых скобок (листинг 7.7).

Листинг 7.7. Получение значения определенного фрагмента

```
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> p = re.compile(r"<b>(.*?)</b>", re.S)
>>> p.findall(s)
['Text1', 'Text3']
```

Круглые скобки часто используются для группировки фрагментов внутри шаблона. В этом случае не требуется, чтобы фрагмент запоминался и был доступен в результатах поиска. Чтобы избежать захвата фрагмента после открывающей круглой скобки следует разместить символы `?:` (листинг 7.8).

Листинг 7.8. Ограничение захвата фрагмента

```
>>> s = "test text"
>>> p = re.compile(r"([a-z]+((st)|(xt)))", re.S)
>>> p.findall(s)
[('test', 'st', 'st', ''), ('text', 'xt', '', 'xt')]
>>> p = re.compile(r"([a-z]+(?:(?:st)|(?:xt)))", re.S)
>>> p.findall(s)
['test', 'text']
```

В первом примере мы получили список с двумя элементами. Каждый элемент списка является кортежем, содержащим четыре элемента. Все эти элементы соответствуют фрагментам, заключенным в шаблоне в круглые скобки. Первый элемент кортежа содержит фрагмент, расположенный в первых круглых скобках, второй — во вторых круглых скобках и т. д. Три последних элемента кортежа являются лишними. Чтобы они не выводились в результатах, мы добавили символы `?:` после каждой открывающей круглой скобкой. В результате список состоит только из фрагментов, полностью соответствующих регулярному выражению.

Обратите внимание на регулярное выражение в предыдущем примере:

```
r"([a-z]+((st)|(xt)))"
```

Здесь мы использовали метасимвол `|`, который позволяет сделать выбор между альтернативными значениями. Выражение `n|m` соответствует одному из символов: `n` или `m`. Пример:

`красн((ая)|(ое))` — красная или красное, но не красный.

К найденному фрагменту в круглых скобках внутри шаблона можно обратиться с помощью механизма *обратных ссылок*. Для этого порядковый номер круглых скобок в шаблоне указывается после слеша, например `\1`. Нумерация скобок внутри шаблона начинается с `1`. Для примера получим текст между одинаковыми парными тегами (листинг 7.9).

Листинг 7.9. Обратные ссылки

```
>>> s = "<b>Text1</b>Text2<I>Text3</I>"
>>> p = re.compile(r"<([a-z]+)>(.*?)</\1>", re.S | re.I)
>>> p.findall(s)
[('b', 'Text1'), ('I', 'Text3')]
```

Фрагментам внутри круглых скобок можно дать имена. Для этого после открывающей круглой скобки следует указать комбинацию символов `?P<name>`. В качестве примера разберем e-mail на составные части (листинг 7.10).

Листинг 7.10. Именованные фрагменты

```
>>> email = "unicross@mail.ru"
>>> p = re.compile(r"""(?P<name>[a-z0-9_.-]+) # Название ящика
        @                                    # Символ "@"
        (?P<host>(?:[a-z0-9-]+\.)+[a-z]{2,6}) # Домен
        """, re.I | re.VERBOSE)
>>> r = p.search(email)
>>> r.group("name")                          # Название ящика
'unicross'
>>> r.group("host")                          # Домен
'mail.ru'
```

Чтобы внутри шаблона обратиться к именованным фрагментам, используется следующий синтаксис: `(?P=name)`. Для примера получим текст между одинаковыми парными тегами (листинг 7.11).

Листинг 7.11. Обращение к именованным фрагментам внутри шаблона

```
>>> s = "<b>Text1</b>Text2<I>Text3</I>"
>>> p = re.compile(r"<(?P<tag>[a-z]+)>(.*?)</(?P=tag)>", re.S | re.I)
>>> p.findall(s)
[('b', 'Text1'), ('I', 'Text3')]
```

Кроме того, внутри круглых скобок могут быть расположены следующие конструкции:

- `(?iLmsux)` — позволяет установить опции регулярного выражения. Буквы "i", "L", "m", "s", "u" и "x" имеют такое же назначение, что и одноименные модификаторы в функции `compile()`;
- `(?#...)` — комментарий. Текст внутри круглых скобок игнорируется;
- `(?=...)` — положительный просмотр вперед. Выведем все слова, после которых расположена запятая:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"\w+(?=[,])", re.S | re.I)
>>> p.findall(s)
['text1', 'text2']
```

- `(?!...)` — отрицательный просмотр вперед. Выведем все слова, после которых нет запятой:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"[a-z]+[0-9](?![,])", re.S | re.I)
>>> p.findall(s)
['text3', 'text4']
```

- `(?<=...)` — положительный просмотр назад. Выведем все слова, перед которыми расположена запятая с пробелом:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"(?<=[,][ ])[a-z]+[0-9]", re.S | re.I)
>>> p.findall(s)
['text2', 'text3']
```

- `(?<!...)` — отрицательный просмотр назад. Выведем все слова, перед которыми расположен пробел, но перед пробелом нет запятой:

```
>>> s = "text1, text2, text3 text4"
>>> p = re.compile(r"(?<![,]) [a-z]+[0-9]", re.S | re.I)
>>> p.findall(s)
[' text4']
```

- `(?(id` или `name)шаблон1|шаблон2)` — если группа с номером или названием найдена, то должно выполняться условие из параметра `шаблон1`, в противном случае должно выполняться условие из параметра `шаблон2`. Выведем все слова, которые расположены внутри апострофов. Если перед словом нет апострофа, то в конце слова должна быть запятая:

```
>>> s = "text1 'text2' 'text3 text4, text5"
>>> p = re.compile(r"(')?([a-z]+[0-9])(?(1)'|,)", re.S | re.I)
>>> p.findall(s)
[("'", 'text2'), ('', 'text4')]
```

Рассмотрим небольшой пример. Предположим, необходимо получить все слова, расположенные после тире, причем перед тире и после слов должны следовать пробельные символы:

```
>>> import re
>>> s = "-word1 -word2 -word3 -word4 -word5"
```

```
>>> re.findall(r"\s\-([a-z0-9]+)\s", s, re.S | re.I)
['word2', 'word4']
```

Как видно из примера, мы получили только два слова вместо пяти. Первое и последнее слово не попали в результат, т. к. расположены в начале и конце строки. Чтобы эти слова попали в результат, необходимо добавить альтернативный выбор `(^|\s)` для начала строки и `(\s|$)` для конца строки. Чтобы найденные выражения внутри круглых скобок не попали в результат, следует добавить символы `?:` после открывающей скобки:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?:\s|$)", s, re.S | re.I)
['word1', 'word3', 'word5']
```

Первое и последнее слово успешно попали в результат. Почему же слова "word2" и "word4" не попали в список совпадений? Ведь перед тире есть пробел и после слова есть пробел. Чтобы понять причину, рассмотрим поиск по шагам. Первое слово успешно попадает в результат, т. к. перед тире расположено начало строки и после слова есть пробел. После поиска указатель перемещается, и строка для дальнейшего поиска примет следующий вид:

```
"-word1 <Указатель>-word2 -word3 -word4 -word5"
```

Обратите внимание на то, что перед фрагментом `"-word2"` больше нет пробела и тире не расположено в начале строки. Поэтому следующим совпадением будет слово "word3", и указатель снова будет перемещен:

```
"-word1 -word2 -word3 <Указатель>-word4 -word5"
```

Опять перед фрагментом `"-word4"` нет пробела и тире не расположено в начале строки. Поэтому следующим совпадением будет слово "word5" и поиск будет завершен. Таким образом, слова "word2" и "word4" не попадают в результат, т. к. пробел до фрагмента уже был использован в предыдущем поиске. Чтобы этого избежать следует воспользоваться положительным просмотром вперед `(?=...)`:

```
>>> re.findall(r"(?:^|\s)\-([a-z0-9]+)(?=\s|$)", s, re.S | re.I)
['word1', 'word2', 'word3', 'word4', 'word5']
```

В этом примере мы заменили фрагмент `(?:\s|$)` на `(?=\s|$)`. Поэтому все слова успешно попали в список совпадений.

7.2. Поиск первого совпадения с шаблоном

Для поиска первого совпадения с шаблоном предназначены следующие функции и методы:

- `match()` — проверяет соответствие с началом строки. Формат метода:
  `match(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])`

Если соответствие найдено, то возвращается объект `MatchObject`, в противном случае возвращается значение `None`. Пример:

```
>>> import re
>>> p = re.compile(r"[0-9]+")
```

```
>>> print "Найдено" if p.match("str123") else "Нет"
Нет
>>> print "Найдено" if p.match("str123", 3) else "Нет"
Найдено
>>> print "Найдено" if p.match("123str") else "Нет"
Найдено
```

Вместо метода `match()` можно воспользоваться функцией `match()`. Формат функции:

```
re.match(<Шаблон>, <Строка>[, <Модификатор>])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Если соответствие найдено, то возвращается объект `MatchObject`, в противном случае возвращается значение `None`. Пример:

```
>>> p = r"[0-9]+"
>>> print "Найдено" if re.match(p, "str123") else "Нет"
Нет
>>> print "Найдено" if re.match(p, "123str") else "Нет"
Найдено
>>> p = re.compile(r"[0-9]+")
>>> print "Найдено" if re.match(p, "123str") else "Нет"
Найдено
```

- `search()` — проверяет соответствие с любой частью строки. Формат метода:

```
search(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Если соответствие найдено, то возвращается объект `MatchObject`, в противном случае возвращается значение `None`. Пример:

```
>>> p = re.compile(r"[0-9]+")
>>> print "Найдено" if p.search("str123") else "Нет"
Найдено
>>> print "Найдено" if p.search("123str") else "Нет"
Найдено
>>> print "Найдено" if p.search("123str", 3) else "Нет"
Нет
```

Вместо метода `search()` можно воспользоваться функцией `search()`. Формат функции:

```
re.search(<Шаблон>, <Строка>[, <Модификатор>])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Если соответствие найдено, то возвращается объект `MatchObject`, в противном случае возвращается значение `None`. Пример:

```
>>> p = r"[0-9]+"
>>> print "Найдено" if re.search(p, "str123") else "Нет"
Найдено
```

```
>>> p = re.compile(r"[0-9]+")
>>> print "Найдено" if re.search(p, "str123") else "Нет"
Найдено
```

В качестве примера переделаем нашу программу (листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 7.12). Кроме того, предусмотрим возможность ввода отрицательных целых чисел.

Листинг 7.12. Суммирование неопределенного количества чисел

```
# -*- coding: cp1251 -*-
import re        # Подключаем модуль
print "Введите слово 'stop' для получения результата"
summa = 0
p = re.compile(r"^[-]?[0-9]+$")
while True:
    x = raw_input("Введите число: ")
    if x == "stop":
        break    # Выход из цикла
    if not p.search(x):
        print "Необходимо ввести число, а не строку!"
        continue # Переходим на следующую итерацию цикла
    x = int(x)   # Преобразуем строку в число
    summa += x
print "Сумма чисел равна:", summa
```

Объект `MatchObject`, возвращаемый методами (функциями) `match()` и `search()`, имеет следующие свойства и методы:

- `re` — ссылка на скомпилированный шаблон, указанный в методах (функциях) `match()` и `search()`. Через эту ссылку доступны следующие свойства:
  - `groups` — количество групп в шаблоне;
  - `groupindex` — словарь с названиями групп и их номерами;
- `string` — значение параметра `<Строка>` в методах (функциях) `match()` и `search()`;
- `pos` — значение параметра `<Начальная позиция>` в методах `match()` и `search()`;
- `endpos` — значение параметра `<Конечная позиция>` в методах `match()` и `search()`;
- `lastindex` — возвращает номер последней группы или значение `None`;
- `lastgroup` — возвращает название последней группы или значение `None`. Пример:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m.re.groups, m.re.groupindex
(2, {'num': 1, 'str': 2})
>>> p.groups, p.groupindex
```

```
(2, {'num': 1, 'str': 2})
>>> m.string
'123456string 67890text'
>>> m.lastindex, m.lastgroup
(2, 'str')
>>> m.pos, m.endpos
(0, 22)
```

- `group([<id1 или name1>[, ..., <idN или nameN>]])` — возвращает фрагменты, соответствующие шаблону. Если параметр не задан или указано значение `0`, то возвращается фрагмент, полностью соответствующий шаблону. Если указан номер или название группы, то возвращается фрагмент, совпадающий с этой группой. Через запятую можно указать несколько номеров или названий групп. В этом случае возвращается кортеж, содержащий фрагменты, соответствующие группам. Если нет группы с указанным номером или названием, то возбуждается исключение `IndexError`. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m.group(), m.group(0) # Полное соответствие шаблону
('123456string', '123456string')
>>> m.group(1), m.group(2) # Обращение по индексу
('123456', 'string')
>>> m.group("num"), m.group("str") # Обращение по названию
('123456', 'string')
>>> m.group(1, 2), m.group("num", "str") # Несколько параметров
(('123456', 'string'), ('123456', 'string'))
```

- `groupdict([<Значение по умолчанию>])` — возвращает словарь, содержащий значения именованных групп. С помощью необязательного параметра можно указать значение, которое будет выводиться вместо значения `None`, для групп, не имеющих совпадений:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groupdict()
{'num': '123456', 'str': None}
>>> m.groupdict("")
{'num': '123456', 'str': ''}
```

- `groups([<Значение по умолчанию>])` — возвращает кортеж, содержащий значения всех групп. С помощью необязательного параметра можно указать значение, которое будет выводиться вместо значения `None`, для групп, не имеющих совпадений:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z])?")
>>> m = p.search("123456")
>>> m.groups()
('123456', None)
>>> m.groups("")
('123456', '')
```

- `start([<Номер группы или название>])` — возвращает индекс начала фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае — соответствие с указанной группой. Если соответствия нет, то возвращается значение `-1`;
- `end([<Номер группы или название>])` — возвращает индекс конца фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае — соответствие с указанной группой. Если соответствия нет, то возвращается значение `-1`;
- `span([<Номер группы или название>])` — возвращает кортеж, содержащий начальный и конечный индексы фрагмента. Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае — соответствие с указанной группой. Если соответствия нет, то возвращается значение `(-1, -1)`. Примеры:

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> s = "str123456str"
>>> m = p.search(s)
>>> m.start(), m.end(), m.span()
(3, 12, (3, 12))
>>> m.start(1), m.end(1), m.start("num"), m.end("num")
(3, 9, 3, 9)
>>> m.start(2), m.end(2), m.start("str"), m.end("str")
(9, 12, 9, 12)
>>> m.span(1), m.span("num"), m.span(2), m.span("str")
((3, 9), (3, 9), (9, 12), (9, 12))
>>> s[m.start(1):m.end(1)], s[m.start(2):m.end(2)]
('123456', 'str')
```

- `expand(<Новая строка>)` — производит замену в строке. Внутри указанной строки можно использовать обратные ссылки: `\номер`, `\g<номер>` и `\g<название>`. В качестве примера поменяем два тега местами:

```
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> m = p.search("<br><hr>")
>>> m.expand(r"<\2><\1>")              # \номер
'<hr><br>'
>>> m.expand(r"<\g<2>><\g<1>>")        # \g<номер>
'<hr><br>'
>>> m.expand(r"<\g<tag2>><\g<tag1>>") # \g<название>
'<hr><br>'
```

В качестве примера использования метода `search()` проверим e-mail, введенный пользователем, на соответствие шаблону (листинг 7.13).

Листинг 7.13. Проверка e-mail на соответствие шаблону

```
# -*- coding: cp1251 -*-
import re
email = raw_input("Введите e-mail: ")
```

```
pe = r"^([a-z0-9_.-]+)@(([a-z0-9-]+\.)+[a-z]{2,6})$"
p = re.compile(pe, re.I)
m = p.search(email)
if not m:
    print "E-mail не соответствует шаблону"
else:
    print "E-mail", m.group(0), "соответствует шаблону"
    print "ящик:", m.group(1), "домен:", m.group(2)
```

Результат выполнения:

```
Введите e-mail: unicross@mail.ru
E-mail unicross@mail.ru соответствует шаблону
ящик: unicross домен: mail.ru
```

7.3. Поиск всех совпадений с шаблоном

Для поиска всех совпадений с шаблоном предназначены следующие функции и методы:

- `findall()` — ищет все совпадения с шаблоном. Формат метода:

  ```
  findall(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствия найдены, то возвращается список с фрагментами, в противном случае возвращается пустой список. Если внутри шаблона есть более одной группы, то каждый элемент списка будет кортежем, а не строкой. Примеры:

  ```
  >>> import re
  >>> p = re.compile(r"[0-9]+")
  >>> p.findall("2007, 2008, 2009, 2010, 2011")
  ['2007', '2008', '2009', '2010', '2011']
  >>> p = re.compile(r"[a-z]+")
  >>> p.findall("2007, 2008, 2009, 2010, 2011")
  []
  >>> p = re.compile(r"((\d{3})-(\d{2})-(\d{2}))")
  >>> p.findall("322-77-20, 528-22-98")
  [('322-77-20', '322', '77', '20'),
   ('528-22-98', '528', '22', '98')]
  ```

  Вместо метода `findall()` можно воспользоваться функцией `findall()`. Формат функции:

  ```
  re.findall(<Шаблон>, <Строка>[, <Модификатор>])
  ```

  В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`.

Пример:

```
>>> p = r"[0-9]+"
>>> re.findall(p, "1 2 3 4 5 6")
['1', '2', '3', '4', '5', '6']
>>> p = re.compile(r"[0-9]+")
>>> re.findall(p, "1 2 3 4 5 6")
['1', '2', '3', '4', '5', '6']
```

- `finditer()` — аналогичен методу `findall()`, но возвращает итератор, а не список. На каждой итерации цикла возвращается объект `MatchObject`. Формат метода:

```
finditer(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
```

Пример:

```
>>> p = re.compile(r"[0-9]+")
>>> for m in p.finditer("2007, 2008, 2009, 2010, 2011"):
       print m.group(0), "start:", m.start(), "end:", m.end()

2007 start: 0 end: 4
2008 start: 6 end: 10
2009 start: 12 end: 16
2010 start: 18 end: 22
2011 start: 24 end: 28
```

Вместо метода `finditer()` можно воспользоваться функцией `finditer()`. Формат функции:

```
re.finditer(<Шаблон>, <Строка>[, <Модификатор>])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. В параметре `<Модификатор>` можно указать флаги, используемые в функции `compile()`. Получим содержимое между тегами:

```
>>> p = re.compile(r"<b>(.+?)</b>", re.I | re.S)
>>> s = "<b>Text1</b>Text2<b>Text3</b>"
>>> for m in re.finditer(p, s):
        print m.group(1)
Text1
Text3
```

7.4. Замена в строке

Для замены в строке с помощью регулярных выражений предназначены следующие функции и методы:

- `sub()` — ищет все совпадения с шаблоном и заменяет их указанным значением. Если совпадения не найдены, возвращается исходная строка.

Метод имеет следующий формат:

```
sub(<Новый фрагмент или ссылка на функцию>, <Строка для замены>
    [, <Максимальное количество замен>])
```

Внутри нового фрагмента можно использовать обратные ссылки `\номер`, `\g<номер>` и `\g<название>`, соответствующие группам внутри шаблона. В качестве примера поменяем два тега местами:

```
>>> import re
>>> p = re.compile(r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>")
>>> p.sub(r"<\2><\1>", "<br><hr>")             # \номер
'<hr><br>'
>>> p.sub(r"<\g<2>><\g<1>>", "<br><hr>")       # \g<номер>
'<hr><br>'
>>> p.sub(r"<\g<tag2>><\g<tag1>>", "<br><hr>") # \g<название>
'<hr><br>'
```

В качестве первого параметра можно указать ссылку на функцию. В эту функцию будет передаваться объект `MatchObject`, соответствующий найденному фрагменту. Результат, возвращаемый этой функцией, служит фрагментом для замены. Для примера найдем все числа в строке и прибавим к ним число `10`:

```
# -*- coding: cp1251 -*-
import re
def repl(m):
    """ Функция для замены. m - объект MatchObject """
    x = int(m.group(0))
    x += 10
    return "%s" % x

p = re.compile(r"[0-9]+")
# Заменяем все вхождения
print p.sub(repl, "2008, 2009, 2010, 2011")
# Заменяем только первые два вхождения
print p.sub(repl, "2008, 2009, 2010, 2011", 2)
```

Результат выполнения:

```
2018, 2019, 2020, 2021
2018, 2019, 2010, 2011
```

Внимание!

Название функции указывается без скобок.

Вместо метода `sub()` можно воспользоваться функцией `sub()`. Формат функции:

```
re.sub(<Шаблон>, <Новый фрагмент или ссылка на функцию>,
       <Строка для замены>[, <Максимальное количество замен>])
```

В качестве параметра <Шаблон> можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Поменяем два тега местами, а также изменим регистр букв:

```
# -*- coding: cp1251 -*-
import re
def repl(m):
    """ Функция для замены. m - объект MatchObject """
    tag1 = m.group("tag1").upper()
    tag2 = m.group("tag2").upper()
    return "<%s><%s>" % (tag2, tag1)

p = r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>"
print re.sub(p, repl, "<br><hr>")
```

Результат выполнения:

```
<HR><BR>
```

- `subn()` — аналогичен методу `sub()`, но возвращает не строку, а кортеж из двух элементов — измененной строки и количества произведенных замен. Метод имеет следующий формат:

```
subn(<Новый фрагмент или ссылка на функцию>, <Строка для замены>
    [, <Максимальное количество замен>])
```

Заменим все числа в строке на `0`:

```
>>> p = re.compile(r"[0-9]+")
>>> p.subn("0", "2008, 2009, 2010, 2011")
('0, 0, 0, 0', 4)
```

Вместо метода `subn()` можно воспользоваться функцией `subn()`. Формат функции:

```
re.subn(<Шаблон>, <Новый фрагмент или ссылка на функцию>,
        <Строка для замены>[, <Максимальное количество замен>])
```

В качестве параметра <Шаблон> можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Пример:

```
>>> p = r"200[79]"
>>> re.subn(p, "2001", "2007, 2008, 2009, 2010")
('2001, 2008, 2001, 2010', 2)
```

7.5. Прочие функции и методы

Метод `split()` разбивает строку по шаблону и возвращает список подстрок. Если во втором параметре задано число, то в списке будет указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент, в котором будет остаток строки.

Метод имеет следующий формат:

```
split(<Исходная строка>[, <Лимит>])
```

Пример:

```
>>> import re
>>> p = re.compile(r"[\s,.]+")
>>> p.split("word1, word2\nword3\r\nword4.word5")
['word1', 'word2', 'word3', 'word4', 'word5']
>>> p.split("word1, word2\nword3\r\nword4.word5", 2)
['word1', 'word2', 'word3\r\nword4.word5']
```

Если разделитель не найден в строке, то список будет содержать только один элемент, содержащий исходную строку:

```
>>> p = re.compile(r"[0-9]+")
>>> p.split("word, word\nword")
['word, word\nword']
```

Вместо метода `split()` можно воспользоваться функцией `split()`. Формат функции:

```
re.split(<Шаблон>, <Исходная строка>[, <Лимит>])
```

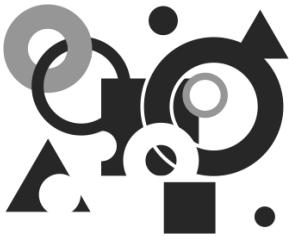
В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Пример:

```
>>> p = re.compile(r"[\s,.]+")
>>> re.split(p, "word1, word2\nword3")
['word1', 'word2', 'word3']
>>> re.split(r"[\s,.]+", "word1, word2\nword3")
['word1', 'word2', 'word3']
```

С помощью функции `escape(<Строка>)` можно экранировать все специальные символы в строке, полученной от пользователя. Эту строку в дальнейшем можно безопасно использовать внутри регулярного выражения. Пример:

```
>>> print re.escape(r"[]().*")
\[\]\(\)\.\*
```

Глава 8



Списки, кортежи и множества

Списки и *кортежи* — это нумерованные наборы объектов. Каждый элемент набора содержит лишь ссылку на объект. По этой причине списки и кортежи могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Позиция элемента в наборе задается *индексом*. Обратите внимание на то, что нумерация элементов начинается с `0`, а не с `1`.

Списки и кортежи являются упорядоченными последовательностями элементов. Как и все последовательности, они поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`).

Списки относятся к изменяемым типам данных. Это означает, что мы можем не только получить элемент по индексу, но и изменить его:

```
>>> arr = [1, 2, 3]          # Создаем список
>>> arr[0]                   # Получаем элемент по индексу
1
>>> arr[0] = 50              # Изменяем элемент по индексу
>>> arr
[50, 2, 3]
```

Кортежи относятся к неизменяемым типам данных. Иными словами, можно получить элемент по индексу, но изменить его нельзя:

```
>>> t = (1, 2, 3)            # Создаем кортеж
>>> t[0]                     # Получаем элемент по индексу
1
>>> t[0] = 50                # Изменить элемент по индексу нельзя!
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    t[0] = 50               # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

Рассмотрим списки, кортежи и множества подробно.

8.1. Создание списка

Создать список можно следующими способами:

- с помощью функции `list([<Последовательность>])`. Функция позволяет преобразовать любую последовательность в список. Если параметр не указан, то создается пустой список. Примеры:

```
>>> list()                    # Создаем пустой список
[]
>>> list("String")            # Преобразуем строку в список
['S', 't', 'r', 'i', 'n', 'g']
>>> list((1, 2, 3, 4, 5))     # Преобразуем кортеж в список
[1, 2, 3, 4, 5]
```

- указав все элементы списка внутри квадратных скобок:

```
>>> arr = [1, "str", 3, "4"]
>>> arr
[1, 'str', 3, '4']
```

- заполнив список поэлементно с помощью метода `append()`:

```
>>> arr = []            # Создаем пустой список
>>> arr.append(1)       # Добавляем элемент1 (индекс 0)
>>> arr.append("str")   # Добавляем элемент2 (индекс 1)
>>> arr
[1, 'str']
```

В некоторых языках программирования (например, в PHP) можно добавить элемент, указав пустые квадратные скобки или индекс больше последнего индекса. В языке Python все эти способы приведут к ошибке:

```
>>> arr = []
>>> arr[] = 10
SyntaxError: invalid syntax
>>> arr[0] = 10
Traceback (most recent call last):
  File "<pyshell#120>", line 1, in <module>
    arr[0] = 10
IndexError: list assignment index out of range
```

При создании списка в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел и строк, но для списков этого делать нельзя. Рассмотрим пример:

```
>>> x = y = [1, 2]        # Якобы создали два объекта
>>> x, y
([1, 2], [1, 2])
```

В этом примере мы создали список из двух элементов и присвоили значение переменным `x` и `y`.

Теперь попробуем изменить значение в переменной `y`:

```
>>> y[1] = 100           # Изменяем второй элемент
>>> x, y                 # Изменилось значение сразу в двух переменных
([1, 100], [1, 100])
```

Как видно из примера, изменение значения в переменной `y` привело также к изменению значения в переменной `x`. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> x, y = [1, 2], [1, 2]
>>> y[1] = 100           # Изменяем второй элемент
>>> x, y
([1, 2], [1, 100])
```

Проверить, ссылаются ли две переменные на один и тот же объект, позволяет оператор `is`. Если переменные ссылаются на один и тот же объект, то оператор `is` возвращает значение `True`:

```
>>> x = y = [1, 2]        # Неправильно
>>> x is y # Переменные содержат ссылку на один и тот же список
True
>>> x, y = [1, 2], [1, 2] # Правильно
>>> x is y                # Это разные объекты
False
```

Как вы уже знаете, операция присваивания сохраняет лишь ссылку на объект, а не сам объект. Что же делать, если необходимо создать копию списка? Первый способ заключается в применении операции извлечения среза, а второй способ — в использовании функции `list()` (листинг 8.1).

Листинг 8.1. Создание поверхностной копии списка

```
>>> x = [1, 2, 3, 4, 5] # Создали список
>>> # Создаем копию списка
>>> y = list(x) # или с помощью среза y = x[:]
>>> y
[1, 2, 3, 4, 5]
>>> x is y # Оператор показывает, что это разные объекты
False
>>> y[1] = 100 # Изменяем второй элемент
>>> x, y       # Изменился только список в переменной y
([1, 2, 3, 4, 5], [1, 100, 3, 4, 5])
```

На первый взгляд может показаться, что мы получили копию. Оператор `is` показывает, что это разные объекты, а изменение элемента затронуло лишь значение переменной `y`. В данном случае вроде все нормально. Но проблема заключается в том, что списки в языке Python могут иметь неограниченную степень вложенности.

Рассмотрим это на примере:

```
>>> x = [1, [2, 3, 4, 5]] # Создали вложенный список
>>> y = list(x)           # Якобы сделали копию списка
>>> x is y                # Разные объекты
False
>>> y[1][1] = 100         # Изменяем элемент
>>> x, y                  # Изменение затронуло переменную x!!!
([1, [2, 100, 4, 5]], [1, [2, 100, 4, 5]])
```

В этом примере мы создали список, в котором второй элемент является вложенным списком. Далее с помощью функции `list()` попытались создать копию списка. Как и в предыдущем примере, оператор `is` показывает, что это разные объекты, но посмотрите на результат. Изменение переменной `y` затронуло и значение переменной `x`. Таким образом, функция `list()` и операция извлечения среза создают лишь *поверхностную копию* списка.

Чтобы получить полную копию списка, следует воспользоваться функцией `deepcopy()` из модуля `copy` (листинг 8.2).

Листинг 8.2. Создание полной копии списка

```
>>> import copy             # Подключаем модуль copy
>>> x = [1, [2, 3, 4, 5]]
>>> y = copy.deepcopy(x)    # Делаем полную копию списка
>>> y[1][1] = 100           # Изменяем второй элемент
>>> x, y                    # Изменился только список в переменной y
([1, [2, 3, 4, 5]], [1, [2, 100, 4, 5]])
```

Функция `deepcopy()` создает копию каждого объекта, при этом сохраняя внутреннюю структуру списка. Иными словами, если в списке существуют два элемента, ссылающиеся на один объект, то будет создана копия объекта и элементы будут ссылаться на этот новый объект, а не на разные объекты. Пример:

```
>>> import copy          # Подключаем модуль copy
>>> x = [1, 2]
>>> y = [x, x]           # Два элемента ссылаются на один объект
>>> y
[[1, 2], [1, 2]]
>>> z = copy.deepcopy(y) # Сделали копию списка
>>> z[0] is x, z[1] is x, z[0] is z[1]
(False, False, True)
>>> z[0][0] = 300        # Изменили один элемент
>>> z                    # Значение изменилось сразу в двух элементах!
[[300, 2], [300, 2]]
>>> x                    # Начальный список не изменился
[1, 2]
```

8.2. Операции над списками

Обращение к элементам списка осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Нумерация элементов списка начинается с нуля. Выведем все элементы списка:

```
>>> arr = [1, "str", 3.2, "4"]
>>> arr[0], arr[1], arr[2], arr[3]
(1, 'str', 3.2000000000000002, '4')
```

С помощью позиционного присваивания можно присвоить значения элементов списка каким-либо переменным. Количество элементов справа и слева от оператора = должно совпадать, иначе будет выведено сообщение об ошибке:

```
>>> x, y, z = [1, 2, 3] # Позиционное присваивание
>>> x, y, z
(1, 2, 3)
>>> x, y = [1, 2, 3]    # Количество элементов должно совпадать
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in <module>
    x, y = [1, 2, 3]    # Количество элементов должно совпадать
ValueError: too many values to unpack
```

Так как нумерация элементов списка начинается с 0, индекс последнего элемента будет на единицу меньше количества элементов. Получить количество элементов списка позволяет функция `len()`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> len(arr)              # Получаем количество элементов
5
>>> arr[len(arr)-1]       # Получаем последний элемент
5
```

Если элемент, соответствующий указанному индексу, отсутствует в списке, то возбуждается исключение `IndexError`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[5]                # Обращение к несуществующему элементу
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    arr[5]                # Обращение к несуществующему элементу
IndexError: list index out of range
```

В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца списка, а точнее сказать, значение вычитается из общего количества элементов списка, чтобы получить положительный индекс:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[-1], arr[len(arr)-1] # Обращение к последнему элементу
(5, 5)
```

Так как списки относятся к изменяемым типам данных, то мы можем изменить элемент по индексу:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[0] = 600              # Изменение элемента по индексу
>>> arr
[600, 2, 3, 4, 5]
```

Кроме того, списки поддерживают операцию извлечения среза, которая возвращает указанный фрагмент списка. Формат операции:

```
[<Начало>:<Конец>:<Шаг>]
```

Все параметры являются необязательными. Если параметр `<Начало>` не указан, то используется значение `0`. Если параметр `<Конец>` не указан, то возвращается фрагмент до конца списка. Следует также заметить, что элемент с индексом, указанным в этом параметре, не входит в возвращаемый фрагмент. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве значения параметров можно указать отрицательные значения.

Теперь рассмотрим несколько примеров. Сначала получим поверхностную копию списка:

```
>>> arr = [1, 2, 3, 4, 5]
>>> m = arr[:]; m # Создаем поверхностную копию и выводим значения
[1, 2, 3, 4, 5]
>>> m is arr      # Оператор is показывает, что это разные объекты
False
```

Теперь выведем символы в обратном порядке:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[::-1]             # Шаг -1
[5, 4, 3, 2, 1]
```

Выведем список без первого и последнего элементов:

```
>>> arr[1:]               # Без первого элемента
[2, 3, 4, 5]
>>> arr[:-1]              # Без последнего элемента
[1, 2, 3, 4]
```

Получим первые два элемента списка:

```
>>> arr[0:2]              # Символ с индексом 2 не входит в диапазон
[1, 2]
```

А теперь получим последний элемент:

```
>>> arr[-1:]              # Последний элемент списка
[5]
```

И, наконец, выведем фрагмент от второго элемента до четвертого включительно:

```
>>> arr[1:4] # Возвращаются элементы с индексами 1, 2 и 3
[2, 3, 4]
```

С помощью среза можно изменить фрагмент списка. Если срезу присвоить пустой список, то элементы, попавшие в срез, будут удалены:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr[1:3] = [6, 7] # Изменяем значения элементов с индексами 1 и 2
>>> arr
[1, 6, 7, 4, 5]
>>> arr[1:3] = []     # Удаляем элементы с индексами 1 и 2
>>> arr
[1, 4, 5]
```

Соединить два списка в один список позволяет оператор `+`. Результатом объединения будет новый список:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = [6, 7, 8, 9]
>>> arr3 = arr1 + arr2
>>> arr3
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Вместо оператора `+` можно использовать оператор `+=`. Следует учитывать, что в этом случае элементы добавляются в текущий список:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr += [6, 7, 8, 9]
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Кроме рассмотренных операций, списки поддерживают операцию повторения и проверку на вхождение. Повторить список указанное количество раз можно с помощью оператора `*`, а выполнить проверку на вхождение элемента в список позволяет оператор `in`:

```
>>> [1, 2, 3] * 3                              # Операция повторения
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5] # Проверка на вхождение
(True, False)
```

8.3. Многомерные списки

Любой элемент списка может содержать объект произвольного типа. Например, элемент списка может быть числом, строкой, списком, кортежем, словарем и т. д. Создать вложенный список можно, например, так:

```
arr = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Как вы уже знаете, выражение внутри скобок может располагаться на нескольких строках. Следовательно, предыдущий пример можно записать иначе:

```
>>> arr = [
              [1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]
          ]
```

Чтобы получить значение элемента во вложенном списке, следует указать два индекса:

```
>>> arr[1][1]
5
```

Элементы вложенного списка также могут иметь элементы произвольного типа. Количество вложений не ограничено. Таким образом, мы можем создать объект любой степени сложности. В этом случае для доступа к элементам указывается несколько индексов подряд. Примеры:

```
>>> arr = [ [1, ["a", "b"], 3], [4, 5, 6], [7, 8, 9] ]
>>> arr[0][1][0]
'a'
>>> mass = [ [1, { "a": 10, "b": ["s", 5] } ] ]
>>> mass[0][1]["b"][0]
's'
```

8.4. Перебор элементов списка

Перебрать все элементы списка можно с помощью цикла `for`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> for i in arr: print i,
```

Результат выполнения:

```
1 2 3 4 5
```

Следует заметить, что переменную `i` внутри цикла можно изменить, но если она ссылается на неизменяемый тип данных (например, число или строку), то это не отразится на исходном списке:

```
>>> arr = [1, 2, 3, 4]       # Элементы имеют неизменяемый тип (число)
>>> for i in arr: i += 10

>>> arr                      # Список не изменился
[1, 2, 3, 4]
>>> arr = [ [1, 2], [3, 4] ] # Элементы имеют изменяемый тип (список)
>>> for i in arr: i[0] += 10

>>> arr                      # Список изменился
[[11, 2], [13, 4]]
```

Чтобы получить доступ к каждому элементу, можно, например, воспользоваться функцией `xrange()` для генерации индексов. В отличие от функции `range()`, которая возвращает список значений, функция `xrange()` возвращает итератор. С помощью этого итератора внутри цикла `for` можно получить текущий индекс. Функция `xrange()` имеет следующий формат:

```
xrange([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если параметр `<Начало>` не указан, то по умолчанию используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый список значений. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве примера умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i in xrange(len(arr)):
    arr[i] *= 2
print arr                    # Результат выполнения: [2, 4, 6, 8]
```

Можно также воспользоваться функцией `enumerate(<Объект>)`, которая на каждой итерации цикла `for` возвращает кортеж из индекса и значения текущего элемента списка. Умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i, elem in enumerate(arr):
    arr[i] *= 2
print arr                    # Результат выполнения: [2, 4, 6, 8]
```

Кроме того, перебрать элементы можно с помощью цикла `while`. Но в этом случае следует помнить, что цикл `while` работает медленнее цикла `for`. В качестве примера умножим каждый элемент списка на 2, используя цикл `while`:

```
arr = [1, 2, 3, 4]
i, c = 0, len(arr)
while i < c:
    arr[i] *= 2
    i += 1
print arr                    # Результат выполнения: [2, 4, 6, 8]
```

8.5. Генераторы списков и выражения-генераторы

В предыдущем разделе мы изменяли элементы списка следующим образом:

```
arr = [1, 2, 3, 4]
for i in xrange(len(arr)):
    arr[i] *= 2
print arr                    # Результат выполнения: [2, 4, 6, 8]
```

С помощью генераторов списков тот же самый код можно записать более компактно. Помимо компактного отображения генераторы списков работают быстрее цикла `for`. Однако вместо изменения исходного списка возвращается новый список:

```
arr = [1, 2, 3, 4]
arr = [ i * 2 for i in arr ]
print arr                    # Результат выполнения: [2, 4, 6, 8]
```

Как видно из примера, мы поместили цикл `for` внутри квадратных скобок, а также изменили порядок следования параметров: выражение, выполняемое внутри цикла, находится перед циклом. Обратите внимание на то, что выражение внутри цикла не содержит оператора присваивания. На каждой итерации цикла будет генерироваться новый элемент, которому неявным образом присваивается результат выполнения выражения внутри цикла. В итоге будет создан новый список, содержащий измененные значения элементов исходного списка.

Генераторы списков могут иметь сложную структуру. Например, состоять из нескольких вложенных циклов `for` и (или) содержать оператор ветвления `if` после цикла. В качестве примера получим четные элементы списка и умножим их на 10:

```
arr = [1, 2, 3, 4]
arr = [ i * 10 for i in arr if i % 2 == 0 ]
print arr                    # Результат выполнения: [20, 40]
```

Последовательность выполнения этого кода эквивалентна последовательности выполнения следующего кода:

```
arr = []
for i in [1, 2, 3, 4]:
    if i % 2 == 0:           # Если число четное
        arr.append(i * 10)   # Добавляем элемент
print arr                    # Результат выполнения: [20, 40]
```

Усложним наш пример. Получим четные элементы вложенного списка и умножим их на 10:

```
arr = [[1, 2], [3, 4], [5, 6]]
arr = [ j * 10 for i in arr for j in i if j % 2 == 0 ]
print arr                    # Результат выполнения: [20, 40, 60]
```

Последовательность выполнения этого кода эквивалентна последовательности выполнения следующего кода:

```
arr = []
for i in [[1, 2], [3, 4], [5, 6]]:
    for j in i:
        if j % 2 == 0:           # Если число четное
            arr.append(j * 10)   # Добавляем элемент
print arr                        # Результат выполнения: [20, 40, 60]
```

Если выражение разместить не внутри квадратных скобок, а внутри круглых скобок, то будет возвращаться не список, а итератор. Такие конструкции называются *выражениями-генераторами*. В качестве примера просуммируем четные числа в списке:

```
>>> arr = [1, 4, 12, 45, 10]
>>> sum( ( i for i in arr if i % 2 == 0 ) )
26
```

8.6. Перебор элементов списка без циклов

Встроенная функция `map()` позволяет перебрать элементы списка без использования циклов. Функция имеет следующий формат:

```
map(<Функция>, <Последовательность1>[, ..., <ПоследовательностьN>])
```

В качестве параметра `<Функция>` указывается ссылка на функцию, которой будет передаваться текущий элемент последовательности. Выведем все элементы списка с помощью функции `map()`:

```
def f_print(elem):
    """ Вывод всех элементов списка """
    print elem,

arr = [1, 2, 3, 4, 5]
map(f_print, arr)
# Результат выполнения: 1 2 3 4 5
```

В качестве значения функция `map()` возвращает новый список. Если в предыдущем примере перед функцией `map()` добавить оператор `print`, то мы получим следующий список:

```
[None, None, None, None, None]
```

Чтобы добавить элемент в этот список, необходимо внутри функции обратного вызова вернуть новое значение. В качестве примера прибавим к каждому элементу списка число 10 (листинг 8.3).

Листинг 8.3. Функция `map()`

```
def f_increment(elem):
    """ Увеличение значения каждого элемента списка """
    return elem + 10 # Возвращаем новое значение

arr = [1, 2, 3, 4, 5]
new_arr = map(f_increment, arr)
print ", ".join( ( str(i) for i in new_arr ) )
# Результат выполнения: 11, 12, 13, 14, 15
```

Функции `map()` можно передать несколько последовательностей. В этом случае в функцию обратного вызова будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковом смещении. Просуммируем элементы трех списков (листинг 8.4).

Листинг 8.4. Суммирование элементов трех списков

```
def f_increment(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3 # Возвращаем новое значение
```

```
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
new_arr = map(f_increment, arr1, arr2, arr3)
print ", ".join( ( str(i) for i in new_arr ) )
# Результат выполнения: 111, 222, 333, 444, 555
```

Если в первом параметре вместо названия функции указать значение `None`, то функция `map()` вернет список, элементами которого будут кортежи. Каждый кортеж будет содержать элементы последовательностей, которые расположены на одинаковом смещении. Пример:

```
>>> map(None, [1, 2, 3], [4, 5, 6], [7, 8, 9])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Если количество элементов в последовательностях будет разным, то вместо недостающих элементов будет вставлено значение `None`:

```
>>> map(None, [1, 2, 3], [4, 6], [7, 8, 9, 10])
[(1, 4, 7), (2, 6, 8), (3, None, 9), (None, None, 10)]
```

Встроенная функция `zip()` также возвращает список, элементами которого являются кортежи. Каждый кортеж содержит элементы последовательностей, которые расположены на одинаковом смещении. Формат функции:

```
zip(<Последовательность1>[, ..., <ПоследовательностьN>])
```

Пример:

```
>>> zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Если количество элементов в последовательностях будет разным, то в список попадут только элементы, которые существуют во всех последовательностях на одинаковом смещении:

```
>>> zip([1, 2, 3], [4, 6], [7, 8, 9, 10])
[(1, 4, 7), (2, 6, 8)]
```

В качестве еще одного примера переделаем нашу программу (листинг 8.4) суммирования элементов трех списков и используем функцию `zip()` вместо функции `map()` (листинг 8.5).

Листинг 8.5. Суммирование элементов трех списков с помощью функции `zip()`

```
arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20, 30, 40, 50]
arr3 = [100, 200, 300, 400, 500]
new_arr = []
for (x, y, z) in zip(arr1, arr2, arr3):
    new_arr.append(x + y + z)
print ", ".join( ( str(i) for i in new_arr ) )
# Результат выполнения: 111, 222, 333, 444, 555
```

Функция `filter()` позволяет выполнить проверку элементов последовательности. Формат функции:

```
filter(<Функция>, <Последовательность>)
```

Если в первом параметре вместо названия функции указать значение `None`, то каждый элемент последовательности будет проверен на соответствие значению `True`. Если элемент в логическом контексте возвращает значение `False`, то он не будет добавлен в возвращаемый результат. В качестве значения для строк возвращается строка, для кортежей — кортеж, а всех остальных последовательностей — список. Пример:

```
>>> filter(None, [1, 0, None, [], 2]) # Возвращается список
[1, 2]
>>> filter(None, (1, 0, None, [], 2)) # Возвращается кортеж
(1, 2)
>>> filter(None, "str0000")           # Возвращается строка
'str0000'
```

В первом параметре можно указать ссылку на функцию. В эту функцию в качестве параметра будет передаваться текущий элемент последовательности. Если элемент следует добавить в возвращаемое функцией `filter()` значение, то внутри функции обратного вызова следует вернуть значение `True`, в противном случае — значение `False`. Удалим все отрицательные значения из списка и кортежа (листинг 8.6).

Листинг 8.6. Пример использования функции `filter()`

```
def f_filter(elem):
    if elem < 0: return False
    return True

arr = [-1, 2, -3, 4, 0, -20, 10] # Список
arr = filter(f_filter, arr)
print arr                        # Результат выполнения: [2, 4, 0, 10]
t = (-1, 2, -3, 4, 0, -20, 10) # Кортеж
t = filter(f_filter, t)
print t                          # Результат выполнения: (2, 4, 0, 10)
```

Функция `reduce()` применяет указанную функцию к парам элементов и накапливает результат. Имеет следующий формат:

```
reduce(<Функция>, <Последовательность>[, <Начальное значение>])
```

В функцию обратного вызова в качестве параметра будут передаваться два элемента. Первый элемент будет содержать результат предыдущих вычислений, а второй — значение текущего элемента. Для примера получим сумму всех элементов списка (листинг 8.7).

Листинг 8.7. Пример использования функции `reduce()`

```
def f_sum(elem1, elem2):
    print "(%s, %s)" % (elem1, elem2),
    return elem1 + elem2

arr = [1, 2, 3, 4, 5]
summa = reduce(f_sum, arr)
# Последовательность: (1, 2) (3, 3) (6, 4) (10, 5)
print summa # Результат выполнения: 15
summa = reduce(f_sum, arr, 10)
# Последовательность: (10, 1) (11, 2) (13, 3) (16, 4) (20, 5)
print summa # Результат выполнения: 25
summa = reduce(f_sum, [], 10)
print summa # Результат выполнения: 10
```

8.7. Добавление и удаление элементов списка

Для добавления и удаления элементов списка используются следующие методы:

- `append(<Объект>)` — добавляет один объект в конец списка. Метод изменяет текущий список и ничего не возвращает. Пример:

```
>>> arr = [1, 2, 3]
>>> arr.append(4); arr          # Добавляем число
[1, 2, 3, 4]
>>> arr.append([5, 6]); arr     # Добавляем список
[1, 2, 3, 4, [5, 6]]
>>> arr.append((7, 8)); arr     # Добавляем кортеж
[1, 2, 3, 4, [5, 6], (7, 8)]
```

- `extend(<Последовательность>)` — добавляет элементы последовательности в конец списка. Метод изменяет текущий список и ничего не возвращает. Пример:

```
>>> arr = [1, 2, 3]
>>> arr.extend([4, 5, 6])       # Добавляем список
>>> arr.extend((7, 8, 9))       # Добавляем кортеж
>>> arr.extend("abc")           # Добавляем буквы из строки
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c']
```

Добавить несколько элементов можно с помощью операции конкатенации:

```
>>> arr = [1, 2, 3]
>>> arr + [4, 5, 6]             # Возвращает новый список
[1, 2, 3, 4, 5, 6]
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> arr = [1, 2, 3]
>>> arr[len(arr):] = [4, 5, 6]  # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6]
```

- `insert(<Индекс>, <Объект>)` — добавляет один объект в указанную позицию. Остальные элементы смещаются. Метод изменяет текущий список и ничего не возвращает. Примеры:

```
>>> arr = [1, 2, 3]
>>> arr.insert(0, 0); arr   # Вставляем 0 в начало списка
[0, 1, 2, 3]
>>> arr.insert(-1, 20); arr # Можно указать отрицательные числа
[0, 1, 2, 20, 3]
>>> arr.insert(2, 100); arr # Вставляем 100 в позицию 2
[0, 1, 100, 2, 20, 3]
>>> arr.insert(10, [4, 5]); arr # Добавляем список
[0, 1, 100, 2, 20, 3, [4, 5]]
```

Метод `insert()` позволяет добавить только один объект. Чтобы добавить несколько объектов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало списка:

```
>>> arr = [1, 2, 3]
>>> arr[:0] = [-2, -1, 0]
>>> arr
[-2, -1, 0, 1, 2, 3]
```

- `pop([<Индекс>])` — удаляет элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, то удаляет и возвращает последний элемент списка. Примеры:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.pop()        # Удаляем последний элемент списка
5
>>> arr              # Список изменился
[1, 2, 3, 4]
>>> arr.pop(0)       # Удаляем первый элемент списка
1
>>> arr              # Список изменился
[2, 3, 4]
```

Удалить элемент списка позволяет также оператор `del`:

```
>>> arr = [1, 2, 3, 4, 5]
>>> del arr[4]; arr  # Удаляем последний элемент списка
[1, 2, 3, 4]
>>> del arr[:2]; arr # Удаляем первый и второй элементы
[3, 4]
```

- `remove(<Значение>)` — удаляет первый элемент, содержащий указанное значение. Если элемент не найден, возбуждается исключение `ValueError`. Метод изменяет текущий список и ничего не возвращает. Примеры:

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.remove(1)    # Удаляет только первый элемент
>>> arr
[2, 3, 1, 1]
>>> arr.remove(5)    # Такого элемента нет
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    arr.remove(5)    # Такого элемента нет
ValueError: list.remove(x): x not in list
```

Если необходимо удалить все повторяющиеся элементы списка, то можно список преобразовать во множество, а затем множество обратно преобразовать в список. Обратите внимание на то, что список должен содержать только числа или строки. В противном случае возбуждается исключение `TypeError`. Пример:

```
>>> arr = [1, 2, 3, 1, 1, 2, 2, 3, 3]
>>> s = set(arr)          # Преобразуем список во множество
>>> s
set([1, 2, 3])
>>> arr = list(s)         # Преобразуем множество в список
>>> arr                   # Все повторы были удалены
[1, 2, 3]
```

8.8. Поиск элемента в списке

Как вы уже знаете, выполнить проверку на вхождение элемента в список позволяет оператор `in`. Если элемент входит в список, то возвращается значение `True`, в противном случае — `False`. Пример:

```
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5] # Проверка на вхождение
(True, False)
```

Тем не менее, оператор `in` не дает никакой информации о местонахождении элемента внутри списка. Чтобы узнать индекс элемента внутри списка, следует воспользоваться методом `index()`. Формат метода:

```
index(<Значение>[, <Начало>[, <Конец>]])
```

Метод `index()` возвращает индекс элемента, имеющего указанное значение. Если значение не входит в список, то возбуждается исключение `ValueError`. Если второй и третий параметры не указаны, то поиск будет производиться с начала списка. Пример:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.index(1), arr.index(2)
(0, 1)
```

```
>>> arr.index(1, 1), arr.index(1, 3, 5)
(2, 4)
>>> arr.index(3)
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    arr.index(3)
ValueError: list.index(x): x not in list
```

Узнать общее количество элементов с указанным значением позволяет метод `count(<Значение>)`. Если элемент не входит в список, то возвращается значение `0`. Пример:

```
>>> arr = [1, 2, 1, 2, 1]
>>> arr.count(1), arr.count(2)
(3, 2)
>>> arr.count(3)                  # Элемент не входит в список
0
```

С помощью функций `max()` и `min()` можно узнать максимальное и минимальное значение списка соответственно. Пример:

```
>>> arr = [1, 2, 3, 4, 5]
>>> max(arr), min(arr)
(5, 1)
```

Функция `any(<Последовательность>)` возвращает значение `True`, если в последовательности существует хоть один элемент, который в логическом контексте возвращает значение `True`. Если последовательность не содержит элементов, возвращается значение `False`. Пример:

```
>>> any([0, None]), any([0, None, 1]), any([])
(False, True, False)
```

Функция `all(<Последовательность>)` возвращает значение `True`, если все элементы последовательности в логическом контексте возвращают значение `True` или последовательность не содержит элементов. Пример:

```
>>> all([0, None]), all([0, None, 1]), all([]), all(["str", 10])
(False, False, True, True)
```

8.9. Переворачивание и перемешивание списка

Метод `reverse()` изменяет порядок следования элементов списка на противоположный. Метод изменяет текущий список и ничего не возвращает. Пример:

```
>>> arr = [1, 2, 3, 4, 5]
>>> arr.reverse()              # Изменяется текущий список
>>> arr
[5, 4, 3, 2, 1]
```

Если необходимо изменить порядок следования и получить новый список, то следует воспользоваться функцией `reversed(<Последовательность>)`. Функция возвращает итератор, который можно преобразовать в список с помощью функции `list()` или генератора списков:

```
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> reversed(arr)
<listreverseiterator object at 0x0186DA70>
>>> list(reversed(arr))              # Использование функции list()
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> for i in reversed(arr): print i, # Вывод с помощью цикла

10 9 8 7 6 5 4 3 2 1
>>> [i for i in reversed(arr)] # Использование генератора списков
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Функция `shuffle(<Список>[, <Число от 0.0 до 1.0>])` из модуля `random` "перемешивает" список случайным образом. Функция перемешивает сам список и ничего не возвращает. Если второй параметр не указан, то используется значение, возвращаемое функцией `random()`. Пример:

```
>>> import random         # Подключаем модуль random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.shuffle(arr)   # Перемешиваем список случайным образом
>>> arr
[2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
```

8.10. Выбор элементов случайным образом

Получить элементы из списка случайным образом позволяют следующие функции из модуля `random`:

- `choice(<Последовательность>)` — возвращает случайный элемент из любой последовательности (строки, списка, кортежа):

  ```
  >>> import random # Подключаем модуль random
  >>> random.choice(["s", "t", "r"]) # Список
  's'
  ```

- `sample(<Последовательность>, <Количество элементов>)` — возвращает список из указанного количества элементов. В этот список попадут элементы из последовательности, выбранные случайным образом. В качестве последовательности можно указать любые объекты, поддерживающие итерации. Пример:

  ```
  >>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  >>> random.sample(arr, 2)
  [7, 10]
  >>> arr                          # Сам список не изменяется
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  ```

8.11. Сортировка списка

Отсортировать список позволяет метод `sort()`. Метод имеет следующие форматы:

```
sort([cmp=None][, key=None][, reverse=False])
sort([<Пользовательская функция>[, <Функция>[, <Порядок элементов>]]])
```

Все параметры являются необязательными. Метод изменяет текущий список и ничего не возвращает. Отсортируем список по возрастанию с параметрами по умолчанию:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort()                              # Изменяет текущий список
>>> arr
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Чтобы отсортировать список по убыванию следует в параметре `reverse` указать значение `True`:

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort(reverse=True)                  # Сортировка по убыванию
>>> arr
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> arr.sort(None, None, True)              # Сортировка по убыванию
>>> arr
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Следует заметить, что стандартная сортировка зависит от регистра символов (листинг 8.8).

Листинг 8.8. Стандартная сортировка

```
# -*- coding: cp1251 -*-
arr = ["единица1", "Единый", "Единица2"]
arr.sort()
for i in arr:
    print i,
# Результат выполнения: Единица2 Единый единица1
```

В результате мы получили неправильную сортировку, ведь `Единый` и `Единица2` больше `единица1`. Чтобы регистр символов не учитывался, можно указать ссылку на функцию для изменения регистра символов в параметре `key` (листинг 8.9).

Листинг 8.9. Пользовательская сортировка

```
# -*- coding: cp1251 -*-
import locale # Настраиваем локаль
locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
```

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=str.lower)                              # Вариант 1
for i in arr:
    print i,
# Результат выполнения: единица1 Единица2 Единый
print # Вставляем перевод строки
arr = ["единица1", "Единый", "Единица2"]
arr.sort(None, str.lower)                            # Вариант 2
for i in arr:
    print i,
# Результат выполнения: единица1 Единица2 Единый
```

Кроме того, в параметре `cmp` можно указать ссылку на пользовательскую функцию сортировки. Функция принимает две переменные и должна возвращать:

- `1` — если первый больше второго;
- `-1` — если второй больше первого;
- `0` — если элементы равны.

Изменим стандартную сортировку на свою сортировку, не учитывающую регистр символов (листинг 8.10).

Листинг 8.10. Сортировка без учета регистра символов

```
# -*- coding: cp1251 -*-
import locale
def f_sort(a, b):
    """ Функция для сортировки без учета регистра """
    a1 = a.lower() # Преобразуем к нижнему регистру
    b1 = b.lower() # Преобразуем к нижнему регистру
    if a1 > b1: return 1
    if a1 < b1: return -1
    return 0

# Настраиваем локаль
locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
arr = ["единица1", "Единый", "Единица2"]
arr.sort(cmp=f_sort) # Функция указывается без скобок!
for i in arr:
    print i,
# Результат выполнения: единица1 Единица2 Единый
print # Вставляем перевод строки
arr = ["единица1", "Единый", "Единица2"]
arr.sort(f_sort) # Можно указать только название функции
for i in arr:
    print i,
# Результат выполнения: единица1 Единица2 Единый
```

Для получения правильной сортировки мы приводим две переменные к одному регистру, а затем производим стандартное сравнение. Заметьте, что регистр самих элементов списка не изменяется, т. к. мы работаем с их копиями.

Метод `sort()` сортирует сам список и не возвращает никакого значения. В некоторых случаях необходимо получить отсортированный список, а текущий список оставить без изменений. Для этого следует воспользоваться функцией `sorted()`. Функция имеет следующий формат:

```
sorted(<Последовательность>[, cmp=None][, key=None][, reverse=False])
```

В первом параметре указывается список, который необходимо отсортировать. Остальные параметры эквивалентны параметрам метода `sort()`. Пример использования функции `sorted()` приведен в листинге 8.11.

Листинг 8.11. Пример использования функции `sorted()`

```
>>> arr = [2, 7, 10, 4, 6, 8, 9, 3, 1, 5]
>>> sorted(arr)                # Возвращает новый список!
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sorted(arr, reverse=True) # Возвращает новый список!
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> arr = ["единица1", "Единый", "Единица2"]
>>> import locale             # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> m = sorted(arr, key=str.lower)
>>> for i in m: print i,
единица1 Единица2 Единый
>>> m = sorted(arr, cmp=lambda a,b: cmp(a.lower(), b.lower()))
>>> for i in m: print i,
единица1 Единица2 Единый
```

8.12. Заполнение списка числами

Создать список, содержащий диапазон чисел, можно с помощью функции `range()`. Функция имеет следующий формат:

```
range([<Начало>, ]<Конец>[, <Шаг>])
```

Первый параметр задает начальное значение. Если параметр `<Начало>` не указан, то по умолчанию используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый список значений. Если параметр `<Шаг>` не указан, то используется значение `1`. В качестве примера заполним список числами от `0` до `10`:

```
>>> range(11)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Создадим список, состоящий из диапазона чисел от 1 до 15:

```
>>> range(1, 16)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Теперь изменим порядок следования чисел на противоположный:

```
>>> range(15, 0, -1)
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Кроме того, можно воспользоваться генераторами списков и функцией `xrange()`, которая имеет такой же формат, как и функция `range()`, но возвращает не список, а итератор. Примеры:

```
>>> [i for i in xrange(11)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [i for i in xrange(1, 16)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> [i for i in xrange(15, 0, -1)]
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Если необходимо получить список со случайными числами (или случайными элементами из другого списка), то следует воспользоваться функцией `sample(<Последовательность>, <Количество элементов>)` из модуля `random`. Пример:

```
>>> import random
>>> arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.sample(arr, 3)
[1, 9, 5]
>>> random.sample(xrange(300), 5)
[259, 294, 142, 292, 245]
```

8.13. Преобразование списка в строку

Преобразовать список в строку позволяет метод `join()`. Элементы добавляются через указанный разделитель. Формат метода:

```
<Строка> = <Разделитель>.join(<Последовательность>)
```

Пример:

```
>>> arr = ["word1", "word2", "word3"]
>>> " - ".join(arr)
'word1 - word2 - word3'
```

Обратите внимание на то, что элементы списка должны быть строками, иначе возвращается исключение `TypeError`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " - ".join(arr)
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    " - ".join(arr)
TypeError: sequence item 3: expected string, int found
```

Избежать этого исключения можно с помощью выражения-генератора, внутри которого текущий элемент списка преобразуется в строку с помощью функции `str()`:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> " - ".join( ( str(i) for i in arr ) )
'word1 - word2 - word3 - 2'
```

Кроме того, с помощью функции `str()` можно сразу получить строковое представление списка:

```
>>> arr = ["word1", "word2", "word3", 2]
>>> str(arr)
"['word1', 'word2', 'word3', 2]"
```

8.14. Кортежи

Кортежи, так же как и списки, являются упорядоченными последовательностями элементов. Кортежи во многом аналогичны спискам, но имеют одно очень важное отличие — изменить кортеж нельзя. Можно сказать, что кортеж — это список, доступный "только для чтения". Создать кортеж можно следующими способами:

- с помощью функции `tuple([<Последовательность>])`. Функция позволяет преобразовать любую последовательность в кортеж. Если параметр не указан, то создается пустой кортеж. Примеры:

```
>>> tuple()                 # Создаем пустой кортеж
()
>>> tuple("String")         # Преобразуем строку в кортеж
('S', 't', 'r', 'i', 'n', 'g')
>>> tuple([1, 2, 3, 4, 5]) # Преобразуем список в кортеж
(1, 2, 3, 4, 5)
```

- указав все элементы через запятую внутри круглых скобок (или без скобок):

```
>>> t1 = ()         # Создаем пустой кортеж
>>> t2 = (5,)       # Создаем кортеж из одного элемента
>>> t3 = (1, "str", (3, 4)) # Кортеж из трех элементов
>>> t4 = 1, "str", (3, 4)   # Кортеж из трех элементов
>>> t1, t2, t3, t4
((), (5,), (1, 'str', (3, 4)), (1, 'str', (3, 4)))
```

Обратите особое внимание на вторую строку примера. Чтобы создать кортеж из одного элемента, необходимо в конце указать запятую. Именно запятые формируют кортеж, а не круглые скобки. Если внутри круглых скобок нет запятых, то будет создан объект другого типа. Пример:

```
>>> t = (5); type(t)       # Получили число, а не кортеж!
<type 'int'>
>>> t = ("str"); type(t)  # Получили строку, а не кортеж!
<type 'str'>
```

Четвертая строка в предыдущем примере также доказывает, что не скобки формируют кортеж, а запятые. Помните, что любое выражение в языке Python можно заключить в круглые скобки, а чтобы получить кортеж, необходимо указать запятые.

Позиция элемента в кортеже задается *индексом*. Обратите внимание на то, что нумерация элементов кортежа (как и списка) начинается с `0`, а не с `1`. Как и все последовательности, кортежи поддерживают обращение к элементу по индексу, получение среза, конкатенацию (оператор `+`), повторение (оператор `*`), проверку на вхождение (оператор `in`). Примеры:

```
>>> t = (1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t[0]                  # Получаем значение первого элемента кортежа
1
>>> t[::-1]               # Изменяем порядок следования элементов
(9, 8, 7, 6, 5, 4, 3, 2, 1)
>>> t[2:5]                # Получаем срез
(3, 4, 5)
>>> 8 in t, 0 in t        # Проверка на вхождение
(True, False)
>>> (1, 2, 3) * 3         # Повторение
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> (1, 2, 3) + (4, 5, 6) # Конкатенация
(1, 2, 3, 4, 5, 6)
```

Кортежи относятся к неизменяемым типам данных. Иными словами, можно получить элемент по индексу, но изменить его нельзя:

```
>>> t = (1, 2, 3)         # Создаем кортеж
>>> t[0]                  # Получаем элемент по индексу
1
>>> t[0] = 50             # Изменить элемент по индексу нельзя!
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    t[0] = 50            # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

Получить количество элементов кортежа позволяет функция `len()`:

```
>>> t = (1, 2, 3)         # Создаем кортеж
>>> len(t)                # Получаем количество элементов
3
```

Начиная с Python 2.6, кортежи поддерживают два метода:

- `index(<Значение>[, <Начало>[, <Конец>]])` — возвращает индекс элемента, имеющего указанное значение. Если значение не входит в кортеж, возбуждается исключение `ValueError`. Если второй и третий параметры не указаны, то поиск будет производиться с начала кортежа. Пример:

  ```
  >>> t = (1, 2, 1, 2, 1)
  >>> t.index(1), t.index(2)
  (0, 1)
  ```

```
>>> t.index(1, 1), t.index(1, 3, 5)
(2, 4)
>>> t.index(3)
... Фрагмент опущен ...
ValueError: tuple.index(x): x not in list
```

- `count(<Значение>)` — возвращает количество элементов с указанным значением. Если элемент не входит в кортеж, то возвращается значение `0`. Пример:

```
>>> t = (1, 2, 1, 2, 1)
>>> t.count(1), t.count(2)
(3, 2)
>>> t.count(3)              # Элемент не входит в кортеж
0
```

Других методов у кортежей нет. Чтобы произвести операции над кортежами, следует воспользоваться встроенными функциями, предназначенными для работы с последовательностями. Все эти функции мы уже рассматривали при изучении списков.

8.15. Множества

Множество — это неупорядоченная последовательность уникальных элементов, с которой можно сравнивать другие элементы, чтобы определить, принадлежат ли они этому множеству. Объявить множество можно с помощью функции `set()`:

```
>>> s = set()
>>> s
set([])
```

Функция `set()` позволяет также преобразовать элементы последовательности во множество:

```
>>> set("string")                    # Преобразуем строку
set(['g', 'i', 'n', 's', 'r', 't'])
>>> set([1, 2, 3, 4, 5])             # Преобразуем список
set([1, 2, 3, 4, 5])
>>> set((1, 2, 3, 4, 5))             # Преобразуем кортеж
set([1, 2, 3, 4, 5])
>>> set([1, 2, 3, 1, 2, 3]) # Остаются только уникальные элементы
set([1, 2, 3])
```

Перебрать элементы множества позволяет цикл `for`:

```
>>> for i in set([1, 2, 3]): print i,

1 2 3
```

Получить количество элементов множества позволяет функция `len()`:

```
>>> len(set([1, 2, 3]))
3
```

Для работы с множествами предназначены следующие операторы и соответствующие им методы:

- `|` и `union()` — объединение множеств:

```
>>> s = set([1, 2, 3])
>>> s.union(set([4, 5, 6])), s | set([4, 5, 6])
(set([1, 2, 3, 4, 5, 6]), set([1, 2, 3, 4, 5, 6]))
```

  Если элемент уже содержится во множестве, то он повторно добавлен не будет:

```
>>> set([1, 2, 3]) | set([1, 2, 3])
set([1, 2, 3])
```

- `a |= b` и `a.update(b)` — добавляют элементы множества `b` во множество `a`:

```
>>> s = set([1, 2, 3])
>>> s.update(set([4, 5, 6]))
>>> s
set([1, 2, 3, 4, 5, 6])
>>> s |= set([7, 8, 9])
>>> s
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- `-` и `difference()` — разница множеств:

```
>>> set([1, 2, 3]) - set([1, 2, 4])
set([3])
>>> s = set([1, 2, 3])
>>> s.difference(set([1, 2, 4]))
set([3])
```

- `a -= b` и `a.difference_update(b)` — удаляют элементы из множества `a`, которые существуют и во множестве `a`, и во множестве `b`:

```
>>> s = set([1, 2, 3])
>>> s.difference_update(set([1, 2, 4]))
>>> s
set([3])
>>> s -= set([3, 4, 5])
>>> s
set([])
```

- `&` и `intersection()` — пересечение множеств. Позволяет получить элементы, которые существуют в обоих множествах:

```
>>> set([1, 2, 3]) & set([1, 2, 4])
set([1, 2])
>>> s = set([1, 2, 3])
>>> s.intersection(set([1, 2, 4]))
set([1, 2])
```

- `a &= b` и `a.intersection_update(b)` — во множестве `a` останутся элементы, которые существуют и во множестве `a`, и во множестве `b`:

```
>>> s = set([1, 2, 3])
>>> s.intersection_update(set([1, 2, 4]))
```

```
>>> s
set([1, 2])
>>> s &= set([1, 6, 7])
>>> s
set([1])
```

- `^` и `symmetric_difference()` — возвращают все элементы обоих множеств, исключая одинаковые элементы:

```
>>> s = set([1, 2, 3])
>>> s ^ set([1, 2, 4]), s.symmetric_difference(set([1, 2, 4]))
(set([3, 4]), set([3, 4]))
>>> s ^ set([1, 2, 3]), s.symmetric_difference(set([1, 2, 3]))
(set([]), set([]))
>>> s ^ set([4, 5, 6]), s.symmetric_difference(set([4, 5, 6]))
(set([1, 2, 3, 4, 5, 6]), set([1, 2, 3, 4, 5, 6]))
```

- `a ^= b` и `a.symmetric_difference_update(b)` — во множестве `a` будут все элементы обоих множеств, исключая одинаковые элементы:

```
>>> s = set([1, 2, 3])
>>> s.symmetric_difference_update(set([1, 2, 4]))
>>> s
set([3, 4])
>>> s ^= set([3, 5, 6])
>>> s
set([4, 5, 6])
```

Операторы сравнения множеств:

- `in` — проверка наличия элемента во множестве:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 in s, 12 in s, 12 not in s
(True, False, True)
```

- `==` — проверка на равенство:

```
>>> set([1, 2, 3]) == set([1, 2, 3])
True
>>> set([1, 2, 3]) == set([3, 2, 1])
True
>>> set([1, 2, 3]) == set([1, 2, 3, 4])
False
```

- `a <= b` и `a.issubset(b)` — проверяют, входят ли все элементы множества `a` во множество `b`:

```
>>> s = set([1, 2, 3])
>>> s <= set([1, 2]), s <= set([1, 2, 3, 4])
(False, True)
>>> s.issubset(set([1, 2])), s.issubset(set([1, 2, 3, 4]))
(False, True)
```

- `a < b` — проверяет, входят ли все элементы множества `a` во множество `b`. Причем множество `a` не должно быть равно множеству `b`:

```
>>> s = set([1, 2, 3])
>>> s < set([1, 2, 3]), s < set([1, 2, 3, 4])
(False, True)
```

- `a >= b` и `a.issuperset(b)` — проверяют, входят ли все элементы множества `b` во множество `a`:

```
>>> s = set([1, 2, 3])
>>> s >= set([1, 2]), s >= set([1, 2, 3, 4])
(True, False)
>>> s.issuperset(set([1, 2])), s.issuperset(set([1, 2, 3, 4]))
(True, False)
```

- `a > b` — проверяет, входят ли все элементы множества `b` во множество `a`. Причем множество `a` не должно быть равно множеству `b`:

```
>>> s = set([1, 2, 3])
>>> s > set([1, 2]), s > set([1, 2, 3])
(True, False)
```

Для работы с множествами предназначены следующие методы:

- `copy()` — создает копию множества. Обратите внимание на то, что оператор `=` присваивает лишь ссылку на тот же объект, а не копирует его. Пример:

```
>>> s = set([1, 2, 3])
>>> c = s; s is c # С помощью = копию создать нельзя!
True
>>> c = s.copy()  # Создаем копию объекта
>>> c
set([1, 2, 3])
>>> s is c        # Теперь это разные объекты
False
```

- `add(<Элемент>)` — добавляет `<Элемент>` во множество:

```
>>> s = set([1, 2, 3])
>>> s.add(4); s
set([1, 2, 3, 4])
```

- `remove(<Элемент>)` — удаляет `<Элемент>` из множества. Если элемент не найден, то возбуждается исключение `KeyError`:

```
>>> s = set([1, 2, 3])
>>> s.remove(3); s        # Элемент существует
set([1, 2])
>>> s.remove(5)           # Элемент НЕ существует
Traceback (most recent call last):
  File "<pyshell#78>", line 1, in <module>
    s.remove(5)           # Элемент НЕ существует
KeyError: 5
```

- `discard(<Элемент>)` — удаляет `<Элемент>` из множества, если он присутствует:

```
>>> s = set([1, 2, 3])
>>> s.discard(3); s       # Элемент существует
set([1, 2])
>>> s.discard(5); s       # Элемент НЕ существует
set([1, 2])
```

- `pop()` — удаляет произвольный элемент из множества и возвращает его. Если элементов нет, то возбуждается исключение `KeyError`:

```
>>> s = set([1, 2])
>>> s.pop(), s
(1, set([2]))
>>> s.pop(), s
(2, set([]))
>>> s.pop() # Если нет элементов, то будет ошибка
Traceback (most recent call last):
  File "<pyshell#89>", line 1, in <module>
    s.pop() # Если нет элементов, то будет ошибка
KeyError: 'pop from an empty set'
```

- `clear()` — удаляет все элементы из множества:

```
>>> s = set([1, 2, 3])
>>> s.clear(); s
set([])
```

В языке Python существует еще один тип множеств — `frozenset`. В отличие от типа `set`, множество типа `frozenset` нельзя изменить. Объявить множество можно с помощью функции `frozenset()`:

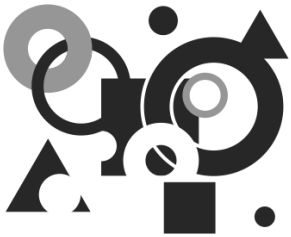
```
>>> f = frozenset()
>>> f
frozenset([])
```

Функция `frozenset()` позволяет также преобразовать элементы последовательности во множество:

```
>>> frozenset("string")                    # Преобразуем строку
frozenset(['g', 'i', 'n', 's', 'r', 't'])
>>> frozenset([1, 2, 3, 4, 4])             # Преобразуем список
frozenset([1, 2, 3, 4])
>>> frozenset((1, 2, 3, 4, 4))             # Преобразуем кортеж
frozenset([1, 2, 3, 4])
```

Множества `frozenset` поддерживают операторы, которые не изменяют само множество, а также следующие методы: `copy()`, `difference()`, `intersection()`, `issubset()`, `issuperset()`, `symmetric_difference()` и `union()`.

Глава 9



Словари

Словари — это наборы объектов, доступ к которым осуществляется не по индексу, а по ключу. В качестве ключа можно указать неизменяемый объект, например число, строку или кортеж. Элементы словаря могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Следует также заметить, что элементы в словарях располагаются в произвольном порядке. Чтобы получить элемент, необходимо указать ключ, который использовался при сохранении значения.

Словари относятся к отображениям, а не к последовательностям. По этой причине функции, предназначенные для работы с последовательностями, а также операции извлечения среза, конкатенации, повторения и др. к словарям не применимы. Также как и списки, словари относятся к изменяемым типам данных. Иными словами, мы можем не только получить значение по ключу, но и изменить его.

9.1. Создание словаря

Создать словарь можно следующими способами:

- с помощью функции `dict()`. Форматы функции:

```
dict(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
dict(<Словарь>)
dict(<Список кортежей с двумя элементами (Ключ, Значение)>)
dict(<Список списков с двумя элементами [Ключ, Значение]>)
```

Если параметры не указаны, то создается пустой словарь. Примеры:

```
>>> d = dict(); d                  # Создаем пустой словарь
{}
>>> d = dict(a=1, b=2); d
{'a': 1, 'b': 2}
>>> d = dict({"a": 1, "b": 2}); d     # Словарь
{'a': 1, 'b': 2}
```

```
>>> d = dict([("a", 1), ("b", 2)]); d # Список кортежей
{'a': 1, 'b': 2}
>>> d = dict([["a", 1], ["b", 2]]); d # Список списков
{'a': 1, 'b': 2}
```

Объединить два списка в список кортежей позволяет функция `zip()`:

```
>>> k = ["a", "b"]          # Список с ключами
>>> v = [1, 2]              # Список со значениями
>>> zip(k, v)               # Создание списка кортежей
[('a', 1), ('b', 2)]
>>> d = dict(zip(k, v)); d # Создание словаря
{'a': 1, 'b': 2}
```

- указав все элементы словаря внутри фигурных скобок. Это наиболее часто используемый способ создания словаря. Между ключом и значением указывается двоеточие, а пары "ключ/значение" перечисляются через запятую. Пример:

```
>>> d = {}; d               # Создание пустого словаря
{}
>>> d = { "a": 1, "b": 2 }; d
{'a': 1, 'b': 2}
```

- заполнив словарь поэлементно. В этом случае ключ указывается внутри квадратных скобок:

```
>>> d = {}             # Создаем пустой словарь
>>> d["a"] = 1         # Добавляем элемент1 (ключ "a")
>>> d["b"] = 2         # Добавляем элемент2 (ключ "b")
>>> d
{'a': 1, 'b': 2}
```

- с помощью метода `dict.fromkeys(<Последовательность>[, <Значение>])`. Метод создает новый словарь, ключами которого будут элементы последовательности. Если второй параметр не указан, то значением элементов словаря будет значение `None`. Пример:

```
>>> d = dict.fromkeys(["a", "b", "c"])
>>> d
{'a': None, 'c': None, 'b': None}
>>> d = dict.fromkeys(["a", "b", "c"], 0) # Указан список
>>> d
{'a': 0, 'c': 0, 'b': 0}
>>> d = dict.fromkeys(("a", "b", "c"), 0) # Указан кортеж
>>> d
{'a': 0, 'c': 0, 'b': 0}
```

При создании словаря в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание можно использовать для чисел и строк, но для списков и словарей этого делать нельзя. Рассмотрим пример:

```
>>> d1 = d2 = { "a": 1, "b": 2 } # Якобы создали два объекта
>>> d2["b"] = 10
```

```
>>> d1, d2                # Изменилось значение в двух переменных !!!
({'a': 1, 'b': 10}, {'a': 1, 'b': 10})
```

Как видно из примера, изменение значения в переменной `d2` привело также к изменению значения в переменной `d1`. Таким образом, обе переменные ссылаются на один и тот же объект, а не на два разных объекта. Чтобы получить два объекта, необходимо производить раздельное присваивание:

```
>>> d1, d2 = { "a": 1, "b": 2 }, { "a": 1, "b": 2 }
>>> d2["b"] = 10
>>> d1, d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Создать поверхностную копию словаря позволяет функция `dict()` (листинг 9.1).

Листинг 9.1. Создание поверхностной копии словаря с помощью функции `dict()`

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = dict(d1)           # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
False
>>> d2["b"] = 10
>>> d1, d2 # Изменилось только значение в переменной d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Кроме того, для создания поверхностной копии можно воспользоваться методом `copy()` (листинг 9.2).

Листинг 9.2. Создание поверхностной копии словаря с помощью метода `copy()`

```
>>> d1 = { "a": 1, "b": 2 } # Создаем словарь
>>> d2 = d1.copy()          # Создаем поверхностную копию
>>> d1 is d2 # Оператор показывает, что это разные объекты
False
>>> d2["b"] = 10
>>> d1, d2   # Изменилось только значение в переменной d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Чтобы создать полную копию словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy` (листинг 9.3).

Листинг 9.3. Создание полной копии словаря

```
>>> d1 = { "a": 1, "b": [20, 30, 40] }
>>> d2 = dict(d1)          # Создаем поверхностную копию
>>> d2["b"][0] = "test"
>>> d1, d2                 # Изменились значения в двух переменных!!!
```

```
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 30, 40]})
>>> import copy
>>> d3 = copy.deepcopy(d1) # Создаем полную копию
>>> d3["b"][1] = 800
>>> d1, d3                 # Изменилось значение только в переменной d3
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 800, 40]})
```

9.2. Операции над словарями

Обращение к элементам словаря осуществляется с помощью квадратных скобок, в которых указывается ключ. В качестве ключа можно указать неизменяемый объект, например, число, строку или кортеж. Выведем все элементы словаря:

```
>>> d = { 1: "int", "a": "str", (1, 2): "tuple" }
>>> d[1], d["a"], d[(1, 2)]
('int', 'str', 'tuple')
```

Если элемент, соответствующий указанному ключу, отсутствует в словаре, то возбуждается исключение `KeyError`:

```
>>> d = { "a": 1, "b": 2 }
>>> d["c"]              # Обращение к несуществующему элементу
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    d["c"]             # Обращение к несуществующему элементу
KeyError: 'c'
```

Проверить существование ключа можно с помощью оператора `in` или метода `has_key()`. Если ключ найден, то возвращается значение `True`, в противном случае — `False`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d, d.has_key("b") # Ключ существует
(True, True)
>>> "c" in d, d.has_key("c") # Ключ не существует
(False, False)
```

Метод `get(<Ключ>[, <Значение по умолчанию>])` позволяет избежать вывода сообщения об ошибке при отсутствии указанного ключа. Если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается значение `None` или значение, указанное во втором параметре. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```

Кроме того, можно воспользоваться методом `setdefault(<Ключ>[, <Значение по умолчанию>])`. Если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то вставляет новый

элемент со значением, указанным во втором параметре. Если второй параметр не указан, значением нового элемента будет `None`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"), d.setdefault("c"), d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'c': None, 'b': 2, 'd': 0}
```

Так как словари относятся к изменяемым типам данных, то мы можем изменить элемент по ключу. Если элемент с указанным ключом отсутствует в словаре, то он будет добавлен в словарь:

```
>>> d = { "a": 1, "b": 2 }
>>> d["a"] = 800               # Изменение элемента по ключу
>>> d["c"] = "string"          # Будет добавлен новый элемент
>>> d
{'a': 800, 'c': 'string', 'b': 2}
```

Получить количество ключей в словаре позволяет функция `len()`:

```
>>> d = { "a": 1, "b": 2 }
>>> len(d)          # Получаем количество ключей в словаре
2
```

Удалить элемент из словаря можно с помощью оператора `del`:

```
>>> d = { "a": 1, "b": 2 }
>>> del d["b"]; d   # Удаляем элемент с ключом "b" и выводим словарь
{'a': 1}
```

9.3. Перебор элементов словаря

Перебрать все элементы списка можно с помощью цикла `for`, хотя словари и не являются последовательностями. В качестве примера выведем элементы словаря двумя способами. Первый способ использует метод `keys()`, возвращающий список всех ключей словаря. Второй способ доступен в последних версиях Python. В этом случае мы просто указываем словарь в качестве параметра. На каждой итерации цикла будет возвращаться ключ, с помощью которого внутри цикла можно получить значение, соответствующее этому ключу (листинг 9.4).

Листинг 9.4. Перебор элементов словаря

```
d = {"x": 1, "y": 2, "z": 3}
for key in d.keys():          # Использование метода keys()
    print "(%s => %s)" % (key, d[key]),
# Выведет: (y => 2) (x => 1) (z => 3)
print                         # Вставляем символ перевода строки
for key in d:                 # Словари также поддерживают итерации
    print "(%s => %s)" % (key, d[key]),
# Выведет: (y => 2) (x => 1) (z => 3)
```

Так как словари являются неупорядоченными структурами, элементы словаря выводятся в произвольном порядке. Чтобы вывести элементы с сортировкой по ключам, следует получить список ключей, а затем воспользоваться методом `sort()`. Пример:

```
d = {"x": 1, "y": 2, "z": 3}
k = d.keys()                    # Получаем список ключей
k.sort()                        # Сортируем список ключей
for key in k:
    print "(%s => %s)" % (key, d[key]),
# Выведет: (x => 1) (y => 2) (z => 3)
```

Для сортировки ключей вместо метода `sort()` можно воспользоваться функцией `sorted()`. Пример:

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d.keys()):
    print "(%s => %s)" % (key, d[key]),
# Выведет: (x => 1) (y => 2) (z => 3)
```

9.4. Методы для работы со словарями

Для работы со словарями предназначены следующие методы:

- `keys()` и `values()` — позволяют получить список всех ключей и значений соответственно:

  ```
  >>> d = { "a": 1, "b": 2 }
  >>> k = d.keys(); v = d.values()
  >>> k, v
  (['a', 'b'], [1, 2])
  ```

  Можно также воспользоваться методами `iterkeys()` и `itervalues()`, которые возвращают не список ключей и значений, а итератор. Пример:

  ```
  >>> d = { "a": 1, "b": 2 }
  >>> for i in d.iterkeys(): print i,   # Ключи
  a b
  >>> for i in d.itervalues(): print i, # Значения
  1 2
  ```

- `items()` — возвращает список кортежей. Каждый кортеж содержит ключ и значение:

  ```
  >>> d = { "a": 1, "b": 2 }
  >>> d.items()               # Получаем список кортежей
  [('a', 1), ('b', 2)]
  ```

  Можно также воспользоваться методом `iteritems()`, который возвращает итератор. Пример:

  ```
  >>> d = { "a": 1, "b": 2 }
  >>> for k, v in d.iteritems(): print "(%s => %s)" % (k, v),
  (a => 1) (b => 2)
  ```

- `has_key(<Ключ>)` — проверяет существование указанного ключа в словаре. Если ключ найден, то возвращается значение `True`, в противном случае — `False`. Вместо метода `has_key()` можно воспользоваться оператором `in`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d, d.has_key("b")   # Ключ существует
(True, True)
>>> "c" in d, d.has_key("c")   # Ключ не существует
(False, False)
```

- `get(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается значение `None` или значение, указанное во втором параметре. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```

- `setdefault(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то вставляет новый элемент со значением, указанным во втором параметре. Если второй параметр не указан, значением нового элемента будет `None`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.setdefault("a"),d.setdefault("c"),d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'c': None, 'b': 2, 'd': 0}
```

- `pop(<Ключ>[, <Значение по умолчанию>])` — удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, то возвращается значение из второго параметра. Если ключ отсутствует и второй параметр не указан, то возбуждается исключение `KeyError`. Пример:

```
>>> d = { "a": 1, "b": 2, "c": 3 }
>>> d.pop("a"), d.pop("n", 0)
(1, 0)
>>> d.pop("n") # Ключ отсутствует и нет второго параметра
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    d.pop("n") # Ключ отсутствует и нет второго параметра
KeyError: 'n'
>>> d
{'c': 3, 'b': 2}
```

- `popitem()` — удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если словарь пустой, возбуждается исключение `KeyError`. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.popitem() # Удаляем произвольный элемент
('a', 1)
>>> d.popitem() # Удаляем произвольный элемент
('b', 2)
```

```
>>> d.popitem() # Словарь пустой. Возбуждается исключение
Traceback (most recent call last):
  File "<pyshell#124>", line 1, in <module>
    d.popitem() # Словарь пустой. Возбуждается исключение
KeyError: 'popitem(): dictionary is empty'
```

- `clear()` — удаляет все элементы словаря. Метод ничего не возвращает в качестве значения. Пример:

```
>>> d = { "a": 1, "b": 2 }
>>> d.clear()               # Удаляем все элементы
>>> d                       # Словарь теперь пустой
{}
```

- `update()` — добавляет элементы в словарь. Метод изменяет текущий словарь и ничего не возвращает. Форматы метода:

```
update(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
update(<Словарь>)
update(<Список кортежей с двумя элементами>)
update(<Список списков с двумя элементами>)
```

Если элемент с указанным ключом уже присутствует в словаре, то его значение будет перезаписано. Примеры:

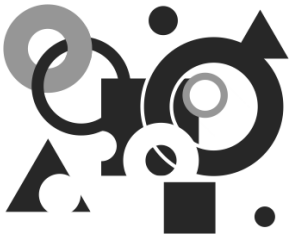
```
>>> d = { "a": 1, "b": 2 }
>>> d.update(c=3, d=4)
>>> d
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
>>> d.update({"c": 10, "d": 20})          # Словарь
>>> d # Значения элементов перезаписаны
{'a': 1, 'c': 10, 'b': 2, 'd': 20}
>>> d.update([("d", 80), ("e", 6)])       # Список кортежей
>>> d
{'a': 1, 'c': 10, 'b': 2, 'e': 6, 'd': 80}
>>> d.update([["a", "str"], ["i", "t"]])  # Список списков
>>> d
{'a': 'str', 'c': 10, 'b': 2, 'e': 6, 'd': 80, 'i': 't'}
```

- `copy()` — создает поверхностную копию словаря:

```
>>> d1 = { "a": 1, "b": [10, 20] }
>>> d2 = d1.copy() # Создаем поверхностную копию
>>> d1 is d2       # Это разные объекты
False
>>> d2["a"] = 800  # Изменяем значение
>>> d1, d2         # Изменилось значение только в d2
({'a': 1, 'b': [10, 20]}, {'a': 800, 'b': [10, 20]})
>>> d2["b"][0] = "new"     # Изменяем значение вложенного списка
>>> d1, d2         # Изменились значения и в d1, и в d2!!!
({'a': 1, 'b': ['new', 20]}, {'a': 800, 'b': ['new', 20]})
```

Чтобы создать полную копию словаря, следует воспользоваться функцией `deepcopy()` из модуля `copy`.

Глава 10



Работа с датой и временем

Для работы с датой и временем в языке Python предназначены следующие модули:

- `time` — позволяет получить текущую дату и время, а также произвести форматированный вывод;
- `datetime` — позволяет производить манипуляции датой и временем. Например, производить арифметические операции, сравнивать даты, выводить дату и время в различных форматах и др.;
- `calendar` — позволяет вывести календарь в виде простого текста или в HTML-формате;
- `timeit` — позволяет измерить время выполнения небольших фрагментов кода с целью оптимизации программы.

10.1. Получение текущей даты и времени

Получить текущую дату и время позволяют следующие функции из модуля `time`:

- `time()` — возвращает вещественное число, представляющее количество секунд, прошедшее с начала эпохи (обычно с 1 января 1970 г.):

```
>>> import time             # Подключаем модуль time
>>> time.time()             # Получаем количество секунд
1275762391.3429999
```

- `gmtime([<Количество секунд>])` — возвращает объект `struct_time`, представляющий универсальное время (UTC). Если параметр не указан, то возвращается текущее время. Если параметр указан, то время будет не текущим, а соответствующим количеству секунд, прошедших с начала эпохи. Пример:

```
>>> time.gmtime(0)                  # Начало эпохи
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
```

```
>>> time.gmtime()                    # Текущая дата и время
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=5, tm_hour=18,
tm_min=39, tm_sec=48, tm_wday=5, tm_yday=156, tm_isdst=0)
>>> time.gmtime(1283307823.0)       # Дата 01-09-2010
time.struct_time(tm_year=2010, tm_mon=9, tm_mday=1, tm_hour=2,
tm_min=23, tm_sec=43, tm_wday=2, tm_yday=244, tm_isdst=0)
```

Получить значение конкретного атрибута можно, указав его название или индекс внутри объекта:

```
>>> d = time.gmtime()
>>> d.tm_year, d[0]
(2010, 2010)
>>> tuple(d)              # Преобразование в кортеж
(2010, 6, 5, 18, 43, 42, 5, 156, 0)
```

- `localtime([<Количество секунд>])` — возвращает объект `struct_time`, представляющий локальное время. Если параметр не указан, то возвращается текущее время. Если параметр указан, то время будет не текущим, а соответствующим количеству секунд, прошедших с начала эпохи. Пример:

```
>>> time.localtime()              # Текущая дата и время
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=5, tm_hour=22,
tm_min=46, tm_sec=31, tm_wday=5, tm_yday=156, tm_isdst=1)
>>> time.localtime(1233368623.0) # Дата 31-01-2009
time.struct_time(tm_year=2009, tm_mon=1, tm_mday=31, tm_hour=5,
tm_min=23, tm_sec=43, tm_wday=5, tm_yday=31, tm_isdst=0)
```

- `mktime(<Объект struct_time>)` — возвращает вещественное число, представляющее количество секунд, прошедших с начала эпохи. В качестве параметра указывается объект `struct_time` или кортеж из девяти элементов. Если указанная дата некорректна, возбуждается исключение `OverflowError`. Пример:

```
>>> tuple(time.localtime(1233368623.0))
(2009, 1, 31, 5, 23, 43, 5, 31, 0)
>>> time.mktime((2009, 1, 31, 5, 23, 43, 5, 31, 0))
1233368623.0
>>> time.mktime((1940, 0, 31, 5, 23, 43, 5, 31, 0))
... Фрагмент опущен ...
OverflowError: mktime argument out of range
```

Объект `struct_time`, возвращаемый функциями `gmtime()` и `localtime()`, содержит следующие атрибуты:

- `tm_year` — `0` — год;
- `tm_mon` — `1` — месяц (число от `1` до `12`);
- `tm_mday` — `2` — день месяца (число от `1` до `31`);
- `tm_hour` — `3` — час (число от `0` до `23`);
- `tm_min` — `4` — минуты (число от `0` до `59`);
- `tm_sec` — `5` — секунды (число от `0` до `59`, изредка до `61`);

- `tm_wday` — `6` — день недели (число от `0` (для понедельника) до `6` (для воскресенья));
- `tm_yday` — `7` — количество дней, прошедшее с начала года (число от `1` до `366`);
- `tm_isdst` — `8` — флаг коррекции летнего времени (значения `0`, `1` или `-1`).

Выведем текущую дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 10.1).

Листинг 10.1. Вывод текущей даты и времени

```
#-*- coding: cp1251 -*-
import time # Подключаем модуль time
d = [ "понедельник", "вторник", "среда", "четверг",
      "пятница", "суббота", "воскресенье" ]
m = [ "", "января", "февраля", "марта", "апреля", "мая",
      "июня", "июля", "августа", "сентября", "октября",
      "ноября", "декабря" ]
t = time.localtime() # Получаем текущее время
print ( "Сегодня:\n%s %s %s %s %02d:%02d:%02d\n%02d.%02d.%02d" %
       ( d[t[6]], t[2], m[t[1]], t[0], t[3], t[4], t[5],
         t[2], t[1], t[0] ) )
```

Результат выполнения:

```
Сегодня:
суббота 5 июня 2010 22:56:20
05.06.2010
```

10.2. Форматирование даты и времени

Получить форматированный вывод даты и времени позволяют следующие функции из модуля `time`:

- `strftime(<Строка формата>[, <Объект struct_time>])` — возвращает строковое представление даты в соответствии со строкой формата. Если второй параметр не указан, будет использоваться текущая дата и время. Если во втором параметре указан объект `struct_time` или кортеж из девяти элементов, то дата будет соответствовать указанному значению. Функция зависит от настройки локали. Пример:

```
>>> import time
>>> time.strftime("%d.%m.%Y") # Форматирование даты
'05.06.2010'
>>> time.strftime("%H:%M:%S") # Форматирование времени
'23:04:34'
>>> time.strftime("%d.%m.%Y", time.localtime(1233368623.0))
'31.01.2009'
```

- `strptime(<Строка с датой>, <Строка формата>)` — разбирает строку, указанную в первом параметре, в соответствии со строкой формата. Возвращает объект `struct_time`. Если строка не соответствует формату, возбуждается исключение `ValueError`. Пример:

```
>>> time.strptime("05.06.2010", "%d.%m.%Y")
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=5, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=5, tm_yday=156, tm_isdst=-1)
>>> time.strptime("05-06-2010", "%d.%m.%Y")
... Фрагмент опущен ...
ValueError: time data '05-06-2010'
does not match format '%d.%m.%Y'
```

- `asctime([<Объект struct_time>])` — возвращает строку специального формата. Если параметр не указан, будет использоваться текущая дата и время. Если в параметре указан объект `struct_time` или кортеж из девяти элементов, то дата будет соответствовать указанному значению. Пример:

```
>>> time.asctime()            # Текущая дата
'Sat Jun 05 23:08:06 2010'
>>> time.asctime(time.localtime(1233368623.0))
'Sat Jan 31 05:23:43 2009'
```

- `ctime([<Количество секунд>])` — функция аналогична `asctime()`, но в качестве параметра принимает не объект `struct_time`, а количество секунд, прошедших с начала эпохи. Пример:

```
>>> time.ctime()                    # Текущая дата
'Sat Jun 05 23:09:10 2010'
>>> time.ctime(1233368623.0)        # Дата в прошлом
'Sat Jan 31 05:23:43 2009'
```

В параметре `<Строка формата>` в функциях `strftime()` и `strptime()` могут быть использованы следующие комбинации специальных символов:

- `%y` — год из двух цифр (от `"00"` до `"99"`);
- `%Y` — год из четырех цифр (например, `"2010"`);
- `%m` — номер месяца с предваряющим нулем (от `"01"` до `"12"`);
- `%b` — аббревиатура месяца в зависимости от настроек локали (например, `"янв"` для января);
- `%B` — название месяца в зависимости от настроек локали (например, `"Январь"`);
- `%d` — номер дня в месяце с предваряющим нулем (от `"01"` до `"31"`);
- `%j` — день с начала года (от `"001"` до `"366"`);
- `%U` — номер недели в году (от `"00"` до `"53"`). Неделя начинается с воскресенья. Все дни с начала года до первого воскресенья относятся к неделе с номером `0`;
- `%W` — номер недели в году (от `"00"` до `"53"`). Неделя начинается с понедельника. Все дни с начала года до первого понедельника относятся к неделе с номером `0`;
- `%w` — номер дня недели (`"0"` — для воскресенья, `"6"` — для субботы);
- `%a` — аббревиатура дня недели в зависимости от настроек локали (например, `"Пн"` для понедельника);

- %A — название дня недели в зависимости от настроек локали (например, "понедельник");
- %H — часы в 24-часовом формате (от "00" до "23");
- %I — часы в 12-часовом формате (от "01" до "12");
- %M — минуты (от "00" до "59");
- %S — секунды (от "00" до "59", изредка до "61");
- %p — эквивалент значениям AM и PM в текущей локали;
- %c — представление даты и времени в текущей локали;
- %x — представление даты в текущей локали;
- %X — представление времени в текущей локали. Пример:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> print time.strftime("%x")   # Представление даты
05.06.2010
>>> print time.strftime("%X")   # Представление времени
23:10:57
>>> print time.strftime("%c")   # Дата и время
05.06.2010 23:11:15
```

- %Z — название часового пояса или пустая строка (например, "Московское время (зима)");
- %% — символ "%".

В качестве примера выведем текущую дату и время с помощью функции strftime() (листинг 10.2).

Листинг 10.2. Форматирование даты и времени

```
#-*- coding: cp1251 -*-
import time
import locale
locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
s = "Сегодня:\n%A %d %b %Y %H:%M:%S\n%d.%m.%Y"
print time.strftime(s)
```

Результат выполнения:

```
Сегодня:
суббота 05 июн 2010 23:12:15
05.06.2010
```

10.3. "Засыпание" скрипта

Функция sleep() из модуля time прерывает выполнение скрипта на указанное время. По истечении срока скрипт продолжит работу. Формат функции:

```
sleep(<Время в секундах>)
```

В качестве параметра можно указать целое или вещественное число. Прежде чем использовать функцию, необходимо подключить модуль `time` с помощью инструкции `import time`:

```
>>> import time                # Подключаем модуль time
>>> time.sleep(5)              # "Засыпаем" на 5 секунд
```

или инструкции `from time import sleep`:

```
>>> from time import sleep     # Подключаем модуль time
>>> sleep(2.5)                 # "Засыпаем" на 2 с половиной секунды
```

10.4. Модуль *datetime*. Манипуляции датой и временем

Модуль `datetime` позволяет производить манипуляции с датой и временем. Например, производить арифметические операции, сравнивать даты, выводить дату и время в различных форматах и др. Прежде чем использовать классы из этого модуля, необходимо подключить модуль с помощью выражения:

```
import datetime
```

Модуль содержит пять классов:

- `timedelta` — дата в виде количества дней, секунд и микросекунд. Экземпляр этого класса можно складывать с экземплярами классов `date` и `datetime`. Кроме того, результат вычитания двух дат будет экземпляром класса `timedelta`;
- `date` — представление даты в виде объекта;
- `time` — представление времени в виде объекта;
- `datetime` — представление комбинации даты и времени в виде объекта;
- `tzinfo` — абстрактный класс, отвечающий за зону времени. За подробной информацией по этому классу обращайтесь к документации по модулю `datetime`.

10.4.1. Класс *timedelta*

Класс `timedelta` из модуля `datetime` позволяет производить операции над датами, например, складывать, вычитать, сравнивать и др. Конструктор класса имеет следующий формат:

```
timedelta([days[, seconds[, microseconds[, milliseconds[, minutes
          [, hours[, weeks]]]]]]])
```

Все параметры являются необязательными и по умолчанию имеют значение `0`. Первые три параметра являются основными:

- `days` — дни (диапазон `-999999999 <= days <= 999999999`);
- `seconds` — секунды (диапазон `0 <= seconds < 3600*24`);
- `microseconds` — микросекунды (диапазон `0 <= microseconds < 1000000`);

Все остальные параметры автоматически преобразуются в следующие значения:

- `milliseconds` — миллисекунды (одна миллисекунда преобразуется в `1000` микросекунд):

```
>>> import datetime
>>> datetime.timedelta(milliseconds=1)
datetime.timedelta(0, 0, 1000)
```

- `minutes` — минуты (одна минута преобразуется в `60` секунд):

```
>>> datetime.timedelta(minutes=1)
datetime.timedelta(0, 60)
```

- `hours` — часы (один час преобразуется в `3600` секунд):

```
>>> datetime.timedelta(hours=1)
datetime.timedelta(0, 3600)
```

- `weeks` — недели (одна неделя преобразуется в `7` дней):

```
>>> datetime.timedelta(weeks=1)
datetime.timedelta(7)
```

Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера укажем один час:

```
>>> datetime.timedelta(0, 0, 0, 0, 0, 1)
datetime.timedelta(0, 3600)
>>> datetime.timedelta(hours=1)
datetime.timedelta(0, 3600)
```

Получить результат можно с помощью следующих свойств:

- `days` — дни;
- `seconds` — секунды;
- `microseconds` — микросекунды.

Пример:

```
>>> d = datetime.timedelta(hours=1, days=2, milliseconds=1)
>>> d
datetime.timedelta(2, 3600, 1000)
>>> d.days, d.seconds, d.microseconds
(2, 3600, 1000)
>>> repr(d), str(d)
('datetime.timedelta(2, 3600, 1000)', '2 days, 1:00:00.001000')
```

Над экземплярами класса `timedelta` можно производить арифметические операции `+`, `-`, `//` и `*`, использовать унарные операторы `+` и `-`, а также получать абсолютное значение с помощью функции `abs()`. Примеры:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d1 + d2, d2 - d1                          # Сложение и вычитание
(datetime.timedelta(9), datetime.timedelta(5))
>>> d1 // 2, d2 // 2                          # Деление
```

```
(datetime.timedelta(1), datetime.timedelta(3, 43200))
>>> d1 * 2, d2 * 2                          # Умножение
(datetime.timedelta(4), datetime.timedelta(14))
>>> 2 * d1, 2 * d2                          # Умножение
(datetime.timedelta(4), datetime.timedelta(14))
>>> d3 = -d1
>>> d3, abs(d3)
(datetime.timedelta(-2), datetime.timedelta(2))
```

Кроме того, можно использовать операторы сравнения `==`, `!=`, `<`, `<=`, `>` и `>=`. Пример:

```
>>> d1 = datetime.timedelta(days=2)
>>> d2 = datetime.timedelta(days=7)
>>> d3 = datetime.timedelta(weeks=1)
>>> d1 == d2, d2 == d3                # Проверка на равенство
(False, True)
>>> d1 != d2, d2 != d3                # Проверка на неравенство
(True, False)
>>> d1 < d2, d2 <= d3                 # Меньше, меньше или равно
(True, True)
>>> d1 > d2, d2 >= d3                 # Больше, больше или равно
(False, True)
```

10.4.2. Класс *date*

Класс `date` из модуля `datetime` позволяет производить операции над датами. Конструктор класса имеет следующий формат:

```
date(<Год>, <Месяц>, <День>)
```

Все параметры являются обязательными. В параметрах можно указать следующий диапазон значений:

- `<Год>` — от `MINYEAR` до `MAXYEAR`. Выведем значения этих констант:

  ```
  >>> import datetime
  >>> datetime.MINYEAR, datetime.MAXYEAR
  (1, 9999)
  ```

- `<Месяц>` — от `1` до `12` включительно;
- `<День>` — от `1` до количества дней в месяце.

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> datetime.date(2010, 1, 19)
datetime.date(2010, 1, 19)
>>> datetime.date(2010, 13, 19)   # Неправильное значение для месяца
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.date(2010, 6, 5)
>>> repr(d), str(d)
```

```
('datetime.date(2010, 6, 5)', '2010-06-05')
```

Для создания экземпляра класса можно также воспользоваться следующими методами:

- `today()` — возвращает текущую дату:

```
>>> datetime.date.today() # Получаем текущую дату
datetime.date(2010, 6, 5)
```

- `fromtimestamp(<Количество секунд>)` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи:

```
>>> import datetime, time
>>> datetime.date.fromtimestamp(time.time()) # Текущая дата
datetime.date(2010, 6, 5)
>>> datetime.date.fromtimestamp(1233368623.0) # Дата 31-01-2009
datetime.date(2009, 1, 31)
```

- `fromordinal(<Количество дней с 1 года>)` — возвращает дату, соответствующую количеству дней, прошедших с `1` года. В качестве параметра указывается число от `1` до `datetime.date.max.toordinal()`. Пример:

```
>>> datetime.date.max.toordinal()
3652059
>>> datetime.date.fromordinal(3652059)
datetime.date(9999, 12, 31)
>>> datetime.date.fromordinal(1)
datetime.date(1, 1, 1)
```

Получить результат можно с помощью следующих свойств:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от `1` до `12`);
- `day` — день (число от `1` до количества дней в месяце).

Пример:

```
>>> d = datetime.date.today() # Текущая дата (05-06-2010)
>>> d.year, d.month, d.day
(2010, 6, 5)
```

Над экземплярами класса `date` можно производить следующие операции:

- `date2 = date1 + timedelta` — прибавляет к дате указанный период в днях. Значения свойств `timedelta.seconds` и `timedelta.microseconds` игнорируются;
- `date2 = date1 - timedelta` — вычитает из даты указанный период в днях. Значения свойств `timedelta.seconds` и `timedelta.microseconds` игнорируются;
- `timedelta = date1 - date2` — возвращает разницу между датами (период в днях). Свойства `timedelta.seconds` и `timedelta.microseconds` будут иметь значение `0`;
- можно также сравнивать две даты с помощью операторов сравнения.

Примеры:

```
>>> d1 = datetime.date(2010, 1, 19)
>>> d2 = datetime.date(2010, 1, 1)
>>> t = datetime.timedelta(days=10)
```

```
>>> d1 + t, d1 - t              # Прибавляем и вычитаем 10 дней
(datetime.date(2010, 1, 29), datetime.date(2010, 1, 9))
>>> d1 - d2                     # Разница между датами
datetime.timedelta(18)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
(False, True, False, True)
>>> d1 == d2, d1 != d2
(False, True)
```

Экземпляры класса `date` поддерживают следующие методы:

- `replace([year[, month[, day]]])` — возвращает дату с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. Примеры:

```
>>> d = datetime.date(2010, 6, 5)
>>> d.replace(2012, 3) # Заменяем год и месяц
datetime.date(2012, 3, 5)
>>> d.replace(year=2009, month=3, day=1)
datetime.date(2009, 3, 1)
>>> d.replace(day=7)   # Заменяем только день
datetime.date(2010, 6, 7)
```

- `strftime(<Строка формата>)` — возвращает отформатированную строку. В строке формата можно задавать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`. Пример:

```
>>> d = datetime.date(2010, 6, 5)
>>> d.strftime("%d.%m.%Y")
'05.06.2010'
```

- `isoformat()` — возвращает дату в формате `ГГГГ-ММ-ДД`:

```
>>> d = datetime.date(2010, 6, 5)
>>> d.isoformat()
'2010-06-05'
```

- `ctime()` — возвращает строку специального формата:

```
>>> d = datetime.date(2010, 6, 5)
>>> d.ctime()
'Sat Jun  5 00:00:00 2010'
```

- `timetuple()` — возвращает объект `struct_time` с датой и временем:

```
>>> d = datetime.date(2010, 6, 5)
>>> d.timetuple()
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=5, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=5, tm_yday=156, tm_isdst=-1)
```

- `toordinal()` — возвращает количество дней, прошедших с 1 года:

```
>>> d = datetime.date(2010, 6, 5)
>>> d.toordinal()
733928
>>> datetime.date.fromordinal(733928)
```

```
datetime.date(2010, 6, 5)
```

- `weekday()` — возвращает порядковый номер дня в неделе (`0` — для понедельника, `6` — для воскресенья):

```
>>> d = datetime.date(2010, 6, 5)
>>> d.weekday() # 5 - это суббота
5
```

- `isoweekday()` — возвращает порядковый номер дня в неделе (`1` — для понедельника, `7` — для воскресенья):

```
>>> d = datetime.date(2010, 6, 5)
>>> d.isoweekday() # 6 - это суббота
6
```

- `isocalendar()` — возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе):

```
>>> d = datetime.date(2010, 6, 5)
>>> d.isocalendar()
(2010, 22, 6)
```

10.4.3. Класс *time*

Класс `time` из модуля `datetime` позволяет производить операции над временем. Конструктор класса имеет следующий формат:

```
time([hour[, minute[, second[, microsecond[, tzinfo]]]]])
```

Все параметры являются необязательными. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В параметрах можно указать следующий диапазон значений:

- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> import datetime
>>> datetime.time(23, 12, 38, 375000)
datetime.time(23, 12, 38, 375000)
>>> t = datetime.time(hour=23, second=38, minute=12)
>>> repr(t), str(t)
('datetime.time(23, 12, 38)', '23:12:38')
>>> datetime.time(25, 12, 38, 375000)
... Фрагмент опущен ...
ValueError: hour must be in 0..23
```

Получить результат можно с помощью следующих свойств:

- `hour` — часы (число от `0` до `23`);

- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Пример:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.hour, t.minute, t.second, t.microsecond
(23, 12, 38, 375000)
```

Над экземплярами класса `time` нельзя производить арифметические операции. Можно только производить сравнения. Пример:

```
>>> t1 = datetime.time(23, 12, 38, 375000)
>>> t2 = datetime.time(12, 28, 17)
>>> t1 < t2, t1 > t2, t1 <= t2, t1 >= t2
(False, True, False, True)
>>> t1 == t2, t1 != t2
(False, True)
```

Экземпляры класса `time` поддерживают следующие методы:

- `replace([hour[, minute[, second[, microsecond[, tzinfo]]]]])` — возвращает время с обновленными значениями. Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. Пример:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.replace(10, 52)       # Заменяем часы и минуты
datetime.time(10, 52, 38, 375000)
>>> t.replace(second=21)    # Заменяем только секунды
datetime.time(23, 12, 21, 375000)
```

- `isoformat()` — возвращает время в формате ISO 8601:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.isoformat()
'23:12:38.375000'
```

- `strftime(<Строка формата>)` — возвращает отформатированную строку. В строке формата можно указывать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`. Пример:

```
>>> t = datetime.time(23, 12, 38, 375000)
>>> t.strftime("%H:%M:%S")
'23:12:38'
```

Примечание

Экземпляры класса `time` поддерживают также методы `dst()`, `utcoffset()` и `tzname()`. За подробной информацией по этим методам, а также по абстрактному классу `tzinfo` обращайтесь к документации по модулю `datetime`.

10.4.4. Класс *datetime*

Класс `datetime` из модуля `datetime` позволяет производить операции над комбинацией даты и времени. Конструктор класса имеет следующий формат:

```
datetime(<Год>, <Месяц>, <День>[, hour[, minute[, second
        [, microsecond[, tzinfo]]]]])
```

Первые три параметра являются обязательными. Остальные значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В параметрах можно указать следующий диапазон значений:

- `<Год>` — число от `MINYEAR` (`1`) до `MAXYEAR` (`9999`);
- `<Месяц>` — число от `1` до `12` включительно;
- `<День>` — число от `1` до количества дней в месяце;
- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Если значения выходят за диапазон, возбуждается исключение `ValueError`. Пример:

```
>>> import datetime
>>> datetime.datetime(2010, 6, 5)
datetime.datetime(2010, 6, 5, 0, 0)
>>> datetime.datetime(2010, 6, 5, hour=12, minute=55)
datetime.datetime(2010, 6, 5, 12, 55)
>>> datetime.datetime(2010, 32, 20)
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.datetime(2010, 6, 5, 5, 19, 21)
>>> repr(d), str(d)
('datetime.datetime(2010, 6, 5, 5, 19, 21)', '2010-06-05 05:19:21')
```

Для создания экземпляра класса можно также воспользоваться следующими методами:

- `today()` — возвращает текущую дату и время:

  ```
  >>> datetime.datetime.today()
  datetime.datetime(2010, 6, 5, 23, 52, 58, 531000)
  ```

- `now([<Зона>])` — возвращает текущую дату и время. Если параметр не задан, то метод аналогичен методу `today()`. Пример:

  ```
  >>> datetime.datetime.now()
  datetime.datetime(2010, 6, 5, 23, 53, 20, 250000)
  ```

- `utcnow()` — возвращает текущее универсальное время (UTC):

  ```
  >>> datetime.datetime.utcnow()
  datetime.datetime(2010, 6, 5, 19, 53, 41, 46000)
  ```

- `fromtimestamp(<Количество секунд>[, <Зона>])` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи:

```
>>> import datetime, time
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2010, 6, 5, 23, 54, 21, 265000)
>>> datetime.datetime.fromtimestamp(1233368623.0)
datetime.datetime(2009, 1, 31, 5, 23, 43)
```

- `utcfromtimestamp(<Количество секунд>)` — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи, в универсальном времени (UTC). Пример:

```
>>> import datetime, time
>>> datetime.datetime.utcfromtimestamp(time.time())
datetime.datetime(2010, 6, 5, 19, 55, 12, 984000)
>>> datetime.datetime.utcfromtimestamp(1233368623.0)
datetime.datetime(2009, 1, 31, 2, 23, 43)
```

- `fromordinal(<Количество дней с 1 года>)` — возвращает дату, соответствующую количеству дней, прошедших с 1 года. В качестве параметра указывается число от `1` до `datetime.datetime.max.toordinal()`. Пример:

```
>>> datetime.datetime.max.toordinal()
3652059
>>> datetime.datetime.fromordinal(3652059)
datetime.datetime(9999, 12, 31, 0, 0)
>>> datetime.datetime.fromordinal(1)
datetime.datetime(1, 1, 1, 0, 0)
```

- `combine(<Экземпляр класса date>, <Экземпляр класса time>)` — создает экземпляр класса `datetime` в соответствии со значениями экземпляров классов `date` и `time`:

```
>>> d = datetime.date(2010, 6, 5)  # Экземпляр класса date
>>> t = datetime.time(9, 12, 35)   # Экземпляр класса time
>>> datetime.datetime.combine(d, t)
datetime.datetime(2010, 6, 5, 9, 12, 35)
```

- `strptime(<Строка с датой>, <Строка формата>)` — разбирает строку, указанную в первом параметре, в соответствии со строкой формата. Если строка не соответствует формату, возбуждается исключение `ValueError`. Пример:

```
>>> datetime.datetime.strptime("05.06.2010", "%d.%m.%Y")
datetime.datetime(2010, 6, 5, 0, 0)
>>> datetime.datetime.strptime("05.06.2010", "%d-%m-%Y")
... Фрагмент опущен ...
ValueError: time data '05.06.2010'
does not match format '%d-%m-%Y'
```

Получить результат можно с помощью следующих свойств:

- `year` — год (число в диапазоне от `MINYEAR` до `MAXYEAR`);
- `month` — месяц (число от `1` до `12`);

- `day` — день (число от `1` до количества дней в месяце);
- `hour` — часы (число от `0` до `23`);
- `minute` — минуты (число от `0` до `59`);
- `second` — секунды (число от `0` до `59`);
- `microsecond` — микросекунды (число от `0` до `999999`);
- `tzinfo` — зона (экземпляр класса `tzinfo` или значение `None`).

Пример:

```
>>> d = datetime.datetime(2010, 6, 5, 5, 19, 21)
>>> d.year, d.month, d.day
(2010, 6, 5)
>>> d.hour, d.minute, d.second, d.microsecond
(5, 19, 21, 0)
```

Над экземплярами класса `datetime` можно производить следующие операции:

- `datetime2 = datetime1 + timedelta` — прибавляет к дате указанный период;
- `datetime2 = datetime1 - timedelta` — вычитает из даты указанный период;
- `timedelta = datetime1 - datetime2` — возвращает разницу между датами;
- можно также сравнивать две даты с помощью операторов сравнения.

Примеры:

```
>>> d1 = datetime.datetime(2010, 1, 20, 23, 48, 23)
>>> d2 = datetime.datetime(2010, 1, 1, 10, 15, 38)
>>> t = datetime.timedelta(days=10, minutes=10)
>>> d1 + t                      # Прибавляем 10 дней и 10 минут
datetime.datetime(2010, 1, 30, 23, 58, 23)
>>> d1 - t                      # Вычитаем 10 дней и 10 минут
datetime.datetime(2010, 1, 10, 23, 38, 23)
>>> d1 - d2                     # Разница между датами
datetime.timedelta(19, 48765)
>>> d1 < d2, d1 > d2, d1 <= d2, d1 >= d2
(False, True, False, True)
>>> d1 == d2, d1 != d2
(False, True)
```

Экземпляры класса `datetime` поддерживают следующие методы:

- `date()` — возвращает экземпляр класса `date`:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.date()
datetime.date(2010, 6, 5)
```

- `time()` — возвращает экземпляр класса `time`:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.time()
datetime.time(23, 48, 23)
```

- `timetz()` — возвращает экземпляр класса `time`. Метод учитывает параметр `tzinfo`;
- `replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]])` — возвращает дату с обновленными значениями. Значения

можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. Пример:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.replace(2008, 12)
datetime.datetime(2008, 12, 5, 23, 48, 23)
>>> d.replace(hour=12, month=10)
datetime.datetime(2010, 10, 5, 12, 48, 23)
```

- `timetuple()` — возвращает объект `struct_time` с датой и временем:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.timetuple()
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=5, tm_hour=23,
tm_min=48, tm_sec=23, tm_wday=5, tm_yday=156, tm_isdst=-1)
```

- `utctimetuple()` — возвращает объект `struct_time` с датой в универсальном времени (UTC):

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.utctimetuple()
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=5, tm_hour=23,
tm_min=48, tm_sec=23, tm_wday=5, tm_yday=156, tm_isdst=0)
```

- `toordinal()` — возвращает количество дней, прошедшее с 1 года:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.toordinal()
733928
```

- `weekday()` — возвращает порядковый номер дня в неделе (`0` — для понедельника, `6` — для воскресенья):

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.weekday() # 5 - это суббота
5
```

- `isoweekday()` — возвращает порядковый номер дня в неделе (`1` — для понедельника, `7` — для воскресенья):

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.isoweekday() # 6 - это суббота
6
```

- `isocalendar()` — возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе):

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.isocalendar()
(2010, 22, 6)
```

- `isoformat([<Разделитель>])` — возвращает дату в формате ISO 8601:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.isoformat()           # Разделитель не указан
'2010-06-05T23:48:23'
>>> d.isoformat(" ")        # Пробел в качестве разделителя
'2010-06-05 23:48:23'
```

- `ctime()` — возвращает строку специального формата:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.ctime()
'Sat Jun  5 23:48:23 2010'
```

- `strftime(<Строка формата>)` — возвращает отформатированную строку. В строке формата можно указывать комбинации специальных символов, которые используются в функции `strftime()` из модуля `time`. Пример:

```
>>> d = datetime.datetime(2010, 6, 5, 23, 48, 23)
>>> d.strftime("%d.%m.%Y %H:%M:%S")
'05.06.2010 23:48:23'
```

Примечание

Экземпляры класса `datetime` поддерживают также методы `astimezone()`, `dst()`, `utcoffset()` и `tzname()`. За подробной информацией по этим методам, а также по абстрактному классу `tzinfo` обращайтесь к документации по модулю `datetime`.

10.5. Модуль *calendar*. Вывод календаря

Модуль `calendar` позволяет вывести календарь в виде простого текста или в HTML-формате. Прежде чем использовать модуль, необходимо подключить его с помощью выражения:

```
import calendar
```

Модуль предоставляет следующие классы:

- `Calendar` — базовый класс, который наследуют все остальные классы. Формат конструктора:

```
Calendar([<Первый день недели>])
```

В качестве примера получим двумерный список всех дней в январе 2010 года, распределенных по дням недели:

```
>>> import calendar
>>> c = calendar.Calendar(0)
>>> c.monthdayscalendar(2010, 1) # 1 — это январь
[[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10],
[11, 12, 13, 14, 15, 16, 17], [18, 19, 20, 21, 22, 23, 24],
[25, 26, 27, 28, 29, 30, 31]]
```

- `TextCalendar` — позволяет вывести календарь в виде простого текста. Формат конструктора:

```
TextCalendar([<Первый день недели>])
```

Выведем календарь на весь 2010 год:

```
>>> c = calendar.TextCalendar(0)
>>> print c.formatyear(2010) # Текстовый календарь на 2010 год
```

- `LocaleTextCalendar` — позволяет вывести календарь в виде простого текста. Названия месяцев и дней недели выводятся в соответствии с указанной локалью. Формат конструктора:

  ```
  LocaleTextCalendar([<Первый день недели>[, <Название локали>]])
  ```

  Выведем календарь на весь 2010 год на русском языке:

  ```
  >>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
  >>> print c.formatyear(2010)
  ```

- `HTMLCalendar` — позволяет вывести календарь в формате HTML. Формат конструктора:

  ```
  HTMLCalendar([<Первый день недели>])
  ```

  Выведем календарь на весь 2010 год:

  ```
  >>> c = calendar.HTMLCalendar(0)
  >>> print c.formatyear(2010)
  ```

- `LocaleHTMLCalendar` — позволяет вывести календарь в формате HTML. Названия месяцев и дней недели выводятся в соответствии с указанной локалью. Формат конструктора:

  ```
  LocaleHTMLCalendar([<Первый день недели>[, <Название локали>]])
  ```

  Выведем календарь на весь 2010 год на русском языке в виде отдельной XHTML-страницы:

  ```
  >>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
  >>> print c.formatyearpage(2010, encoding="windows-1251")
  ```

В первом параметре всех конструкторов указывается число от `0` (для понедельника) до `6` (для воскресенья). Если параметр не указан, то значение равно `0`. Вместо чисел можно использовать встроенные константы `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`. Изменить значение параметра позволяет метод `setfirstweekday(<Первый день недели>)`. В качестве примера выведем текстовый календарь на январь 2010 года, где первым днем недели является воскресенье (листинг 10.3).

Листинг 10.3. Вывод текстового календаря

```
>>> c = calendar.TextCalendar()        # Первый день понедельник
>>> c.setfirstweekday(calendar.SUNDAY) # Первый день теперь воскресенье
>>> print c.formatmonth(2010, 1) # Текстовый календарь на январь 2010 г.
```

10.5.1. Методы классов *TextCalendar* и *LocaleTextCalendar*

Экземпляры классов `TextCalendar` и `LocaleTextCalendar` имеют следующие методы:

- `formatmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов переноса строки>]])` — возвращает текстовый календарь на указанный

месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов переноса строки между строками. Выведем календарь на январь 2010 года:

```
>>> import calendar
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print c.formatmonth(2010, 1)
    Январь 2010
Пн Вт Ср Чт Пт Сб Вс
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

- `prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов переноса строки>]])` — метод аналогичен методу `formatmonth()`, но не возвращает календарь в виде строки, а сразу выводит его. Выведем календарь на январь 2010 года и укажем ширину поля с днем равной 4 символам:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.prmonth(2010, 1, 4)
           Январь 2010
 Пн   Вт   Ср   Чт   Пт   Сб   Вс
                    1    2    3
  4    5    6    7    8    9   10
 11   12   13   14   15   16   17
 18   19   20   21   22   23   24
 25   26   27   28   29   30   31
```

- `formatyear(<Год>[, w[, l[, c[, m]]]])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
  - `w` — задает ширину поля с днем (по умолчанию `2`);
  - `l` — количество символов переноса строки между строками (по умолчанию `1`);
  - `c` — количество пробелов между месяцами (по умолчанию `6`);
  - `m` — количество месяцев на строке (по умолчанию `3`).

  Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера выведем календарь на 2010 год. На одной строке выведем сразу четыре месяца и установим количество пробелов между месяцами:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print c.formatyear(2010, m=4, c=2)
```

- `pryear(<Год>[, w[, l[, c[, m]]]])` — метод аналогичен методу `formatyear()`, но не возвращает календарь в виде строки, а сразу выводит его. В качестве примера выведем календарь на 2011 год по два месяца на строке.

Расстояние между месяцами установим равным 4 символам, ширину поля с датой равной 2 символам, а строки разделим одним символом переноса строки:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.pryear(2011, 2, 1, 4, 2)
```

10.5.2. Методы классов *HTMLCalendar* и *LocaleHTMLCalendar*

Экземпляры классов `HTMLCalendar` и `LocaleHTMLCalendar` имеют следующие методы:

- `formatmonth(<Год>, <Месяц>[, <True | False>])` — возвращает HTML-календарь на указанный месяц в году. Если в третьем параметре указано значение `True` (значение по умолчанию), то в заголовке таблицы после названия месяца будет указан год. Календарь будет отформатирован с помощью HTML-таблицы. Для каждой ячейки таблицы задается стилевой класс, с помощью которого можно управлять внешним видом календаря. Названия стилевых классов доступны через свойство `cssclasses`, которое содержит список названий для каждого дня недели:

```
>>> import calendar
>>> c = calendar.HTMLCalendar(0)
>>> print c.cssclasses
['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun']
```

Выведем календарь на январь 2010 года. Для будних дней укажем класс `"workday"`, а для выходных дней — класс `"week-end"`:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> c.cssclasses = ["workday", "workday", "workday", "workday",
                    "workday", "week-end", "week-end"]
>>> print c.formatmonth(2010, 1, False)
```

- `formatyear(<Год>[, <Количество месяцев на строке>])` — возвращает HTML-календарь на указанный год. Календарь будет отформатирован с помощью нескольких HTML-таблиц. В качестве примера выведем календарь на 2010 год. На одной строке выведем сразу четыре месяца:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> print c.formatyear(2010, 4)
```

- `formatyearpage(<Год>[, width[, css[, encoding]]])` — возвращает HTML-календарь на указанный год в виде отдельной XHTML-страницы. Параметры имеют следующее предназначение:
  - `width` — количество месяцев на строке (по умолчанию `3`);
  - `css` — название файла с таблицей стилей (по умолчанию `"calendar.css"`);
  - `encoding` — кодировка файла. Название кодировки будет указано в параметре `encoding` XML-пролога, а также в теге `<meta>`.

Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера выведем календарь на 2010 год. На одной строке выведем сразу четыре месяца и укажем кодировку файла:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> print c.formatyearpage(2010, 4, encoding="windows-1251")
```

10.5.3. Другие полезные функции

Модуль `calendar` предоставляет также несколько функций, которые позволяют вывести текстовый календарь без создания экземпляра класса, а также возвращают дополнительную информацию о дате:

- `setfirstweekday(<Первый день недели>)` — устанавливает первый день недели для календаря. В качестве параметра указывается число от `0` (для понедельника) до `6` (для воскресенья). Вместо чисел можно использовать встроенные константы `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` или `SUNDAY`. Получить текущее значение параметра можно с помощью функции `firstweekday()`. Установим воскресенье первым днем недели:

```
>>> import calendar
>>> calendar.firstweekday()     # По умолчанию 0
0
>>> calendar.setfirstweekday(6) # Изменяем значение
>>> calendar.firstweekday()     # Проверяем установку
6
```

- `month(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов переноса строки>]])` — возвращает текстовый календарь на указанный месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов переноса строки между строками. Выведем календарь на январь 2010 года:

```
>>> calendar.setfirstweekday(0)
>>> print calendar.month(2010, 1) # Январь 2010 года
    January 2010
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

- `prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов переноса строки>]])` — функция аналогична функции `month()`, но не возвращает календарь в виде строки, а сразу выводит его. Выведем календарь на январь 2010 года:

```
>>> calendar.prmonth(2010, 1) # Январь 2010 года
```

- `monthcalendar(<Год>, <Месяц>)` — возвращает двумерный список всех дней в указанном месяце, распределенных по дням недели. Дни, выходящие за пределы месяца, будут представлены нулями. Выведем массив для января 2010 года:

```
>>> calendar.monthcalendar(2010, 1) # Январь 2010 года
[[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10],
 [11, 12, 13, 14, 15, 16, 17], [18, 19, 20, 21, 22, 23, 24],
 [25, 26, 27, 28, 29, 30, 31]]
```

- `monthrange(<Год>, <Месяц>)` — возвращает кортеж из двух элементов: количество недель в месяце и число дней в месяце:

```
>>> print calendar.monthrange(2010, 1)
(4, 31)
>>> # В январе 2010 года 4 недели и 31 день
```

- `calendar(<Год>[, w[, l[, c[, m]]]])` — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
  - `w` — задает ширину поля с днем (по умолчанию `2`);
  - `l` — количество символов переноса строки между строками (по умолчанию `1`);
  - `c` — количество пробелов между месяцами (по умолчанию `6`);
  - `m` — количество месяцев на строке (по умолчанию `3`).

  Значения можно указывать через запятую в порядке следования параметров или присвоить значение названию параметра. В качестве примера выведем календарь на 2010 год. На одной строке выведем сразу четыре месяца и установим количество пробелов между месяцами:

```
>>> print calendar.calendar(2010, m=4, c=2)
```

- `prcal(<Год>[, w[, l[, c[, m]]]])` — функция аналогична функции `calendar()`, но не возвращает календарь в виде строки, а сразу выводит его. В качестве примера выведем календарь на 2011 год по два месяца на строке. Расстояние между месяцами установим равным 4 символам, ширину поля с датой равной 2 символам, а строки разделим одним символом переноса строки:

```
>>> calendar.prcal(2011, 2, 1, 4, 2)
```

- `isleap(<Год>)` — возвращает значение `True`, если указанный год является високосным, в противном случае — `False`:

```
>>> calendar.isleap(2010), calendar.isleap(2012)
(False, True)
```

- `leapdays(<Год1>, <Год2>)` — возвращает количество високосных лет в диапазоне от `<Год1>` до `<Год2>` (`<Год2>` не учитывается):

```
>>> calendar.leapdays(2010, 2012) # 2012 не учитывается
0
>>> calendar.leapdays(2010, 2013) # 2012 - високосный год
1
```

- `weekday(<Год>, <Месяц>, <День>)` — возвращает номер дня недели (`0` — для понедельника, `6` — для воскресенья):

```
>>> calendar.weekday(2010, 6, 4) # 4 — это пятница
4
```

- `timegm(<Объект struct_time>)` — возвращает число, представляющее количество секунд, прошедших с начала эпохи. В качестве параметра указывается объект `struct_time` с датой и временем, возвращаемый функцией `gmtime()` из модуля `time`. Пример:

```
>>> import calendar, time
>>> d = time.gmtime(1275762391.0) # Дата 05-06-2010
>>> d
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=5, tm_hour=18,
tm_min=26, tm_sec=31, tm_wday=5, tm_yday=156, tm_isdst=0)
>>> tuple(d)
(2010, 6, 5, 18, 26, 31, 5, 156, 0)
>>> calendar.timegm(d)
1275762391
>>> calendar.timegm((2010, 6, 5, 18, 26, 31, 5, 156, 0))
1275762391
```

Модуль `calendar` предоставляет также несколько свойств:

- `day_name` — полные названия дней недели в текущей локали:

```
>>> [i for i in calendar.day_name]
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
```

- `day_abbr` — аббревиатуры названий дней недели в текущей локали:

```
>>> [i for i in calendar.day_abbr]
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

- `month_name` — полные названия месяцев в текущей локали:

```
>>> [i for i in calendar.month_name]
['', 'January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December']
```

- `month_abbr` — аббревиатуры названий месяцев в текущей локали:

```
>>> [i for i in calendar.month_abbr]
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec']
>>> import locale # Настройка локали
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> for i in calendar.month_abbr: print i,
янв фев мар апр май июн июл авг сен окт ноя дек
```

10.6. Измерение времени выполнения фрагментов кода

Модуль `timeit` позволяет измерить время выполнения небольших фрагментов кода с целью оптимизации программы. Прежде чем использовать модуль, необходимо подключить его с помощью выражения:

```
from timeit import Timer
```

Измерения производятся с помощью класса `Timer`. Конструктор класса имеет следующий формат:

```
Timer([stmt='pass'[, setup='pass'[, timer=<timer function>]]])
```

В параметре `stmt` указывается код (в виде строки), для которого измеряем время выполнения. Параметр `setup` позволяет указать код, который будет выполнен перед измерением времени выполнения кода в параметре `stmt`. Например, в параметре `setup` можно подключить модуль.

Получить время выполнения можно с помощью метода `timeit([number=1000000])`. В параметре `number` указывается количество повторений. Для примера просуммируем числа от `1` до `10001` тремя способами и выведем время выполнения каждого способа (листинг 10.4).

Листинг 10.4. Измерение времени выполнения

```
# -*- coding: cp1251 -*-
from timeit import Timer
code1 = """\
i, j = 1, 0
while i < 10001:
    j += i
    i += 1
"""
t1 = Timer(stmt=code1)
print "while:", t1.timeit(number=10000)
code2 = """\
j = 0
for i in xrange(1, 10001):
    j += i
"""
t2 = Timer(stmt=code2)
print "for:", t2.timeit(number=10000)
code3 = """\
j = sum(xrange(1, 10001))
"""
t3 = Timer(stmt=code3)
print "sum:", t3.timeit(number=10000)
```

Результат выполнения:

```
while: 11.9412368511
for: 4.86864243359
sum: 1.23282160686
```

Как видно из результата, цикл `for` работает в два раза быстрее цикла `while`, а функция `sum()` в данном случае является самым оптимальным решением задачи.

Метод `repeat([repeat=3[, number=1000000]])` вызывает метод `timeit()` указанное количество раз (задается в параметре `repeat`) и возвращает список значений. Аргумент `number` передается в качестве параметра методу `timeit()`. Для примера создадим список со строковыми представлениями чисел от `1` до `10000`. В первом случае для создания списка используем цикл `for` и метод `append()`, а во втором — генератор списков (листинг 10.5).

Листинг 10.5. Использование метода `repeat()`

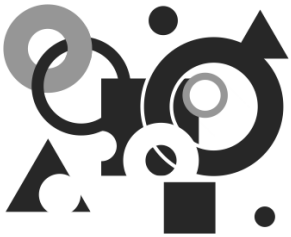
```
# -*- coding: cp1251 -*-
from timeit import Timer
code1 = """\
arr1 = []
for i in xrange(1, 10001):
    arr1.append(str(i))
"""
t1 = Timer(stmt=code1)
print "append:", t1.repeat(repeat=3, number=2000)
code2 = """\
arr2 = [str(i) for i in xrange(1, 10001)]
"""
t2 = Timer(stmt=code2)
print "генератор:", t2.repeat(repeat=3, number=2000)
```

Результат выполнения:

```
append: [7.6997649829907004, 7.6997334811098881, 7.7073105405090701]
генератор: [6.3373980661542646, 6.3322780897085522, 6.3315509547333022]
```

Как видно из результата, генераторы списков работают быстрее.

Глава 11



Пользовательские функции

Функция — это фрагмент кода, который можно вызвать из любого места программы. В предыдущих главах мы уже не один раз использовали встроенные функции языка Python, например, с помощью функции `len()` получали количество элементов последовательности. В этой главе мы рассмотрим создание пользовательских функций, которые позволят уменьшить избыточность программного кода и повысить его структурированность.

11.1. Создание функции и ее вызов

Функция описывается с помощью ключевого слова `def` по следующей схеме:

```
def <Имя функции>([<Параметры>]):
    [""" Строка документирования """]
    <Тело функции>
    [return <Значение>]
```

Имя функции должно быть уникальным идентификатором, состоящим из латинских букв, цифр и знаков подчеркивания, причем имя функции не может начинаться с цифры. В качестве имени нельзя использовать ключевые слова; кроме того, следует избегать совпадений с названиями встроенных идентификаторов. Регистр символов в названии функции имеет значение. Для наглядности все названия пользовательских функций в этой книге будут начинаться с префикса `"f_"`. Это позволит вам отличать имена встроенных функций от названий, определенных нами в программе.

После имени функции в круглых скобках можно указать один или несколько параметров через запятую. Если функция не принимает параметры, то просто указываются круглые скобки. После круглых скобок ставится двоеточие.

Тело функции является составной конструкцией. Как и в любой составной конструкции, выражения внутри функции выделяются одинаковым количеством пробелов слева. Концом функции считается выражение, перед которым меньшее коли-

чество пробелов. Если тело функции не содержит выражений, то внутри необходимо разместить оператор `pass`. Этот оператор удобно использовать на этапе отладки программы, когда мы определили функцию, а тело будем дописывать позже. Пример функции, которая ничего не делает:

```
def f_pass():
    pass
```

Необязательная инструкция `return` позволяет вернуть значение из функции. После исполнения этой инструкции выполнение функции будет остановлено. Это означает, что выражения после оператора `return` никогда не будут выполнены. Пример:

```
def f_test():
    print "Текст до инструкции return"
    return "Возвращаемое значение"
    print "Это выражение никогда не будет выполнено"
print f_test() # Вызываем функцию
```

Результат выполнения:

```
Текст до инструкции return
Возвращаемое значение
```

Инструкции `return` может не быть вообще. В этом случае выполняются все выражения внутри функции и возвращается значение `None`.

В качестве примера создадим три функции (листинг 11.1).

Листинг 11.1. Определения функций

```
def f_print_ok():
    """ Пример функции без параметров """
    print "Сообщение при удачно выполненной операции"
def f_print(m):
    """ Пример функции с параметром """
    print m
def f_sum(x, y):
    """ Пример функции с параметрами,
        возвращающей сумму двух переменных """
    return x + y
```

Вызвать эти функции можно способами, указанными в листинге 11.2.

Листинг 11.2. Вызов функций

```
f_print_ok()              # Вызываем функцию без параметров
f_print("Сообщение")      # Функция выведет сообщение
v1 = f_sum(5, 2)          # Переменной v1 будет присвоено значение 7
a, b = 10, 50
v2 = f_sum(a, b)          # Переменной v2 будет присвоено значение 60
```

Как видно из последнего примера, имя переменной в вызове функции может не совпадать с именем переменной в определении функции. Необходимо также заметить, что количество параметров в определении функции должно совпадать с количеством параметров при вызове, иначе будет выведено сообщение об ошибке.

Оператор `+`, используемый в функции `f_sum()`, применяется не только для сложения чисел, но и позволяет объединить последовательности. Таким образом, функция `f_sum()` может использоваться не только для сложения чисел. В качестве примера выполним конкатенацию строк и объединение списков:

```
def f_sum(x, y):
    return x + y
print f_sum("str", "ing")   # Выведет: string
print f_sum([1, 2], [3, 4]) # Выведет: [1, 2, 3, 4]
```

Как вы уже знаете, все в языке Python является объектом, например, строки, списки и даже сами типы данных. Функции не являются исключением. Инструкция `def` создает объект, имеющий тип `function`, и сохраняет ссылку на него в идентификаторе, указанном после инструкции `def`. Таким образом, мы можем сохранить ссылку на функцию в другой переменной. Для этого название функции указывается без круглых скобок. Сохраним ссылку в переменной и вызовем функцию через нее (листинг 11.3).

Листинг 11.3. Сохранение ссылки на функцию в переменной

```
def f_sum(x, y):
    return x + y
f = f_sum                    # Сохраняем ссылку в переменной f
v = f(10, 20)                # Вызываем функцию через переменную f
```

Можно также передать ссылку на функцию в качестве параметра другой функции. Функции, передаваемые по ссылке, обычно называются *функциями обратного вызова* (листинг 11.4).

Листинг 11.4. Функции обратного вызова

```
def f_sum(x, y):
    return x + y
def f_sum2(f, a, b):
    """ Через переменную f будет доступна ссылка на
        функцию f_sum() """
    return f(a, b) # Вызываем функцию f_sum()
# Передаем ссылку на функцию в качестве параметра
v = f_sum2(f_sum, 10, 20)
```

11.2. Расположение определений функций

Все выражения в программе выполняются последовательно сверху вниз. Это означает, что прежде чем использовать идентификатор в программе, его необходимо предварительно объявить, присвоив ему значение. Поэтому определение функции должно быть расположено перед вызовом функции.

Правильно:

```
def f_sum(x, y):
    return x + y
v = f_sum(10, 20) # Вызываем после определения. Все нормально
```

Неправильно:

```
v = f_sum(10, 20) # Идентификатор еще не определен. Это ошибка!!!
def f_sum(x, y):
    return x + y
```

В последнем случае будет выведено сообщение об ошибке `"NameError: name 'f_sum' is not defined"`. Чтобы избежать ошибки, определение функции размещают в самом начале программы после подключения модулей или в отдельном файле, который называется *модулем*.

С помощью оператора ветвления `if` можно изменить порядок выполнения программы. Таким образом, можно разместить внутри условия несколько определений функций с одинаковым названием, но разной реализацией (листинг 11.5).

Листинг 11.5. Определение функции в зависимости от условия

```
# -*- coding: cp1251 -*-
m = raw_input("Введите 1 для вызова первой функции: ")
if m == "1":
    def f_print():
        return "Вы ввели число 1"
else:
    def f_print():
        return "Альтернативная функция"
print f_print() # Вызываем функцию
```

При вводе числа 1 мы получим сообщение `"Вы ввели число 1"`, в противном случае — `"Альтернативная функция"`.

Если определение одной функции встречается в программе несколько раз, то будет использоваться функция, которая расположена последней. Пример:

```
def f_print():
    return "Вы ввели число 1"
def f_print():
    return "Альтернативная функция"
print f_print() # Всегда выводит "Альтернативная функция"
```

11.3. Необязательные параметры и сопоставление по ключам

Чтобы сделать некоторые параметры необязательными, следует в определении функции присвоить этому параметру начальное значение. Переделаем функцию суммирования двух чисел и сделаем второй параметр необязательным (листинг 11.6).

Листинг 11.6. Необязательные параметры

```
def f_sum(x, y=2):          # y — необязательный параметр
    return x + y
v1 = f_sum(5)               # Переменной v1 будет присвоено значение 7
v2 = f_sum(10, 50)          # Переменной v2 будет присвоено значение 60
```

Таким образом, если второй параметр не задан, то его значение будет равно `2`. Обратите внимание на то, что необязательные параметры должны следовать после обязательных параметров, иначе будет выведено сообщение об ошибке.

До сих пор мы использовали позиционную передачу параметров в функцию:

```
def f_sum(x, y):
    return x + y
print f_sum(10, 20)         # Выведет: 30
```

Переменной `x` при сопоставлении будет присвоено значение `10`, а переменной `y` — значение `20`. Язык Python позволяет также передать значения в функцию, используя сопоставление по ключам (листинг 11.7). Для этого при вызове функции параметрам присваиваются значения. Последовательность указания параметров может быть произвольной.

Листинг 11.7. Сопоставление по ключам

```
def f_sum(x, y):
    return x + y
print f_sum(y=20, x=10)    # Сопоставление по ключам
```

Сопоставление по ключам очень удобно использовать, если функция имеет несколько необязательных параметров. В этом случае не нужно перечислять все значения, а достаточно присвоить значение нужному параметру. Пример:

```
def f_sum(a=2, b=3, c=4): # Все параметры являются необязательными
    return a + b + c
print f_sum(2, 3, 20)      # Позиционное присваивание
print f_sum(c=20)          # Сопоставление по ключам
```

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ `*`. Пример передачи значений из кортежа и списка приведен в листинге 11.8.

Листинг 11.8. Пример передачи значений из кортежа и списка

```
def f_sum(a, b, c):
    return a + b + c
t1, arr = (1, 2, 3), [1, 2, 3]
print f_sum(*t1)                    # Распаковываем кортеж
print f_sum(*arr)                   # Распаковываем список
t2 = (2, 3)
print f_sum(1, *t2)                 # Можно комбинировать значения
```

Если значения параметров содержатся в словаре, то распаковать словарь можно, указав перед ним две звездочки (`**`) (листинг 11.9).

Листинг 11.9. Пример передачи значений из словаря

```
def f_sum(a, b, c):
    return a + b + c
d1 = {"a": 1, "b": 2, "c": 3}
print f_sum(**d1)                   # Распаковываем словарь
t, d2 = (1, 2), {"c": 3}
print f_sum(*t, **d2)               # Можно комбинировать значения
```

Распаковать кортежи, списки и словари позволяет также функция `apply()`. Следует заметить, что функция `apply()` применялась в ранних версиях Python и, начиная с версии 2.3, признана устаревшей. Функция имеет следующий формат:

```
apply(<Ссылка на функцию>[, <Кортеж или список>[, <Словарь>]])
```

Пример использования функции `apply()` приведен в листинге 11.10.

Листинг 11.10. Пример использования функции `apply()`

```
def f_sum(a, b, c):
    return a + b + c
t1 = (1, 2, 3)
print apply(f_sum, t1)              # Распаковываем кортеж
d1 = {"a": 1, "b": 2, "c": 3}
print apply(f_sum, [], d1)          # Распаковываем словарь
t2, d2 = (1, 2), {"c": 3}
print apply(f_sum, t2, d2)          # Можно комбинировать значения
```

Объекты в функцию передаются по ссылке. Если объект относится к неизменяемому типу, то изменение значения внутри функции не затронет значение переменной вне функции:

```
def f_test(a, b):
    a, b = 20, "str"
x, s = 80, "test"
f_test(x, s)              # Значения переменных x и s не изменяются
print x, s                # Выведет: 80 test
```

В этом примере значения в переменных `x` и `s` не изменились. Однако, если объект относится к изменяемому типу, то ситуация будет другой:

```
def f_test(a, b):
    a[0], b["a"] = "str", 800
x = [1, 2, 3]            # Список
y = {"a": 1, "b": 2}     # Словарь
f_test(x, y)             # Значения будут изменены!!!
print x, y               # Выведет: ['str', 2, 3] {'a': 800, 'b': 2}
```

Как видно из примера, значения в переменных `x` и `y` изменились, т. к. список и словарь относятся к изменяемым типам. Чтобы избежать изменения значений, внутри функции, следует создать копию объекта:

```
def f_test(a, b):
    a = a[:]                  # Создаем поверхностную копию списка
    b = b.copy()              # Создаем поверхностную копию словаря
    a[0], b["a"] = "str", 800
x = [1, 2, 3]                 # Список
y = {"a": 1, "b": 2}          # Словарь
f_test(x, y)                  # Значения останутся прежними
print x, y                    # Выведет: [1, 2, 3] {'a': 1, 'b': 2}
```

Можно также сразу передавать копию объекта в вызове функции:

```
f_test(x[:], y.copy())
```

Если указать объект, имеющий изменяемый тип, в качестве значения по умолчанию, то этот объект будет сохраняться между вызовами функции. Пример:

```
def f_test(a=[]):
    a.append(2)
    return a
print f_test()                 # Выведет: [2]
print f_test()                 # Выведет: [2, 2]
print f_test()                 # Выведет: [2, 2, 2]
```

Как видно из примера, значения накапливаются внутри списка. Обойти эту проблему можно, например, следующим образом:

```
def f_test(a=None):
    # Создаем новый список, если значение равно None
    if a is None: a = []
    a.append(2)
    return a
```

```
print f_test()    # Выведет: [2]
print f_test()    # Выведет: [2]
print f_test([1]) # Выведет: [1, 2]
```

11.4. Переменное число параметров в функции

Если перед параметром в определении функции указать символ *, то функции можно будет передать произвольное количество параметров. Все переданные параметры сохраняются в кортеже. В качестве примера напишем функцию суммирования произвольного количества чисел (листинг 11.11).

Листинг 11.11. Сохранение переданных данных в кортеже

```
def f_sum(*t):
    """ Функция принимает произвольное количество параметров """
    res = 0
    for i in t:      # Перебираем кортеж с переданными параметрами
        res += i
    return res
print f_sum(10, 20)                    # Выведет: 30
print f_sum(10, 20, 30, 40, 50, 60)    # Выведет: 210
```

Можно также вначале указать несколько обязательных параметров и параметров, имеющих значения по умолчанию:

```
def f_sum(x, y=5, *t): # Комбинация параметров
    res = x + y
    for i in t:        # Перебираем кортеж с переданными параметрами
        res += i
    return res
print f_sum(10)                        # Выведет: 15
print f_sum(10, 20, 30, 40, 50, 60)    # Выведет: 210
```

Если перед параметром в определении функции указать две звездочки (**), то все именованные параметры будут сохранены в словаре (листинг 11.12).

Листинг 11.12. Сохранение переданных данных в словаре

```
def f_test(**d):
    for i in d:     # Перебираем словарь с переданными параметрами
        print "%s => %s" % (i, d[i]),
f_test(a=1, b=2, c=3) # Выведет: a => 1 c => 3 b => 2
```

При комбинировании параметров параметр с двумя звездочками указывается самым последним. Если в определении функции указывается комбинация параметров с одной звездочкой и двумя звездочками, то функция примет любые переданные ей параметры (листинг 11.13).

Листинг 11.13. Комбинирование параметров

```
def f_test(*t, **d):
    """ Функция примет любые параметры """
    for i in t:
        print i,
    for i in d:    # Перебираем словарь с переданными параметрами
        print "%s => %s" % (i, d[i]),
f_test(35, 10, a=1, b=2, c=3) # Выведет: 35 10 a => 1 c => 3 b => 2
f_test(10)                    # Выведет: 10
f_test(a=1, b=2)              # Выведет: a => 1 b => 2
```

11.5. Анонимные функции

Помимо обычных функций язык Python позволяет использовать анонимные функции, которые называются *лямбда-функциями*. Анонимная функция описывается с помощью ключевого слова `lambda` по следующей схеме:

```
lambda [<Параметр1>[, ..., <ПараметрN>]]: <Возвращаемое значение>
```

После ключевого слова `lambda` можно указать передаваемые параметры. В качестве параметра `<Возвращаемое значение>` указывается выражение, результат выполнения которого будет возвращен функцией. Как видно из схемы, у лямбда-функций нет имени. По этой причине их и называют анонимными функциями.

В качестве значения лямбда-функция возвращает ссылку на объект-функцию, которую можно сохранить в переменной или передать в качестве параметра в другую функцию. Вызвать лямбда-функцию можно, как и обычную, с помощью круглых скобок, внутри которых расположены передаваемые параметры. Пример использования лямбда-функций приведен в листинге 11.14.

Листинг 11.14. Пример использования лямбда-функций

```
f1 = lambda: 10 + 20              # Функция без параметров
f2 = lambda x, y: x + y           # Функция с двумя параметрами
f3 = lambda x, y, z: x + y + z    # Функция с тремя параметрами
print f1()                        # Выведет: 30
print f2(5, 10)                   # Выведет: 15
print f3(5, 10, 30)               # Выведет: 45
```

Как и в обычных функциях, некоторые параметры лямбда-функций могут быть необязательными. Для этого параметрам в определении функции присваивается значение по умолчанию (листинг 11.15).

Листинг 11.15. Необязательные параметры в лямбда-функциях

```
f = lambda x, y=2: x + y
print f(5)                          # Выведет: 7
print f(5, 6)                       # Выведет: 11
```

Наиболее часто не сохраняют ссылку в переменной, а сразу передают в качестве параметра в другую функцию. Например, метод списков `sort()` позволяет указать пользовательскую функцию сортировки. Отсортируем список без учета регистра символов, указав в качестве параметра лямбда-функцию (листинг 11.16).

Листинг 11.16. Сортировка без учета регистра символов

```
# -*- coding: cp1251 -*-
import locale                       # Настройка локали
locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
arr = ["единица1", "Единый", "Единица2"]
arr.sort(cmp=lambda a, b: cmp(a.lower(), b.lower()))
for i in arr: print i,
# Выведет: единица1 Единица2 Единый
```

11.6. Функции-генераторы

Функцией-генератором называется функция, которая может возвращать одно значение из нескольких значений на каждой итерации. Приостановить выполнение функции и превратить функцию в генератор позволяет ключевое слово `yield`. В качестве примера напишем функцию, которая возводит элементы последовательности в указанную степень (листинг 11.17).

Листинг 11.17. Пример использования функций-генераторов

```
def f_test(x, y):
    for i in xrange(1, x+1):
        yield i ** y

for n in f_test(10, 2):
    print n,             # Выведет: 1 4 9 16 25 36 49 64 81 100
print                    # Вставляем пустую строку
for n in f_test(10, 3):
    print n,             # Выведет: 1 8 27 64 125 216 343 512 729 1000
```

Функции-генераторы поддерживают метод `next()`, который позволяет получить следующее значение. Когда значения заканчиваются, метод возбуждает исключение `StopIteration`. Вызов метода `next()` в цикле `for` производится незаметно для нас. В качестве примера перепишем предыдущую программу и используем метод `next()` вместо цикла `for` (листинг 11.18).

Листинг 11.18. Использование метода `next()`

```
def f_test(x, y):
    for i in xrange(1, x+1):
       yield i ** y

r = f_test(3, 3)
print r.next()                    # Выведет: 1 (1 ** 3)
print r.next()                    # Выведет: 8 (2 ** 3)
print r.next()                    # Выведет: 27 (3 ** 3)
print r.next()                    # Исключение StopIteration
```

Таким образом, с помощью обычных функций мы можем вернуть все значения сразу в виде списка, а с помощью функций-генераторов только одно значение за раз. Эта особенность очень полезна при обработке большого количества значений.

11.7. Декораторы функций

Декораторы позволяют изменить поведение обычных функций. Например, выполнить какие-либо действия перед выполнением функции. Рассмотрим это на примере (листинг 11.19).

Листинг 11.19. Декораторы функций

```
def f_deco(f):                        # Функция-декоратор
    print "Вызвана функция f_test()"
    return f                          # Возвращаем ссылку на функцию
@f_deco
def f_test(x):
    return "x = %s" % x

print f_test(10)
```

Выведет:

```
Вызвана функция f_test()
x = 10
```

В этом примере перед определением функции `f_test()` указывается название функции `f_deco()` с предваряющим символом `@`:

```
@f_deco
```

Таким образом, функция `f_deco()` становится декоратором функции `f_test()`. В качестве параметра функция-декоратор принимает ссылку на функцию, поведение которой необходимо изменить, и должна возвращать ссылку на ту же функцию или какую-либо другую. Наш предыдущий пример эквивалентен следующему коду:

```
def f_deco(f):
    print "Вызвана функция f_test()"
    return f
def f_test(x):
    return "x = %s" % x
# Вызываем функцию f_test() через функцию f_deco()
print f_deco(f_test)(10)
```

Перед определением функции можно указать сразу несколько функций-декораторов. В качестве примера обернем функцию `f_test()` в два декоратора: `f_deco1()` и `f_deco2()` (листинг 11.20).

Листинг 11.20. Указание нескольких декораторов

```
def f_deco1(f):
    print "Вызвана функция f_deco1()"
    return f
def f_deco2(f):
    print "Вызвана функция f_deco2()"
    return f
@f_deco1
@f_deco2
def f_test(x):
    return "x = %s" % x
print f_test(10)
```

Выведет:

```
Вызвана функция f_deco2()
Вызвана функция f_deco1()
x = 10
```

Использование двух декораторов эквивалентно следующему коду:

```
f_test = f_deco1(f_deco2(f_test))
```

Сначала будет вызвана функция `f_deco2()`, а затем функция `f_deco1()`. Результат выполнения будет присвоен идентификатору `f_test`.

В качестве еще одного примера использования декораторов рассмотрим выполнение функции только при правильно введенном пароле (листинг 11.21).

Листинг 11.21. Ограничение доступа с помощью декоратора

```
passw = raw_input("Введите пароль: ")
def f_test_passw(p):
    def f_deco(f):
        if p == "10": return f
        else: return lambda: "Доступ закрыт"
    return f_deco # Возвращаем функцию-декоратор
@f_test_passw(passw)
def f_print():
    return "Доступ открыт"
print f_print() # Вызываем функцию
```

В этом примере после символа `@` указана не ссылка на функцию, а выражение, которое возвращает декоратор. Иными словами, декоратором является не функция `f_test_passw()`, а результат ее выполнения (функция `f_deco()`). Если введенный пароль является правильным, то будет выполнена функция `f_print()`, в противном случае будет выведена надпись `"Доступ закрыт"`, которую возвращает анонимная функция.

11.8. Рекурсия. Вычисление факториала

Рекурсия — это возможность функции вызывать саму себя. Рекурсию удобно использовать для перебора объекта, имеющего заранее неизвестную структуру, или выполнения неопределенного количества операций. В качестве примера рассмотрим вычисление факториала (листинг 11.22).

Листинг 11.22. Вычисление факториала

```
# -*- coding: cp1251 -*-
def factorial(n):
    if n == 0 or n == 1: return 1
    else:
        return n * factorial(n - 1)
while True:
    x = raw_input("Введите число: ")
    if x.isdigit():             # Если строка содержит только цифры
        x = int(x)              # Преобразуем строку в число
        break                   # Выходим из цикла
    else:
        print "Вы ввели не число!"
print "Факториал числа %s = %s" % (x, factorial(x))
```

Начиная с версии 2.6, для вычисления факториала можно воспользоваться функцией `factorial()` из модуля `math`. Пример:

```
>>> import math
>>> math.factorial(5), math.factorial(6)
(120, 720)
```

11.9. Глобальные и локальные переменные

Глобальные переменные — это переменные, объявленные в программе вне функции. В Python глобальные переменные видны в любой части модуля, включая функции (листинг 11.23).

Листинг 11.23. Глобальные переменные

```
def f_test(glob2):
    print "Значение глобальной переменной glob1 = %s" % glob1
    glob2 += 10
    print "Значение локальной переменной glob2 = %s" % glob2
glob1, glob2 = 10, 5
f_test(77) # Вызываем функцию
print "Значение глобальной переменной glob2 = %s" % glob2
```

Результат выполнения:

```
Значение глобальной переменной glob1 = 10
Значение локальной переменной glob2 = 87
Значение глобальной переменной glob2 = 5
```

Переменной `glob2` внутри функции присваивается значение параметра. По этой причине создается новое имя `glob2`, которое является локальным. Все изменения этой переменной внутри функции не затронут значение одноименной глобальной переменной.

Локальные переменные — это переменные, которым внутри функции присваивается значение. Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной не изменяется. Локальные переменные видны только внутри тела функции (листинг 11.24).

Листинг 11.24. Локальные переменные

```
def f_test():
    local1 = 77                    # Локальная переменная
    glob1 = 25                     # Локальная переменная
    print "Значение glob1 внутри функции = %s" % glob1
```

```
glob1 = 10                          # Глобальная переменная
f_test()                            # Вызываем функцию
print "Значение glob1 вне функции = %s" % glob1
try:
    print local1                    # Вызовет исключение NameError
except NameError:                   # Обрабатываем исключение
    print "Переменная local1 не видна вне функции"
```

Результат выполнения:

```
Значение glob1 внутри функции = 25
Значение glob1 вне функции = 10
Переменная local1 не видна вне функции
```

Как видно из примера, переменная `local1`, объявленная внутри функции `f_test()`, недоступна вне функции. Объявление внутри функции локальной переменной `glob1` не изменило значения одноименной глобальной переменной.

Если обращение к переменной производится до присваивания значения (даже если существует одноименная глобальная переменная), то будет возбуждено исключение `UnboundLocalError` (листинг 11.25).

Листинг 11.25. Ошибка при обращении к переменной до присваивания значения

```
def f_test():
    # Локальная переменная еще не определена
    print glob1                     # Эта строка вызовет ошибку!!!
    glob1 = 25                      # Локальная переменная
glob1 = 10                          # Глобальная переменная
f_test()                            # Вызываем функцию
# Результат выполнения:
# UnboundLocalError: local variable 'glob1' referenced before assignment
```

Для того чтобы значение глобальной переменной можно было изменить внутри функции, необходимо объявить переменную глобальной с помощью ключевого слова `global`. Продемонстрируем это на примере (листинг 11.26).

Листинг 11.26. Использование ключевого слова `global`

```
def f_test():
    # Объявляем переменную glob1 глобальной
    global glob1
    glob1 = 25        # Изменяем значение глобальной переменной
    print "Значение glob1 внутри функции = %s" % glob1
glob1 = 10            # Глобальная переменная
print "Значение glob1 вне функции = %s" % glob1
f_test()              # Вызываем функцию
print "Значение glob1 после функции = %s" % glob1
```

Результат выполнения:

```
Значение glob1 вне функции = 10
Значение glob1 внутри функции = 25
Значение glob1 после функции = 25
```

Таким образом, поиск идентификатора, используемого внутри функции, будет производиться в следующем порядке:

1. Поиск объявления идентификатора внутри функции (в локальной области видимости).
2. Поиск объявления идентификатора в глобальной области.
3. Поиск во встроенной области видимости (встроенные функции, операторы, ключевые слова и т. д.).

При использовании анонимных функций следует учитывать, что при указании внутри функции глобальной переменной будет сохранена ссылка на эту переменную, а не ее значение в момент определения функции:

```
x = 5
# Сохраняется ссылка, а не значение переменной x!!!
f_test = lambda: x
x = 80                          # Изменили значение
print f_test()                  # Выведет: 80, а не 5
```

Если необходимо сохранить именно текущее значение переменной, то можно воспользоваться способом, приведенным в листинге 11.27.

Листинг 11.27. Сохранение значения переменной

```
x = 5
# Сохраняется значение переменной x
f_test = (lambda y: lambda: y)(x)
x = 80                                  # Изменили значение
print f_test()                          # Выведет: 5
```

Обратите внимание на третью строку примера. В ней мы определили анонимную функцию с одним параметром, возвращающую ссылку на вложенную анонимную функцию. Далее мы вызываем первую функцию с помощью круглых скобок и передаем ей значение переменной `x`. В результате сохраняется текущее значение переменной, а не ссылка на нее.

Сохранить текущее значение переменной можно так же, указав глобальную переменную в качестве значения параметра по умолчанию в определении функции (листинг 11.28).

Листинг 11.28. Сохранение значения с помощью параметра по умолчанию

```
x = 5
# Сохраняется значение переменной x
f_test = lambda x=x: x
x = 80                                  # Изменили значение
print f_test()                          # Выведет: 5
```

Получить все идентификаторы и их значения позволяют следующие функции:

- `globals()` — возвращает словарь с глобальными идентификаторами;
- `locals()` — возвращает словарь с локальными идентификаторами. Пример:

```
def f_test():
    local1 = 54
    glob2 = 25
    print "Глобальные идентификаторы внутри функции"
    print sorted(globals().keys())
    print "Локальные идентификаторы внутри функции"
    print sorted(locals().keys())
glob1, glob2 = 10, 88
f_test()
print "Глобальные идентификаторы вне функции"
print sorted(globals().keys())
```

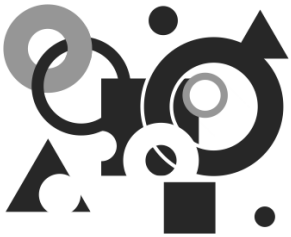
Результат выполнения:

```
Глобальные идентификаторы внутри функции
['__builtins__', '__doc__', '__file__', '__name__',
'__package__', 'f_test', 'glob1', 'glob2', 'idlelib']
Локальные идентификаторы внутри функции
['glob2', 'local1']
Глобальные идентификаторы вне функции
['__builtins__', '__doc__', '__file__', '__name__',
'__package__', 'f_test', 'glob1', 'glob2', 'idlelib']
```

- `vars([<Объект>])` — если вызывается без параметра внутри функции, то возвращает словарь с локальными идентификаторами. Если вызывается без параметра вне функции, то возвращает словарь с глобальными идентификаторами. При указании объекта в качестве параметра возвращает идентификаторы этого объекта (эквивалентно вызову `<Объект>.__dict__`). Пример:

```
def f_test():
    local1 = 54
    glob2 = 25
    print "Локальные идентификаторы внутри функции"
    print sorted(vars().keys())
glob1, glob2 = 10, 88
f_test()
print "Глобальные идентификаторы вне функции"
print sorted(vars().keys())
print "Указание объекта в качестве параметра"
print sorted(vars(__builtins__.dict).keys())
print "Альтернативный вызов"
print sorted(__builtins__.dict.__dict__.keys())
```

Глава 12



Модули и пакеты

Модулем в языке Python называется любой файл с программой. Каждый модуль может импортировать другой модуль, получая, таким образом, доступ к идентификаторам внутри импортированного модуля. Следует заметить, что импортируемый модуль может содержать программу не только на языке Python. Например, можно импортировать скомпилированный модуль, написанный на языке C.

Все программы, которые мы запускали ранее, были расположены в модуле с названием `"__main__"`. Получить имя модуля позволяет предопределенный атрибут `__name__`. Атрибут `__name__` для запускаемого модуля содержит значение `"__main__"`, а для импортируемого модуля — его имя. Выведем название модуля:

```
print __name__                    # Выведет: __main__
```

Проверить, является модуль главной программой или импортированным модулем, позволяет код, приведенный в листинге 12.1.

Листинг 12.1. Проверка способа запуска модуля

```
if __name__ == "__main__":
    print "Это главная программа"
else:
    print "Импортированный модуль"
```

12.1. Инструкция *import*

Импортировать модуль позволяет инструкция `import`. Мы уже не раз использовали эту инструкцию для подключения встроенных модулей. Например, подключали модуль `time` для получения текущей даты с помощью функции `strftime()`:

```
import time                       # Импортируем модуль time
print time.strftime("%d.%m.%Y")   # Выводим текущую дату
```

Инструкция `import` имеет следующий формат:

```
import <Название модуля 1> [as <Псевдоним 1>][, ...,
       <Название модуля N> [as <Псевдоним N>]]
```

После ключевого слова `import` указывается название модуля. Обратите внимание на то, что название не должно содержать расширения и пути к файлу. При именовании модулей необходимо учитывать, что операция импорта создает одноименный идентификатор. Это означает, что название модуля должно полностью соответствовать правилам именований переменных. Можно создать модуль с именем, начинающимся с цифры, но подключить такой модуль будет нельзя. Кроме того, следует избегать совпадения с ключевыми словами, встроенными идентификаторами и названиями модулей, входящих в стандартную библиотеку.

За один раз можно импортировать сразу несколько модулей, перечислив их через запятую. В качестве примера подключим модули `time` и `math` (листинг 12.2).

Листинг 12.2. Подключение нескольких модулей сразу

```
import time, math                # Импортируем несколько модулей сразу
print time.strftime("%d.%m.%Y") # Текущая дата
print math.pi                    # Число pi
```

После импортирования модуля его название становится идентификатором, через который можно получить доступ к атрибутам, определенным внутри модуля. Доступ к атрибутам модуля осуществляется с помощью точечной нотации. Например, обратиться к константе `pi`, расположенной внутри модуля `math`, можно так:

```
math.pi
```

Функция `getattr()` позволяет получить значение атрибута модуля по его названию, заданному в виде строки. С помощью этой функции можно сформировать название атрибута динамически во время выполнения программы. Формат функции:

```
getattr(<Объект модуля>, <Атрибут>[, <Значение по умолчанию>])
```

Если указанный атрибут не найден, возбуждается исключение `AttributeError`. Чтобы избежать вывода сообщения об ошибке, можно в третьем параметре указать значение, которое будет возвращаться, если атрибут не существует. Пример использования функции приведен в листинге 12.3.

Листинг 12.3. Пример использования функции `getattr()`

```
import math
print getattr(math, "pi")       # Число pi
print getattr(math, "x", 50)    # Число 50, т. к. x не существует
```

Проверить существование атрибута позволяет функция `hasattr(<Объект>, <Название атрибута>)`. Если атрибут существует, функция возвращает значение `True`. Напишем функцию проверки существования атрибута в модуле `math` (листинг 12.4).

Листинг 12.4. Проверка существования атрибута

```
import math
def f_hasattr_math(attr):
    if hasattr(math, attr):
        return "Атрибут существует"
    else:
        return "Атрибут не существует"
print f_hasattr_math("pi")       # Атрибут существует
print f_hasattr_math("x")        # Атрибут не существует
```

Если название модуля является слишком длинным и его неудобно указывать каждый раз для доступа к идентификаторам внутри модуля, то можно создать псевдоним. Псевдоним задается после ключевого слова `as`. Создадим псевдоним для модуля `math` (листинг 12.5).

Листинг 12.5. Использование псевдонимов

```
import math as m                # Создание псевдонима
print m.pi                      # Число pi
```

Теперь доступ к атрибутам модуля `math` может осуществляться только с помощью идентификатора `m`. Идентификатор `math` в этом случае использовать уже нельзя.

Все идентификаторы внутри импортированного модуля доступны только через идентификатор, указанный в инструкции `import`. Это означает, что любая глобальная переменная на самом деле является глобальной переменной модуля. По этой причине модули часто используются как пространства имен. В качестве примера создадим модуль под названием tests.py, в котором определим переменную `x` (листинг 12.6).

Листинг 12.6. Содержимое модуля tests.py

```
# -*- coding: cp866 -*-
x = 50
```

В основной программе также определим переменную `x`, но с другим значением. Затем подключим файл tests.py и выведем значения переменных (листинг 12.7).

Листинг 12.7. Содержимое основной программы

```
# -*- coding: cp866 -*-
import tests            # Подключаем файл tests.py
x = 22
print tests.x           # Значение переменной x внутри модуля
print x                 # Значение переменной x в основной программе
raw_input()
```

Оба файла размещаем в одной папке, а затем запускаем файл с основной программой с помощью двойного щелчка на значке файла. Как видно из результата, никакого конфликта имен нет, т. к. одноименные переменные расположены в разных пространствах имен.

Обратите внимание на содержимое папки с файлами после подключения модуля tests.py. Внутри папки автоматически был создан файл tests.pyc. Этот файл содержит скомпилированный байт-код одноименного модуля. Байт-код создается при первом импортировании модуля и изменяется только после изменения кода внутри модуля. При всех последующих подключениях модуля tests.py будет исполняться код из файла tests.pyc. Следует заметить, что для импортирования модуля достаточно иметь только файл tests.pyc. Для примера переименуйте файл tests.py (например, в tests1.py) и запустите основную программу. Программа будет нормально выполняться. Таким образом, чтобы скрыть исходный код модулей, можно поставлять программу клиентам только с файлами, имеющими расширение pyc.

Существует еще одно обстоятельство, на которое следует обратить особое внимание. Импортирование модуля выполняется только при первом вызове инструкции `import`. При каждом вызове инструкции `import` проверяется наличие объекта модуля в словаре `modules` из модуля `sys`. Если ссылка на модуль находится в этом словаре, то модуль повторно импортироваться не будет. В качестве примера выведем ключи словаря `modules`, предварительно отсортировав их (листинг 12.8).

Листинг 12.8. Вывод ключей словаря `modules`

```
# -*- coding: cp866 -*-
import tests, sys                  # Подключаем модули tests и sys
print sorted(sys.modules.keys())
raw_input()
```

Инструкция `import` требует явного указания объекта модуля. Например, передать название модуля в виде строки нельзя. Чтобы подключить модуль, название которого создается динамически в зависимости от определенных условий, следует воспользоваться функцией `__import__()`. В качестве примера подключим модуль tests.py с помощью функции `__import__()` (листинг 12.9).

Листинг 12.9. Использование функции `__import__()`

```
# -*- coding: cp866 -*-
s = "test" + "s"             # Динамическое создание названия модуля
m = __import__(s)            # Подключение модуля tests
print m.x                    # Вывод значения атрибута x
raw_input()
```

Получить список всех идентификаторов внутри модуля позволяет функция `dir()`. Кроме того, можно воспользоваться словарем `__dict__`, который содержит все идентификаторы и их значения (листинг 12.10).

Листинг 12.10. Вывод списка всех идентификаторов

```
# -*- coding: cp866 -*-
import tests
print dir(tests)
print sorted(tests.__dict__.keys())
raw_input()
```

12.2. Инструкция *from*

Для импортирования определенных идентификаторов из модуля можно воспользоваться инструкцией `from`. Инструкция имеет несколько форматов:

```
from <Название модуля> import <Идентификатор 1> [as <Псевдоним 1>]
                      [, ..., <Идентификатор N> [as <Псевдоним N>]]
from <Название модуля> import (<Идентификатор 1> [as <Псевдоним 1>],
                         [..., <Идентификатор N> [as <Псевдоним N>]])
from <Название модуля> import *
```

Первые два формата позволяют импортировать модуль и сделать доступными только указанные идентификаторы. Для длинных имен можно назначить псевдоним, указав его после ключевого слова `as`. В качестве примера сделаем доступными константу `pi` и функцию `floor()` из модуля `math`, а для названия функции создадим псевдоним (листинг 12.11).

Листинг 12.11. Инструкция `from`

```
# -*- coding: cp866 -*-
from math import pi, floor as f
print pi                                # Вывод числа pi
# Вызываем функцию floor() через идентификатор f
print f(5.49)                           # Выведет: 5.0
raw_input()
```

Идентификаторы можно разместить на нескольких строках, указав их названия через запятую внутри круглых скобок:

```
from math import (pi, floor,
                  sin, cos)
```

Третий формат инструкции `from` позволяет импортировать все идентификаторы из модуля. Для примера импортируем все идентификаторы из модуля `math` (листинг 12.12).

Листинг 12.12. Импорт всех идентификаторов из модуля

```
# -*- coding: cp866 -*-
from math import *    # Импортируем все идентификаторы из модуля math
```

```
print pi              # Вывод числа pi
print floor(5.49)     # Вызываем функцию floor()
raw_input()
```

Следует заметить, что идентификаторы, названия которых начинаются с символа подчеркивания, импортированы не будут. Кроме того, необходимо учитывать, что импортирование всех идентификаторов из модуля может нарушить пространство имен главной программы, т. к. идентификаторы, имеющие одинаковые имена, будут перезаписаны. Создадим два модуля и подключим их с помощью инструкций `from` и `import`. Содержимое файла module1.py приведено в листинге 12.13.

Листинг 12.13. Содержимое файла module1.py

```
# -*- coding: cp866 -*-
s = "Значение из модуля module1"
```

Содержимое файла module2.py приведено в листинге 12.14.

Листинг 12.14. Содержимое файла module2.py

```
# -*- coding: cp866 -*-
s = "Значение из модуля module2"
```

Исходный код основной программы приведен в листинге 12.15.

Листинг 12.15. Содержимое основной программы

```
# -*- coding: cp866 -*-
from module1 import *
from module2 import *
import module1, module2
print s                  # Выведет: "Значение из модуля module2"
print module1.s          # Выведет: "Значение из модуля module1"
print module2.s          # Выведет: "Значение из модуля module2"
raw_input()
```

Размещаем все файлы в одной папке, а затем запускаем основную программу с помощью двойного щелчка на значке файла. Итак, в обоих модулях определена переменная с именем `s`. При импортировании всех идентификаторов значением переменной `s` будет значение из модуля, который был импортирован последним. В нашем случае это значение из модуля module2.py. Получить доступ к обеим переменным можно только при использовании инструкции `import`. Благодаря точечной нотации пространство имен не нарушается.

В атрибуте `__all__` можно указать список идентификаторов, которые будут импортироваться с помощью выражения `from module import *`. Идентификато-

ры внутри списка указываются в виде строки. Создадим файл module1.py (листинг 12.16).

Листинг 12.16. Использование атрибута __all__

```
# -*- coding: cp866 -*-
x, y, z, _s = 10, 80, 22, "Строка"
__all__ = ["x", "_s"]
```

Затем подключим его к основной программе (листинг 12.17).

Листинг 12.17. Содержимое основной программы

```
# -*- coding: cp866 -*-
from module1 import *
print sorted(vars().keys()) # Получаем список всех идентификаторов
raw_input()
```

После запуска основной программы (с помощью двойного щелчка на значке файла) получим следующий результат:

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
 '_s', 'x']
```

Как видно из примера, были импортированы только переменные `_s` и `x`. Если бы мы не указали идентификаторы внутри списка `__all__`, то результат был бы другим:

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'x', 'y', 'z']
```

Обратите внимание на то, что переменная `_s` в этом случае не копируется из модуля, т. к. ее имя начинается с символа подчеркивания.

12.3. Пути поиска модулей

До сих пор мы размещали модули в одной папке с исполняемым файлом. В этом случае нет необходимости настраивать пути поиска модулей, т. к. папка с исполняемым файлом автоматически добавляется в начало списка путей. Получить полный список путей поиска позволяет следующий код:

```
>>> import sys            # Подключаем модуль sys
>>> sys.path              # path содержит список путей поиска модулей
```

Список `sys.path` содержит пути поиска, получаемые из следующих источников:

- путь к текущему каталогу с исполняемым файлом;
- значение переменной окружения `PYTHONPATH`. Для добавления переменной в меню **Пуск** выбираем пункт **Панель управления** (или **Настройка | Панель управления**). В открывшемся окне выбираем пункт **Система**. Переходим на

вкладку **Дополнительно** и нажимаем кнопку **Переменные среды**. В разделе **Переменные среды пользователя** нажимаем кнопку **Создать**. В поле **Имя переменной** вводим "PYTHONPATH", а в поле **Значение переменной** задаем пути к папкам с модулями через точку с запятой, например "C:\folder1;C:\folder2". После этого изменения перезагружать компьютер не нужно, достаточно заново запустить программу;

- пути поиска стандартных модулей;
- содержимое файлов с расширением pth, расположенных в каталогах поиска стандартных модулей, например, в каталоге C:\Python26\Lib\site-packages. Название файла может быть произвольным, главное, чтобы расширение файла было pth. Каждый путь (абсолютный или относительный) должен быть расположен на отдельной строке. В качестве примера создайте файл mypath.pth в каталоге C:\Python26\Lib\site-packages со следующим содержимым:

```
# Это комментарий
C:\folder1
C:\folder2
```

Обратите внимание на то, что каталоги должны существовать, в противном случае они не будут добавлены в список `sys.path`.

При поиске модуля список `sys.path` просматривается слева направо. Поиск прекращается после первого найденного модуля. Таким образом, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, то будет использоваться модуль из папки C:\folder1, т. к. он расположен первым в списке путей поиска.

Список `sys.path` можно изменять из программы с помощью списковых методов. Например, добавить каталог в конец списка можно с помощью метода `append()`, а в начало списка — с помощью метода `insert()` (листинг 12.18).

Листинг 12.18. Изменение списка путей поиска модулей

```
# -*- coding: cp866 -*-
import sys
sys.path.append(r"C:\folder1")          # Добавляем в конец списка
sys.path.insert(0, r"C:\folder2")       # Добавляем в начало списка
print sys.path
raw_input()
```

В этом примере мы добавили папку C:\folder2 в начало списка. Теперь, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, то будет использоваться модуль из папки C:\folder2, а не в C:\folder1, как в предыдущем примере.

Обратите внимание на символ `r` перед открывающей кавычкой. В этом режиме специальные последовательности символов не интерпретируются. Если используются обычные строки, то необходимо удвоить каждый слеш в пути:

```
sys.path.append("C:\\folder1\\folder2\\folder3")
```

12.4. Повторная загрузка модулей

Как вы уже знаете, модуль загружается только один раз при первой операции импорта. Все последующие операции импортирования этого модуля будут возвращать уже загруженный объект модуля, даже если сам модуль был изменен. Чтобы повторно загрузить модуль, следует воспользоваться функцией `reload()`. Формат функции:

```
reload(<Объект модуля>)
```

В качестве примера создадим модуль tests.py со следующим содержимым:

```
# -*- coding: cp1251 -*-
x = 150
```

Подключим этот модуль в окне **Python Shell** редактора IDLE и выведем текущее значение переменной `x`:

```
>>> import tests                # Подключаем модуль tests.py
>>> print tests.x               # Выводим текущее значение
150
```

Не закрывая окно **Python Shell**, изменим значение переменной `x` на `800`, а затем попробуем заново импортировать модуль и вывести текущее значение переменной:

```
>>> # Изменяем значение в модуле на 800
>>> import tests
>>> print tests.x               # Значение не изменилось
150
```

Как видно из примера, значение переменной `x` не изменилось. Теперь перезагрузим модуль с помощью функции `reload()` (листинг 12.19).

Листинг 12.19. Повторная загрузка модуля

```
>>> reload(tests)               # Перезагружаем модуль
<module 'tests' from 'C:\book\tests.py'>
>>> print tests.x               # Значение изменилось
800
```

При использовании функции `reload()` следует учитывать, что идентификаторы, импортированные с помощью инструкции `from`, перезагружены не будут. Кроме того, повторно не загружаются скомпилированные модули, написанные на других языках программирования, например, C.

12.5. Пакеты

Пакетом называется каталог с модулями, в котором расположен файл инициализации __init__.py. Файл инициализации может быть пустым или содержать код, который будет выполнен при первом обращении к пакету. В любом случае он обязательно должен присутствовать внутри каталога с модулями.

В качестве примера создадим следующую структуру файлов и каталогов:

```
main.py                 # Основной файл с программой
folder1\                # Папка на одном уровне вложенности с main.py
    __init__.py         # Файл инициализации
    module1.py          # Модуль folder1\module1.py
    folder2\            # Вложенная папка
        __init__.py     # Файл инициализации
        module2.py      # Модуль folder1\folder2\module2.py
        module3.py      # Модуль folder1\folder2\module3.py
```

Содержимое файлов __init__.py приведено в листинге 12.20.

Листинг 12.20. Содержимое файлов __init__.py

```
# -*- coding: cp866 -*-
print "__init__ из", __name__
```

Содержимое модулей module1.py, module2.py и module3.py приведено в листинге 12.21.

Листинг 12.21. Содержимое модулей module1.py, module2.py и module3.py

```
# -*- coding: cp866 -*-
msg = "Модуль %s" % __name__
```

Теперь импортируем эти модули в основном файле main.py и получим значение переменной `msg` разными способами. Файл main.py будем запускать с помощью двойного щелчка на значке файла. Содержимое файла main.py приведено в листинге 12.22.

Листинг 12.22. Содержимое файла main.py

```
# -*- coding: cp866 -*-

# Доступ к модулю folder1\module1.py
import folder1.module1 as m1
                        # Выведет: __init__ из folder1
print m1.msg            # Выведет: Модуль folder1.module1
from folder1 import module1 as m2
print m2.msg            # Выведет: Модуль folder1.module1
from folder1.module1 import msg
print msg               # Выведет: Модуль folder1.module1

# Доступ к модулю folder1\folder2\module2.py
import folder1.folder2.module2 as m3
                        # Выведет: __init__ из folder1.folder2
print m3.msg            # Выведет: Модуль folder1.folder2.module2
```

```
from folder1.folder2 import module2 as m4
print m4.msg             # Выведет: Модуль folder1.folder2.module2
from folder1.folder2.module2 import msg
print msg                # Выведет: Модуль folder1.folder2.module2

raw_input()
```

Как видно из примера, пакеты позволяют распределить модули по каталогам. Чтобы импортировать модуль, расположенный во вложенном каталоге, необходимо указать путь к нему, перечислив имена каталогов через точку. Если модуль расположен в каталоге C:\folder1\folder2\, то путь к нему из C:\ будет выглядеть так: `folder1.folder2`. При использовании инструкции `import` путь к модулю должен включать не только названия каталогов, но и название модуля без расширения:

```
import folder1.folder2.module2
```

Получить доступ к идентификаторам внутри импортированного модуля можно следующим образом:

```
print folder1.folder2.module2.msg
```

Так как постоянно указывать такой длинный идентификатор очень неудобно, можно создать псевдоним, указав его после ключевого слова `as`, и обращаться к идентификаторам модуля через него:

```
import folder1.folder2.module2 as m
print m.msg
```

При использовании инструкции `from` можно импортировать как объект модуля, так и определенные идентификаторы из модуля. Чтобы импортировать объект модуля, его название следует указать после ключевого слова `import`:

```
from folder1.folder2 import module2
print module2.msg
```

Для импортирования только определенных идентификаторов название модуля указывается в составе пути, а после ключевого слова `import` через запятую перечисляются идентификаторы:

```
from folder1.folder2.module2 import msg
print msg
```

Если необходимо импортировать все идентификаторы из модуля, то после ключевого слова `import` указывается символ `*`:

```
from folder1.folder2.module2 import *
print msg
```

Инструкция `from` позволяет также импортировать сразу несколько модулей из пакета. Для этого внутри файла инициализации __init__.py в атрибуте `__all__` необходимо указать список модулей, которые будут импортироваться с помощью выражения `from` *пакет* `import *`. В качестве примера изменим содержимое файла __init__.py из каталога C:\folder1\folder2\:

```
# -*- coding: cp866 -*-
__all__ = ["module2", "module3"]
```

Теперь изменим содержимое основного файла main.py (листинг 12.23) и запустим его.

Листинг 12.23. Содержимое файла main.py

```
# -*- coding: cp866 -*-
from folder1.folder2 import *
print module2.msg         # Выведет: Модуль folder1.folder2.module2
print module3.msg         # Выведет: Модуль folder1.folder2.module3
raw_input()
```

Как видно из примера, после ключевого слова `from` указывается лишь путь к каталогу без имени модуля. В результате выполнения инструкции `from` все модули, указанные в списке `__all__`, будут импортированы в пространство имен модуля main.py.

До сих пор мы рассматривали импортирование модулей из основной программы. Теперь рассмотрим импорт модулей внутри пакета. В этом случае инструкция `from` поддерживает относительный импорт модулей. Чтобы импортировать модуль, расположенный в том же каталоге, перед названием модуля указывается точка:

```
from .module import *
```

Чтобы импортировать модуль, расположенный в родительском каталоге, перед названием модуля указываются две точки:

```
from ..module import *
```

Если необходимо обратиться еще уровнем выше, то указываются три точки:

```
from ...module import *
```

Чем выше уровень, тем больше точек необходимо указать. После ключевого слова `from` можно ставить только точки. В этом случае имя модуля вводится после ключевого слова `import`. Пример:

```
from .. import module
```

Рассмотрим относительный импорт на примере. Для этого изменим содержимое модуля module3.py, как показано в листинге 12.24.

Листинг 12.24. Содержимое модуля module3.py

```
# -*- coding: cp866 -*-

# Импорт модуля module2.py из текущего каталога
from . import module2 as m1
var1 = "Значение из: %s" % m1.msg
from .module2 import msg as m2
var2 = "Значение из: %s" % m2

# Импорт модуля module1.py из родительского каталога
from .. import module1 as m3
```

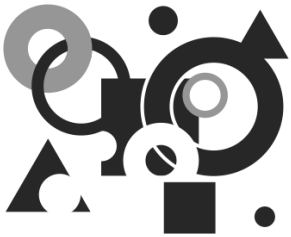
```
var3 = "Значение из: %s" % m3.msg
from ..module1 import msg as m4
var4 = "Значение из: %s" % m4
```

Теперь изменим содержимое основного файла main.py (листинг 12.25) и запустим его с помощью двойного щелчка на значке файла.

Листинг 12.25. Содержимое файла main.py

```
# -*- coding: cp866 -*-
from folder1.folder2 import module3 as m
print m.var1          # Значение из: Модуль folder1.folder2.module2
print m.var2          # Значение из: Модуль folder1.folder2.module2
print m.var3          # Значение из: Модуль folder1.module1
print m.var4          # Значение из: Модуль folder1.module1
raw_input()
```

Глава 13



Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — это способ организации программы, позволяющий использовать один и тот же код многократно. В отличие от функций и модулей ООП позволяет не только разделить программу на фрагменты, но и описать предметы реального мира в виде объектов, а также организовать связи между этими объектами.

Основным "кирпичиком" ООП является класс. *Класс* — это объект, включающий набор переменных и функций для управления этими переменными. Переменные называют *атрибутами*, а функции — *методами*. Класс является фабрикой объектов, т. е. позволяет создать неограниченное количество объектов, основанных на этом классе.

13.1. Определение класса и создание экземпляра класса

Класс описывается с помощью ключевого слова `class` по следующей схеме:

```
class <Название класса>[(<Класс1>[, ..., <КлассN>])]:
    [""" Строка документирования """]
    <Описание атрибутов и методов>
```

Инструкция создает новый объект и присваивает ссылку на него идентификатору, указанному после ключевого слова `class`. Это означает, что название класса должно полностью соответствовать правилам именований переменных. После названия класса в круглых скобках можно указать один или несколько базовых классов через запятую. Если класс не наследует базовые классы, то круглые скобки можно не указывать. Следует заметить, что все выражения внутри инструкции `class` выполняются при создании объекта, а не при создании экземпляра класса.

В качестве примера создадим класс, внутри которого просто выводится сообщение (листинг 13.1).

Листинг 13.1. Создание определения класса

```
# -*- coding: cp866 -*-
class Class1:
    """ Это строка документирования """
    print "Инструкции выполняются сразу"
raw_input()
```

Этот пример содержит только определение класса `Class1` и не создает экземпляр класса. Как только поток выполнения достигнет инструкции `class`, сообщение, указанное в операторе `print`, будет сразу выведено.

Создание переменной (атрибута) внутри класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, с помощью инструкции `def`. Методам класса в первом параметре автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть именем `self`, хотя это и не обязательно. Доступ к атрибутам и методам класса производится через переменную `self` с помощью точечной нотации. Например, к атрибуту `var1` из метода класса можно обратиться так: `self.var1`.

Чтобы использовать атрибуты и методы класса, необходимо создать экземпляр класса. Для этого используется следующий синтаксис:

```
<Экземпляр класса> = <Название класса>([<Параметры>])
```

Определим класс `Class1` с атрибутом `var1` и методом `f_print()`, выводящим значение этого атрибута, а затем создадим экземпляр класса и вызовем метод (листинг 13.2).

Листинг 13.2. Создание атрибута и метода

```
class Class1:
    var1 = 10              # Переменная внутри класса
    def f_print(self):     # self - это ссылка на экземпляр класса
        print self.var1    # Выводим значение переменной
c1 = Class1()              # Создание экземпляра класса
                           # Вызываем метод f_print()
c1.f_print()               # self не указывается при вызове метода
print c1.var1              # К атрибуту можно обратиться непосредственно
```

При обращении к методам класса используется следующий формат:

```
<Экземпляр класса>.<Имя метода>([<Параметры>])
```

Обратите внимание на то, что при вызове метода не нужно передавать ссылку на экземпляр класса в качестве параметра, как это делается в определении метода внутри класса. Ссылку на экземпляр класса интерпретатор передает автоматически.

Обращение к атрибутам класса осуществляется аналогично:

```
<Экземпляр класса>.<Имя атрибута>
```

Для доступа к атрибутам и методам можно также использовать следующие функции:

- `getattr()` — возвращает значение атрибута по его названию, заданному в виде строки. С помощью этой функции можно сформировать название атрибута динамически во время выполнения программы. Формат функции:

  ```
  getattr(<Объект>, <Атрибут>[, <Значение по умолчанию>])
  ```

  Если указанный атрибут не найден, возбуждается исключение `AttributeError`. Чтобы избежать вывода сообщения об ошибке, можно в третьем параметре указать значение, которое будет возвращаться, если атрибут не существует;
- `setattr()` — задает значение атрибута. Название атрибута указывается в виде строки. Формат функции:

  ```
  setattr(<Объект>, <Атрибут>, <Значение>)
  ```

- `delattr(<Объект>, <Атрибут>)` — удаляет указанный атрибут. Название атрибута указывается в виде строки;
- `hasattr(<Объект>, <Атрибут>)` — проверяет наличие указанного атрибута. Если атрибут существует, функция возвращает значение `True`.

Продемонстрируем работу функций на примере (листинг 13.3).

Листинг 13.3. Функции `getattr()`, `setattr()` и `hasattr()`

```
class Class1:
    var1 = 10
    def f_test(self):
        return self.var1
c1 = Class1()                    # Создаем экземпляр класса
print getattr(c1, "var1")        # Выведет: 10
print getattr(c1, "f_test")()    # Выведет: 10
print getattr(c1, "var2", 0)     # Выведет: 0, т. к. атрибут не найден
setattr(c1, "var2", 20)          # Создаем атрибут var2
print getattr(c1, "var2", 0)     # Выведет: 20
delattr(c1, "var2")              # Удаляем атрибут var2
print getattr(c1, "var2", 0)     # Выведет: 0, т. к. атрибут не найден
print hasattr(c1, "var1")        # Выведет: True
print hasattr(c1, "var2")        # Выведет: False
```

Все атрибуты класса в языке Python являются открытыми (`public`), т. е. доступны для непосредственного изменения. Кроме того, атрибуты можно создавать динамически после создания класса. Можно создать как атрибут объекта класса, так и атрибут экземпляра класса. Рассмотрим это на примере (листинг 13.4).

Листинг 13.4. Атрибуты объкта класса и экземпляра класса

```
class Class1:                       # Определяем пустой класс
    pass
Class1.var1 = 50                    # Создаем атрибут объекта класса
c1, c2 = Class1(), Class1()         # Создаем два экземпляра класса
c1.var2 = 10                        # Создаем атрибут экземпляра класса
c2.var2 = 20                        # Создаем атрибут экземпляра класса
print c1.var1, c1.var2              # Выведет: 50, 10
print c2.var1, c2.var2              # Выведет: 50, 20
```

В этом примере мы определяем пустой класс, разместив в нем инструкцию `pass`. Далее создаем атрибут объекта класса (`var1`). Этот атрибут будет доступен всем создаваемым экземплярам класса. Затем создаем два экземпляра класса и добавляем одноименные атрибуты (`var2`). Значение этого атрибута будет разным в каждом экземпляре класса. Если создать новый экземпляр (например, `c3`), то атрибут `var2` в нем определен не будет. Таким образом, с помощью классов можно имитировать типы данных, определенные в других языках программирования, например, тип `struct` в языке C++.

13.2. Методы *__init__()* и *__del__()*

При создании экземпляра класса интерпретатор автоматически вызывает метод инициализации `__init__()`. В других языках программирования такой метод принято называть *конструктором класса*. Формат метода:

```
def __init__(self[, <Значение1>[, ..., <ЗначениеN>]]):
    <Выражения>
```

С помощью метода `__init__()` можно присвоить начальные значения атрибутам класса. При создании экземпляра класса начальные значения указываются после имени класса в круглых скобках:

```
<Экземпляр класса> = <Имя класса>([<Значение1>[, ..., <ЗначениеN>]])
```

Пример использования метода `__init__()` приведен в листинге 13.5.

Листинг 13.5. Метод `__init__()`

```
class Class1:
    def __init__(self, value1, value2): # Конструктор
        self.var1 = value1
        self.var2 = value2
c1 = Class1(100, 300)                    # Создаем экземпляр класса
print c1.var1, c1.var2                   # Выведет: 100 300
```

Если конструктор вызывается при создании объекта, то перед уничтожением объекта автоматически вызывается метод, называемый *деструктором*. В языке

Python деструктор реализуется в виде предопределенного метода `__del__()` (листинг 13.6). Следует заметить, что метод не будет вызван, если на экземпляр класса существует хотя бы одна ссылка. Кроме того, т. к. интерпретатор самостоятельно заботится об удалении объектов, использование деструктора в языке Python не имеет особого смысла.

Листинг 13.6. Метод `__del__()`

```
class Class1:
    def __init__(self): # Конструктор класса
        print "Вызван метод __init__()"
    def __del__(self):  # Деструктор класса
        print "Вызван метод __del__()"
c1 = Class1()           # Выведет: Вызван метод __init__()
del c1                  # Выведет: Вызван метод __del__()
c2 = Class1()           # Выведет: Вызван метод __init__()
c3 = c2                 # Создаем ссылку на экземпляр класса
del c2                  # Ничего не выведет, т. к. существует ссылка
del c3                  # Выведет: Вызван метод __del__()
```

13.3. Наследование

Наследование является, пожалуй, самым главным понятием ООП. Предположим, у нас есть класс (например, `Class1`). При помощи *наследования* мы можем создать новый класс (например, `Class2`), в котором будет доступ ко всем атрибутам и методам класса `Class1`, а также к некоторым новым атрибутам и методам (листинг 13.7).

Листинг 13.7. Наследование

```
class Class1:         # Базовый класс
    def f_func1(self):
        print "Метод f_func1() класса Class1"
    def f_func2(self):
        print "Метод f_func2() класса Class1"

class Class2(Class1): # Класс Class2 наследует класс Class1
    def f_func3(self):
        print "Метод f_func3() класса Class2"

c1 = Class2()         # Создаем экземпляр класса Class2
c1.f_func1()          # Выведет: Метод f_func1() класса Class1
c1.f_func2()          # Выведет: Метод f_func2() класса Class1
c1.f_func3()          # Выведет: Метод f_func3() класса Class2
```

Как видно из примера, класс `Class1` указывается внутри круглых скобок в определении класса `Class2`. Таким образом, класс `Class2` наследует все атрибуты и методы класса `Class1`. Класс `Class1` называется базовым классом или суперклассом, а класс `Class2` — производным классом или подклассом.

Если имя метода в классе `Class2` совпадает с именем метода класса `Class1`, то будет использоваться метод из класса `Class2`. Чтобы вызвать одноименный метод из базового класса, следует указать перед методом название базового класса. Кроме того, в первом параметре метода необходимо явно указать ссылку на экземпляр класса. Рассмотрим это на примере (листинг 13.8).

Листинг 13.8. Переопределение методов

```
class Class1:               # Базовый класс
    def __init__(self):
        print "Конструктор базового класса"
    def f_func1(self):
        print "Метод f_func1() класса Class1"

class Class2(Class1):       # Класс Class2 наследует класс Class1
    def __init__(self):
        print "Конструктор производного класса"
        Class1.__init__(self) # Вызываем конструктор базового класса
    def f_func1(self):
        print "Метод f_func1() класса Class2"
        Class1.f_func1(self)  # Вызываем метод базового класса

c1 = Class2()                 # Создаем экземпляр класса Class2
c1.f_func1()                  # Вызываем метод f_func1()
```

Выведет:

```
Конструктор производного класса
Конструктор базового класса
Метод f_func1() класса Class2
Метод f_func1() класса Class1
```

ВНИМАНИЕ!

Конструктор базового класса автоматически не вызывается.

13.4. Множественное наследование

В определении класса в круглых скобках можно указать сразу несколько базовых классов через запятую. В этом случае поиск идентификаторов производится вначале в производном классе, затем в базовом классе, расположенном первым в

списке, далее просматриваются все базовые классы базового класса. Только после этого просматривается базовый класс, расположенный в списке правее, и все его базовые классы. Список базовых классов просматривается слева направо. Результатом поиска будет первый найденный идентификатор. Рассмотрим множественное наследование на примере (листинг 13.9).

Листинг 13.9. Множественное наследование

```
class Class1:         # Базовый класс для класса Class2
    def f_func1(self):
        print "Метод f_func1() класса Class1"
class Class2(Class1): # Класс Class2 наследует класс Class1
    def f_func2(self):
        print "Метод f_func2() класса Class2"
class Class3(Class1): # Класс Class3 наследует класс Class1
    def f_func1(self):
        print "Метод f_func1() класса Class3"
    def f_func2(self):
        print "Метод f_func2() класса Class3"
    def f_func3(self):
        print "Метод f_func3() класса Class3"
    def f_func4(self):
        print "Метод f_func4() класса Class3"
class Class4(Class2, Class3): # Множественное наследование
    def f_func4(self):
        print "Метод f_func4() класса Class4"
c1 = Class4()             # Создаем экземпляр класса Class4
c1.f_func1()              # Выведет: Метод f_func1() класса Class1
c1.f_func2()              # Выведет: Метод f_func2() класса Class2
c1.f_func3()              # Выведет: Метод f_func3() класса Class3
c1.f_func4()              # Выведет: Метод f_func4() класса Class4
```

Итак, метод `f_func1()` определен в двух классах — `Class1` и `Class3`. Так как класс `Class2` стоит первым в списке базовых классов, вначале просматривается этот класс, а затем все его базовые классы. Поэтому метод `f_func1()` будет найден в классе `Class1`, а не в классе `Class3`.

Метод `f_func2()` также определен в двух классах — `Class2` и `Class3`. Так как класс `Class2` стоит первым в списке базовых классов, то метод будет найден именно в этом классе. Чтобы наследовать метод из класса `Class3`, следует указать это явным образом. Переделаем определение класса `Class4` из предыдущего примера и наследуем метод `f_func2()` из класса `Class3` (листинг 13.10).

Листинг 13.10. Указание класса при наследовании метода

```
class Class4(Class2, Class3): # Множественное наследование
    # Наследуем f_func2() из класса Class3, а не из класса Class2
```

```
    f_func2 = Class3.f_func2
    def f_func4(self):
        print "Метод f_func4() класса Class4"
```

Метод `f_func3()` определен только в классе `Class3`, поэтому метод наследуется от этого класса. Метод `f_func4()`, определенный в классе `Class3`, переопределяется в производном классе. Если метод найден в производном классе, то вся иерархия наследования просматриваться не будет.

Если необходимо получить перечень базовых классов, то можно воспользоваться атрибутом `__bases__`. В качестве значения атрибут возвращает кортеж. В качестве примера выведем базовые классы для класса `Class4` из предыдущего примера:

```
print Class4.__bases__
```

Выведет:

```
(<class __main__.Class2 at 0x00A7DE10>, <class __main__.Class3
at 0x00A7DE40>)
```

13.5. Классы нового стиля

Начиная с Python 2.2, помимо классических классов (рассмотренных нами в предыдущих разделах) существуют классы так называемого нового стиля. *Классом нового стиля* называется класс, у которого базовым классом является встроенный объект (например, `list` или `dict`) или объект `object`. Для классов старого и нового стилей отличается результат выполнения функции `type()`, а также вывод атрибутов `__class__` и `__bases__` для экземпляров классов (листинг 13.11).

Листинг 13.11. Классы нового стиля

```
class Class1:          # Классический класс
    pass
class Class2(object):  # Класс нового стиля
    pass
class Class3(list):    # Класс нового стиля
    pass
print type(Class1)     # Выведет: <type 'classobj'>
print type(Class2)     # Выведет: <type 'type'>
print type(Class3)     # Выведет: <type 'type'>
# __bases__ содержит кортеж с базовыми классами
print Class1.__bases__ # Выведет: ()
print Class2.__bases__ # Выведет: (<type 'object'>,)
print Class3.__bases__ # Выведет: (<type 'list'>,)
c1, c2, c3 = Class1(), Class2(), Class3()
print c1.__class__     # Выведет: __main__.Class1
```

```
print c2.__class__      # Выведет: <class '__main__.Class2'>
print c3.__class__      # Выведет: <class '__main__.Class3'>
print type(c1)          # Выведет: <type 'instance'>
print type(c2)          # Выведет: <class '__main__.Class2'>
print type(c3)          # Выведет: <class '__main__.Class3'>
```

Кроме того, в классах нового стиля другой порядок поиска идентификаторов при множественном наследовании. Рассмотрим это на примере (листинг 13.12).

Листинг 13.12. Поиск идентификаторов при множественном наследовании

```
class Class1(object):  # Класс нового стиля
#class Class1:         # Классический класс
    var1 = "Это значение в классических классах"
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2):
    var1 = "Это значение в классах нового стиля"
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c1 = Class7()
print c1.var1
```

Если используются классические классы, то атрибут `var1` будет найден в классе `Class1`, т. к. просматривается вся ветка наследования для класса `Class4`. Если используются классы нового стиля, то атрибут `var1` будет найден в классе `Class5`, т. к. при достижении класса `Class2` поиск будет производиться в ветке класса `Class6`, стоящего в списке наследования вторым. Класс `Class2` для классов `Class4` и `Class6` является общим предком.

Последовательность поиска в классических классах будет такой:

```
Class7 -> Class4 -> Class3 -> Class2 -> Class1 - > Class6 -> Class5
```

Последовательность поиска в классах нового стиля:

```
Class7 -> Class4 -> Class3 - > Class6 -> Class5 -> Class2 -> Class1
```

13.6. Специальные методы

Классы старого и нового стилей поддерживают следующие специальные методы:

- `__call__()` — позволяет обработать вызов экземпляра класса как вызов функции. Формат метода:

  ```
  __call__(self[, <Параметр1>[, ..., <ПараметрN>]])
  ```

Пример:

```
class Class1(object):
    def __init__(self, m):
        self.msg = m
    def __call__(self):
        print self.msg
c1 = Class1("Значение1") # Создание экземпляра класса
c2 = Class1("Значение2") # Создание экземпляра класса
c1()                     # Выведет: Значение1
c2()                     # Выведет: Значение2
```

- `__setitem__(self, <Ключ>, <Значение>)` — вызывается в случае присваивания значения по ключу;
- `__getitem__(self, <Ключ>)` — вызывается при доступе к значению по ключу. Метод автоматически вызывается при использовании цикла `for`, а также при других операциях, применимых к последовательностям. Пример:

```
class Class1(object):
    def __init__(self, a):
        self.arr = a
    def __getitem__(self, index):
        return self.arr[index]
    def __setitem__(self, index, value):
        self.arr[index] = value
c1 = Class1( [1, 2, 3, 4, 5] )
print c1[0]  # Выведет: 1
c1[0] = 0    # Присваивание по индексу
print c1[0]  # Выведет: 0
for i in c1: # for автоматически вызывает __getitem__()
    print i,
# Выведет: 0 2 3 4 5
print list(c1) # Выведет: [0, 2, 3, 4, 5]
print "Есть" if 0 in c1 else "Нет" # Выведет: Есть
```

- `__delitem__(self, <Ключ>)` — вызывается в случае удаления элемента по ключу с помощью выражения `del <Экземпляр класса>[<Ключ>]`;
- `__getattr__(self, <Атрибут>)` — вызывается при обращении к несуществующему атрибуту класса. Пример:

```
class Class1(object):
    def __init__(self):
        self.i = 20
    def __getattr__(self, attr):
        print "Вызван метод __getattr__()"
        return 0
c1 = Class1()
# Атрибут i существует
print c1.i     # Выведет: 20. Метод __getattr__() не вызывается
```

```
# Атрибут s не существует
print c1.s     # Выведет: Вызван метод __getattr__() 0
```

- `__setattr__(self, <Атрибут>, <Значение>)` — вызывается при попытке присваивания значения атрибуту экземпляра класса. Если внутри метода необходимо присвоить значение атрибуту, то следует использовать словарь `__dict__`, иначе при точечной нотации метод `__setattr__()` будет вызван повторно, и это приведет к зацикливанию. Пример:

```
class Class1(object):
    def __setattr__(self, attr, value):
        print "Вызван метод __setattr__()"
        self.__dict__[attr] = value    # Только так!!!
c1 = Class1()
c1.i = 10  # Выведет: Вызван метод __setattr__()
print c1.i # Выведет: 10
```

- `__delattr__(self, <Атрибут>)` — вызывается при удалении атрибута с помощью выражения `del <Экземпляр класса>.<Атрибут>`;
- `__iter__(self)` — если метод определен, то считается, что объект поддерживает итерационный протокол. Если в классе одновременно определены методы `__iter__()` и `__getitem__()`, то предпочтение отдается методу `__iter__()`. Помимо метода `__iter__()` в классе должен быть определен метод `next()`, который будет вызываться на каждой итерации. Метод `next()` должен возвращать текущее значение или возбуждать исключение `StopIteration`, которое сообщает об окончании итераций. Пример:

```
class Class1(object):
    def __init__(self, a):
        self.arr = a
        self.i = 0 # Текущий индекс
    def __iter__(self):
        return self
    def next(self):
        if self.i >= len(self.arr):
            self.i = 0             # Устанавливаем в нач. состояние
            raise StopIteration    # Возбуждаем исключение
        else:
            elem = self.arr[self.i]
            self.i += 1
            # Возвращаем элемент по текущему индексу
            return elem
c1 = Class1( [1, 2, 3, 4, 5] )
for i in c1:
    print i,     # Выведет: 1 2 3 4 5
print c1.next()  # Выведет: 1
print c1.next()  # Выведет: 2
for i in c1:
    print i,     # Выведет: 3 4 5
```

- `__len__(self)` — вызывается при использовании функции `len()`. Метод должен возвращать положительное целое число. Пример:

```
class Class1(object):
    def __len__(self):
        return 50
c1 = Class1()
print len(c1) # Выведет: 50
```

- `__nonzero__(self)` — вызывается при использовании функции `bool()`;
- `__int__(self)` — вызывается при преобразовании объекта в целое число с помощью функции `int()`;
- `__long__(self)` — вызывается при преобразовании объекта в длинное целое число с помощью функции `long()`;
- `__float__(self)` — вызывается при преобразовании объекта в вещественное число с помощью функции `float()`;
- `__complex__(self)` — вызывается при использовании функции `complex()`;
- `__repr__(self)` и `__str__(self)` — служат для преобразования объекта в строку. Метод `__repr__()` вызывается при выводе в интерактивной оболочке, а также при использовании функции `repr()`. Метод `__str__()` вызывается при выводе с помощью оператора `print`, а также при использовании функции `str()`. Если метод `__str__()` отсутствует, то будет вызван метод `__repr__()`. В качестве значения методы `__repr__()` и `__str__()` должны возвращать строку. Пример:

```
class Class1(object):
    def __init__(self, m):
        self.msg = m
    def __repr__(self):
        return "Вызван метод __repr__() %s" % self.msg
    def __str__(self):
        return "Вызван метод __str__() %s" % self.msg
c1 = Class1("Значение")
print repr(c1) # Выведет: Вызван метод __repr__() Значение
print str(c1)  # Выведет: Вызван метод __str__() Значение
print c1       # Выведет: Вызван метод __str__() Значение
```

- `__unicode__(self)` — вызывается при использовании функции `unicode()`.

13.7. Перегрузка операторов

Перегрузка операторов позволяет экземплярам классов участвовать в обычных операциях. Чтобы перегрузить оператор, необходимо в классе определить метод со специальным названием.

Для перегрузки математических операторов используются следующие методы:

- `x + y` — сложение — `x.__add__(y)`;
- `y + x` — сложение (экземпляр класса справа) — `x.__radd__(y)`;
- `x += y` — сложение и присваивание — `x.__iadd__(y)`;
- `x - y` — вычитание — `x.__sub__(y)`;
- `y - x` — вычитание (экземпляр класса справа) — `x.__rsub__(y)`;
- `x -= y` — вычитание и присваивание — `x.__isub__(y)`;
- `x * y` — умножение — `x.__mul__(y)`;
- `y * x` — умножение (экземпляр класса справа) — `x.__rmul__(y)`;
- `x *= y` — умножение и присваивание — `x.__imul__(y)`;
- `x / y` — деление — `x.__div__(y)`;
- `y / x` — деление (экземпляр класса справа) — `x.__rdiv__(y)`;
- `x /= y` — деление и присваивание — `x.__idiv__(y)`;
- `x // y` — деление с округлением вниз — `x.__floordiv__(y)`;
- `y // x` — деление с округлением вниз (экземпляр класса справа) — `x.__rfloordiv__(y)`;
- `x //= y` — деление с округлением вниз и присваивание — `x.__ifloordiv__(y)`;
- `x % y` — остаток от деления — `x.__mod__(y)`;
- `y % x` — остаток от деления (экземпляр класса справа) — `x.__rmod__(y)`;
- `x %= y` — остаток от деления и присваивание — `x.__imod__(y)`;
- `x ** y` — возведение в степень — `x.__pow__(y)`;
- `y ** x` — возведение в степень (экземпляр класса справа) — `x.__rpow__(y)`;
- `x **= y` — возведение в степень и присваивание — `x.__ipow__(y)`;
- `-x` — унарный - (минус) — `x.__neg__()`;
- `+x` — унарный + (плюс) — `x.__pos__()`;
- `abs(x)` — абсолютное значение — `x.__abs__()`.

Пример перегрузки математических операторов приведен в листинге 13.13.

Листинг 13.13. Пример перегрузки математических операторов

```
class Class1(object):
    def __init__(self):
        self.x = 50
    def __add__(self, y):            # Перегрузка оператора +
        print "Экземпляр слева"
        return self.x + y
    def __radd__(self, y):           # Перегрузка оператора +
        print "Экземпляр справа"
        return self.x + y
    def __iadd__(self, y):           # Перегрузка оператора +=
        print "Сложение с присваиванием"
        return self.x + y
```

```
c1 = Class1()
print c1 + 10                  # Выведет: Экземпляр слева 60
print 20 + c1                  # Выведет: Экземпляр справа 70
c1 += 30                       # Выведет: Сложение с присваиванием
print c1                       # Выведет: 80
```

Методы перегрузки двоичных операторов:

- `~x` — двоичная инверсия — `x.__invert__()`;
- `x & y` — двоичное И — `x.__and__(y)`;
- `y & x` — двоичное И (экземпляр класса справа) — `x.__rand__(y)`;
- `x &= y` — двоичное И и присваивание — `x.__iand__(y)`;
- `x | y` — двоичное ИЛИ — `x.__or__(y)`;
- `y | x` — двоичное ИЛИ (экземпляр класса справа) — `x.__ror__(y)`;
- `x |= y` — двоичное ИЛИ и присваивание — `x.__ior__(y)`;
- `x ^ y` — двоичное исключающее ИЛИ — `x.__xor__(y)`;
- `y ^ x` — двоичное исключающее ИЛИ (экземпляр класса справа) — `x.__rxor__(y)`;
- `x ^= y` — двоичное исключающее ИЛИ и присваивание — `x.__ixor__(y)`;
- `x << y` — сдвиг влево — `x.__lshift__(y)`;
- `y << x` — сдвиг влево (экземпляр класса справа) — `x.__rlshift__(y)`;
- `x <<= y` — сдвиг влево и присваивание — `x.__ilshift__(y)`;
- `x >> y` — сдвиг вправо — `x.__rshift__(y)`;
- `y >> x` — сдвиг вправо (экземпляр класса справа) — `x.__rrshift__(y)`;
- `x >>= y` — сдвиг вправо и присваивание — `x.__irshift__(y)`.

Перегрузка операторов сравнения производится с помощью следующих методов:

- `x == y` — равно — `x.__eq__(y)`;
- `x != y` и `x <> y` — не равно — `x.__ne__(y)`;
- `x < y` — меньше — `x.__lt__(y)`;
- `x > y` — больше — `x.__gt__(y)`;
- `x <= y` — меньше или равно — `x.__le__(y)`;
- `x >= y` — больше или равно — `x.__ge__(y)`;
- `cmp(x, y)` — сравнивает два объекта — `x.__cmp__(y)`;
- `y in x` — проверка на вхождение — `x.__contains__(y)`.

Пример перегрузки операторов сравнения приведен в листинге 13.14.

Листинг 13.14. Пример перегрузки операторов сравнения

```
class Class1(object):
    def __init__(self):
        self.x = 50
        self.arr = [1, 2, 3, 4, 5]
    def __eq__(self, y):           # Перегрузка оператора ==
        return self.x == y
```

```
    def __contains__(self, y):      # Перегрузка оператора in
        return y in self.arr
    def __cmp__(self, y):           # Перегрузка функции cmp()
        if self.x > y: return 1
        elif self.x < y: return -1
        else: return 0
c1 = Class1()
print "Равно" if c1 == 50 else "Не равно" # Выведет: Равно
print "Равно" if c1 == 51 else "Не равно" # Выведет: Не равно
print "Есть" if 5 in c1 else "Нет"        # Выведет: Есть
print cmp(c1, 51)                         # Выведет: -1
print cmp(50, c1)                         # Выведет: 0
```

13.8. Статические методы и методы класса

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса. Для этого перед определением метода внутри класса следует указать декоратор `@staticmethod`. Вызов статического метода без создания экземпляра класса осуществляется следующим образом:

```
<Название класса>.<Название метода>(<Параметры>)
```

Кроме того, можно вызвать статический метод через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Пример использования статических методов приведен в листинге 13.15.

Листинг 13.15. Статические методы

```
class Class1(object):
    @staticmethod
    def sum1(x, y):              # Статический метод
        return x + y
    def sum2(self, x, y):        # Обычный метод в классе
        return x + y
    def sum3(self, x, y):
        return Class1.sum1(x, y) # Вызов из метода класса

print Class1.sum1(10, 20)        # Вызываем статический метод
c1 = Class1()
print c1.sum2(15, 6)             # Вызываем метод класса
print c1.sum1(50, 12)            # Вызываем статический метод
                                 # через экземпляр класса
print c1.sum3(23, 5)             # Вызываем статический метод
                                 # внутри класса
```

Обратите внимание на то, что в определении статического метода нет параметра `self`. Это означает, что внутри статического метода нет доступа к атрибутам и методам экземпляра класса.

Методы класса создаются с помощью декоратора `@classmethod`. В качестве первого параметра в метод класса передается ссылка на класс, а не на экземпляр класса. Вызов метода класса осуществляется следующим образом:

```
<Название класса>.<Название метода>(<Параметры>)
```

Кроме того, можно вызвать метод класса через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

Пример использования методов класса приведен в листинге 13.16.

Листинг 13.16. Методы класса

```
class Class1(object):
    @classmethod
    def test(cls, x): # Метод класса
        print cls, x
Class1.test(10)       # Вызываем метод через название класса
c1 = Class1()
c1.test(50)           # Вызываем метод класса через экземпляр
```

13.9. Абстрактные методы

Абстрактные методы содержат только определение метода без реализации. Предполагается, что класс-потомок должен переопределить метод и реализовать его функциональность. Чтобы такое предположение сделать более очевидным, часто внутри абстрактного метода возбуждают исключение (листинг 13.17).

Листинг 13.17. Абстрактные методы

```
class Class1(object):
    def test(self, x):     # Абстрактный метод
        # Возбуждаем исключение с помощью raise
        raise NotImplementedError("Необходимо переопределить метод")
class Class2(Class1):      # Наследуем абстрактный метод
    def test(self, x):     # Переопределяем метод
        print x
class Class3(Class1):      # Класс не переопределяет метод
    pass

c2 = Class2()
c2.test(50)                # Выведет: 50
c3 = Class3()
```

```
try:                        # Перехватываем исключения
    c3.test(50)             # Ошибка. Метод test() не переопределен
except NotImplementedError, msg:
    print msg               # Выведет: Необходимо переопределить метод
```

Начиная с версии Python 2.6, в состав стандартной библиотеки входит модуль `abc`. В этом модуле определен декоратор `@abstractmethod`, который позволяет указать, что метод, перед которым расположен декоратор, является абстрактным. При попытке создать экземпляр класса-потомка, в котором не переопределен абстрактный метод, возбуждается исключение `TypeError`. Рассмотрим использование декоратора `@abstractmethod` на примере (листинг 13.18).

Листинг 13.18. Использование декоратора `@abstractmethod`

```
from abc import ABCMeta, abstractmethod
class Class1(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def test(self, x):      # Абстрактный метод
        pass
class Class2(Class1):       # Наследуем абстрактный метод
    def test(self, x):      # Переопределяем метод
        print x
class Class3(Class1):       # Класс не переопределяет метод
    pass

c2 = Class2()
c2.test(50)                 # Выведет: 50
try:
    c3 = Class3()           # Ошибка. Метод test() не переопределен
    c3.test(50)
except TypeError, msg:
    print msg               # Can't instantiate abstract class Class3
                            # with abstract methods test
```

13.10. Ограничение доступа к идентификаторам внутри класса

Все идентификаторы внутри класса в языке Python являются открытыми, т. е. доступны для непосредственного изменения. Для имитации частных идентификаторов можно воспользоваться методами `__getattr__()`, `__getattribute__()` и `__setattr__()`, которые перехватывают обращения к атрибутам класса. Кроме того, можно воспользоваться идентификаторами, названия которых начинаются с

двух символов подчеркивания. Такие идентификаторы называются *псевдочастными*. Псевдочастные идентификаторы доступны внутри класса, но не доступны по имени через экземпляр класса. Тем не менее, изменить идентификатор через экземпляр класса все равно можно, зная, каким образом искажается название идентификатора. Например, идентификатор `__privateVar` внутри класса `Class1` будет доступен по имени `_Class1__privateVar`. Как видно из примера, перед идентификатором добавляется название класса с предваряющим символом подчеркивания. Приведем пример использования псевдочастных идентификаторов (листинг 13.19).

Листинг 13.19. Псевдочастные идентификаторы

```
class Class1(object):
    def __init__(self, x):
        self.__privateVar = x
    def setVar(self, x):          # Изменение значения
        self.__privateVar = x
    def getVar(self):             # Получение значения
        return self.__privateVar
c1 = Class1(10)                   # Создаем экземпляр класса
print c1.getVar()                 # Выведет: 10
c1.setVar(20)                     # Изменяем значение
print c1.getVar()                 # Выведет: 20
try:                              # Перехватываем ошибки
    print c1.__privateVar         # Ошибка!!!
except AttributeError, msg:
    print msg                     # Выведет: 'Class1' object has
                                  # no attribute '__privateVar'
c1._Class1__privateVar = 50       # Значение псевдочастных атрибутов
                                  # все равно можно изменить
print c1.getVar()                 # Выведет: 50
```

В классах нового стиля можно ограничить перечень атрибутов, разрешенных для экземпляров класса. Для этого разрешенные атрибуты перечисляются внутри класса в атрибуте `__slots__`. В качестве значения атрибуту можно присвоить строку или список строк с названиями идентификаторов. Если производится попытка обращения к атрибуту, не перечисленному в `__slots__`, то возбуждается исключение `AttributeError` (листинг 13.20).

Листинг 13.20. Атрибут `__slots__`

```
class Class1(object):
    __slots__ = ["x", "y"]
    def __init__(self, a, b):
        self.x, self.y = a, b
c1 = Class1(1, 2)
```

```
print c1.x, c1.y                # Выведет: 1 2
c1.x, c1.y = 10, 20             # Изменяем значения атрибутов
print c1.x, c1.y                # Выведет: 10 20
try:                            # Перехватываем исключения
    c1.z = 50                   # Атрибут z не указан в __slots__
                                # поэтому возбуждается исключение
except AttributeError, msg:
    print msg                   # 'Class1' object has no attribute 'z'
```

13.11. Свойства класса

Классы нового стиля позволяют создать идентификатор, через который можно получить, изменить или удалить значение атрибута класса. Создается такой идентификатор с помощью функции `property()`. Формат функции:

```
<Свойство> = property(<Чтение>[, <Запись>[, <Удаление>
                      [, <Строка документирования>]]])
```

В первых трех параметрах указывается ссылка на соответствующий метод класса. При попытке получить значение будет вызван метод, указанный в первом параметре. При операции присваивания значения будет вызван метод, указанный во втором параметре. Этот метод должен принимать один параметр. В случае удаления атрибута вызывается метод, указанный в третьем параметре. Если в качестве какого-либо параметра задано значение `None`, то это означает, что соответствующий метод не поддерживается. Рассмотрим свойства класса на примере (листинг 13.21).

Листинг 13.21. Свойства класса

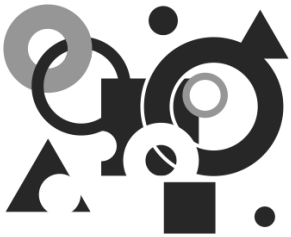
```
class Class1(object):
    def __init__(self, value):
        self.__var = value
    def getVar(self):        # Чтение
        return self.__var
    def setVar(self, value): # Запись
        self.__var = value
    def delVar(self):        # Удаление
        del self.__var
    v = property(getVar, setVar, delVar, "Строка документирования")
c1 = Class1(5)
c1.v = 35                    # Вызывается метод setVar()
print c1.v                   # Вызывается метод getVar()
del c1.v                     # Вызывается метод delVar()
```

В Python 2.6 были добавлены методы `getter()`, `setter()` и `deleter()`, позволяющие создавать свойства классов с помощью декораторов функций. Пример использования декораторов приведен в листинге 13.22.

Листинг 13.22. Методы `getter()`, `setter()` и `deleter()`

```
class Class1(object): # Работает, начиная с версии Python 2.6
    def __init__(self, value):
        self.__var = value
    @property
    def v(self):                        # Чтение
        return self.__var
    @v.setter
    def v(self, value):                 # Запись
        self.__var = value
    @v.deleter
    def v(self):                        # Удаление
        del self.__var
c1 = Class1(5)
c1.v = 35                               # Запись
print c1.v                              # Чтение
del c1.v                                # Удаление
```

Глава 14



Обработка исключений

Исключения — это извещения интерпретатора, возбуждаемые в случае возникновения ошибки в программном коде или при наступлении какого-либо события. Если в коде не предусмотрена обработка исключения, то программа прерывается и выводится сообщение об ошибке.

Существуют три типа ошибок в программе:

- *синтаксические* — это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д., т. е. ошибки в синтаксисе языка. Как правило, интерпретатор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки:

```
>>> print "Нет завершающей кавычки!
SyntaxError: EOL while scanning string literal
```

- *логические* — это ошибки в логике работы программы, которые можно выявить только по результатам работы скрипта. Как правило, интерпретатор не предупреждает о наличии ошибки. А программа будет выполняться, т. к. не содержит синтаксических ошибок. Такие ошибки достаточно трудно выявить и исправить;
- *ошибки времени выполнения* — это ошибки, которые возникают во время работы скрипта. Причиной являются события, не предусмотренные программистом. Классическим примером служит деление на ноль:

```
>>> def test(x, y): return x / y

>>> test(4, 2)                          # Нормально
2
>>> test(4, 0)                          # Ошибка
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    test(4, 0)                          # Ошибка
  File "<pyshell#2>", line 1, in test
    def test(x, y): return x / y
ZeroDivisionError: integer division or modulo by zero
```

Необходимо заметить, что в языке Python исключения возбуждаются не только при ошибке, но и как уведомление о наступлении каких-либо событий. Например, метод `index()` возбуждает исключение `ValueError`, если искомый фрагмент не входит в строку:

```
>>> "Строка".index("текст")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    "Строка".index("текст")
ValueError: substring not found
```

14.1. Инструкция *try...except...else...finally*

Для обработки исключений предназначена инструкция `try`. Формат инструкции:

```
try:
    <Блок, в котором перехватываются исключения>
[except [<Исключение1>[, <Объект исключения>]]:
    <Блок, выполняемый при возникновении исключения>
[...
except [<ИсключениеN>[, <Объект исключения>]]:
    <Блок, выполняемый при возникновении исключения>]]
[else:
    <Блок, выполняемый, если исключение не возникло>]
[finally:
    <Блок, выполняемый в любом случае>]
```

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока `try`. В блоке `except` в параметре `<Исключение1>` указывается класс обрабатываемого исключения. Например, обработать исключение, возникающее при делении на ноль, можно так, как показано в листинге 14.1.

Листинг 14.1. Обработка деления на ноль

```
try:                                # Перехватываем исключения
    x = 1 / 0                       # Ошибка: деление на 0
except ZeroDivisionError:           # Указываем класс исключения
    print "Обработали деление на 0"
    x = 0
print x                             # Выведет: 0
```

Если в блоке `try` возникло исключение, то управление передается блоку `except`. В случае, если исключение не соответствует указанному классу, управление передается следующему блоку `except`. Если ни один блок `except` не соответствует исключению, то исключение "всплывает" к обработчику более высокого уров-

ня. Если исключение нигде не обрабатывается в программе, то управление передается обработчику по умолчанию, который останавливает выполнение программы и выводит стандартную информацию об ошибке. Таким образом, в обработчике может быть несколько блоков `except` с разными классами исключений. Кроме того, один обработчик можно вложить в другой (листинг 14.2).

Листинг 14.2. Вложенные обработчики

```
try:                                    # Обрабатываем исключения
    try:                                # Вложенный обработчик
        x = 1 / 0                       # Ошибка: деление на 0
    except NameError:
        print "Неопределенный идентификатор"
    except IndexError:
        print "Несуществующий индекс"
    print "Выражение после вложенного обработчика"
except ZeroDivisionError:
    print "Обработка деления на 0"
    x = 0
print x                                 # Выведет: 0
```

В этом примере во вложенном обработчике не указано исключение `ZeroDivisionError`, поэтому исключение "всплывает" к обработчику более высокого уровня. После обработки исключения управление передается выражению, расположенному сразу после обработчика. В нашем примере управление будет передано выражению, выводящему значение переменной `x` (`print x`). Обратите внимание на то, что выражение `print "Выражение после вложенного обработчика"` выполнено не будет.

В инструкции `except` можно указать сразу несколько исключений, перечислив их через запятую внутри круглых скобок (листинг 14.3).

Листинг 14.3. Обработка нескольких исключений

```
try:
    x = 1 / 0
except (NameError, IndexError, ZeroDivisionError):
    # Обработка сразу нескольких исключений
    x = 0
print x # Выведет: 0
```

Получить информацию об обрабатываемом исключении можно через второй параметр в инструкции `except` (листинг 14.4).

Листинг 14.4. Получение информации об исключении

```
try:
    x = 1 / 0                       # Ошибка деления на 0
```

```
except (NameError, IndexError, ZeroDivisionError), err:
    print err.__class__.__name__  # Название класса исключения
    print err                     # Текст сообщения об ошибке
```

Результат выполнения:

```
ZeroDivisionError
integer division or modulo by zero
```

Начиная с версии Python 2.6, для разделения параметров в инструкции `except` вместо запятой можно использовать ключевое слово `as` (листинг 14.5).

Листинг 14.5. Использование ключевого слова `as`

```
try:
    x = 1 / 0
except NameError as err:
    print err
except (IndexError, ZeroDivisionError) as err:
    print err
```

Для получения информации об исключении можно воспользоваться функцией `exc_info()` из модуля `sys`, которая возвращает кортеж из трех элементов: типа исключения, значения и объекта с трассировочной информацией. Преобразовать эти значения в удобочитаемый вид позволяет модуль `traceback`. Пример использования функции `exc_info()` и модуля `traceback` приведен в листинге 14.6.

Листинг 14.6. Пример использования функции `exc_info()`

```
import sys, traceback
try:
    x = 1 / 0
except ZeroDivisionError:
    Type, Value, Trace = sys.exc_info()
    print "Type: ", Type
    print "Value:", Value
    print "Trace:", Trace
    print "\n", "print_exception()".center(40, "-")
    traceback.print_exception(Type, Value, Trace, limit=5,
                              file=sys.stdout)
    print "\n", "print_tb()".center(40, "-")
    traceback.print_tb(Trace, limit=1, file=sys.stdout)
    print "\n", "format_exception()".center(40, "-")
    print traceback.format_exception(Type, Value, Trace, limit=5)
    print "\n", "format_exception_only()".center(40, "-")
    print traceback.format_exception_only(Type, Value)
```

Результат выполнения:

```
Type:  <type 'exceptions.ZeroDivisionError'>
Value: integer division or modulo by zero
Trace: <traceback object at 0x00A934E0>

-----------print_exception()------------
Traceback (most recent call last):
  File "C:\book\tests.py", line 4, in <module>
    x = 1 / 0
ZeroDivisionError: integer division or modulo by zero

---------------print_tb()---------------
  File "C:\book\tests.py", line 4, in <module>
    x = 1 / 0

-----------format_exception()-----------
['Traceback (most recent call last):\n', '  File "C:\\book\\tests.py",
line 4, in <module>\n    x = 1 / 0\n', 'ZeroDivisionError: integer
division or modulo by zero\n']

--------format_exception_only()---------
['ZeroDivisionError: integer division or modulo by zero\n']
```

Если в инструкции `except` не указан класс исключения, то такой блок перехватывает все исключения. На практике следует избегать пустых инструкций `except`, т. к. можно перехватить исключение, которое является лишь сигналом системе, а не ошибкой. Пример пустой инструкции `except` приведен в листинге 14.7.

Листинг 14.7. Пример перехвата всех исключений

```
try:
    x = 1 / 0                    # Ошибка деления на 0
except:                          # Обработка всех исключений
    x = 0
print x                          # Выведет: 0
```

Если в обработчике присутствует блок `else`, то выражения внутри этого блока будут выполнены только при отсутствии ошибок. При необходимости выполнить какие-либо завершающие действия вне зависимости от того, возникло исключение или нет, следует воспользоваться блоком `finally`. В качестве примера выведем последовательность выполнения блоков (листинг 14.8).

Листинг 14.8. Блоки `else` и `finally`

```
try:
    x = 10 / 2                   # Нет ошибки
    #x = 10 / 0                  # Ошибка деления на 0
```

```
except ZeroDivisionError:
    print "Деление на 0"
else:
    print "Блок else"
finally:
    print "Блок finally"
```

Результат выполнения при отсутствии исключения:

```
Блок else
Блок finally
```

Последовательность выполнения блоков при наличии исключения будет другой:

```
Деление на 0
Блок finally
```

Необходимо заметить, что при наличии исключения и отсутствии блока `except` выражения внутри блока `finally` будут выполнены, но исключение не будет обработано. Оно продолжит "всплывание" к обработчику более высокого уровня. Если пользовательский обработчик отсутствует, то управление передается обработчику по умолчанию, который прерывает выполнение программы и выводит сообщение об ошибке. Пример:

```
>>> try:
        x = 10 / 0
finally: print "Блок finally"

Блок finally
Traceback (most recent call last):
  File "<pyshell#3>", line 2, in <module>
    x = 10 / 0
ZeroDivisionError: integer division or modulo by zero
```

В качестве примера переделаем нашу программу (листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой (листинг 14.9).

Листинг 14.9. Суммирование неопределенного количества чисел

```
# -*- coding: cp1251 -*-
print "Введите слово 'stop' для получения результата"
summa = 0
while True:
    x = raw_input("Введите число: ")
    if x == "stop":
        break      # Выход из цикла
    try:
        x = int(x) # Преобразуем строку в число
```

```
    except ValueError:
        print "Необходимо ввести целое число!"
    else:
        summa += x
print "Сумма чисел равна:", summa
```

Процесс ввода значений и получения результата выглядит так:

```
Введите слово 'stop' для получения результата
Введите число: 10
Введите число: str
Необходимо ввести целое число!
Введите число: -5
Введите число:
Необходимо ввести целое число!
Введите число: stop
Сумма чисел равна: 5
```

Значения, введенные пользователем, выделены полужирным шрифтом.

14.2. Инструкция *with...as*

Начиная с версии 2.6, язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует выполнение завершающих действий (например, закрытие файла) вне зависимости от того, произошло исключение внутри блока кода или нет. Необходимо заметить, что в Python 2.5 также можно использовать протокол, предварительно указав выражение (в Python 2.6 и выше это выражение указывать не нужно):

```
from __future__ import with_statement
```

Для работы с протоколом предназначена инструкция `with...as`. Инструкция имеет следующий формат:

```
with <Выражение>[ as <Переменная>]:
    <Блок, в котором перехватываем исключения>
```

Вначале вычисляется `<Выражение>`, которое должно возвращать объект, поддерживающий протокол. Этот объект должен иметь два метода: `__enter__()` и `__exit__()`. Метод `__enter__()` вызывается после создания объекта. Значение, возвращаемое этим методом, присваивается переменной, указанной после ключевого слова `as`. Если переменная не указана, возвращаемое значение игнорируется. Формат метода `__enter__()`:

```
__enter__(self)
```

Далее выполняются выражения внутри тела инструкции `with`. Если при выполнении возникло исключение, то управление передается методу `__exit__()`. Метод имеет следующий формат:

```
__exit__(self, <Тип исключения>, <Значение>, <Объект traceback>)
```

Значения, доступные через последние три параметра, полностью эквивалентны значениям, возвращаемым функцией `exc_info()` из модуля `sys`. Если исключение обработано, метод должен вернуть значение `True`, в противном случае — `False`. Если метод возвращает `False`, то исключение передается вышестоящему обработчику.

Если при выполнении выражений, расположенных внутри тела инструкции `with`, исключение не возникло, то управление все равно передается методу `__exit__()`. В этом случае последние три параметра будут содержать значение `None`.

Рассмотрим последовательность выполнения протокола на примере (листинг 14.10).

Листинг 14.10. Протокол менеджеров контекста

```
#from __future__ import with_statement # Для Python 2.5
class Class1(object):
    def __enter__(self):
        print "Вызван метод __enter__()"
        return self
    def __exit__(self, Type, Value, Trace):
        print "Вызван метод __exit__()"
        if Type is None:  # Если исключение не возникло
            print "Исключение не возникло"
        else:             # Если возникло исключение
            print "Value =", Value
            return False  # False - исключение не обработано
                          # True  - исключение обработано
print "Последовательность при отсутствии исключения:"
with Class1():
    print "Блок внутри with"
print "\nПоследовательность при наличии исключения:"
with Class1() as obj:
    print "Блок внутри with"
    raise TypeError("Исключение TypeError")
```

Результат выполнения:

```
Последовательность при отсутствии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
Исключение не возникло

Последовательность при наличии исключения:
Вызван метод __enter__()
Блок внутри with
Вызван метод __exit__()
```

```
Value = Исключение TypeError
Traceback (most recent call last):
  File "C:\test.py", line 21, in <module>
    raise TypeError("Исключение TypeError")
TypeError: Исключение TypeError
```

Некоторые встроенные объекты по умолчанию поддерживают протокол, например, файлы. Если в инструкции `with` указана функция `open()`, то после выполнения инструкций внутри блока файл автоматически будет закрыт. Пример использования инструкции `with` приведен в листинге 14.11.

Листинг 14.11. Инструкция `with...as`

```
#from __future__ import with_statement # Для Python 2.5
with open("test.txt", "a") as f:
    f.write("Строка\n") # Записываем строку в конец файла
```

В этом примере файл test.txt открывается на дозапись в конец файла. После выполнения функции `open()` переменной `f` будет присвоена ссылка на объект файла. С помощью этой переменной мы можем работать с файлом внутри тела инструкции `with`. После выхода из блока, вне зависимости от наличия исключения, файл будет закрыт.

14.3. Классы встроенных исключений

Все встроенные исключения в языке Python представлены в виде классов. Иерархия встроенных классов исключений показана в листинге 14.12.

Листинг 14.12. Иерархия встроенных классов исключений

```
BaseException
    GeneratorExit (в Python 2.6 и выше)
    KeyboardInterrupt
    SystemExit
    Exception
        GeneratorExit (в Python 2.5)
        StopIteration
        Warning
            BytesWarning (в Python 2.6 и выше)
            DeprecationWarning, FutureWarning, ImportWarning
            PendingDeprecationWarning, RuntimeWarning, SyntaxWarning
            UnicodeWarning, UserWarning
        StandardError
            ArithmeticError
                FloatingPointError, OverflowError, ZeroDivisionError
```

```
AssertionError
AttributeError
BufferError (в Python 2.6)
EnvironmentError
    IOError
    OSError
        WindowsError
EOFError
ImportError
LookupError
    IndexError, KeyError
MemoryError
NameError
    UnboundLocalError
ReferenceError
RuntimeError
    NotImplementedError
SyntaxError
    IndentationError
        TabError
SystemError
TypeError
ValueError
    UnicodeError
        UnicodeDecodeError, UnicodeEncodeError
        UnicodeTranslateError
```

Основное преимущество использования классов для обработки исключений заключается в возможности указания базового класса для перехвата всех исключений соответствующих классов-потомков. Например, для перехвата деления на ноль мы использовали класс `ZeroDivisionError`. Если вместо этого класса указать базовый класс `ArithmeticError`, то будут перехватываться исключения классов `FloatingPointError`, `OverflowError` и `ZeroDivisionError`. Пример:

```
try:
    x = 1 / 0                       # Ошибка: деление на 0
except ArithmeticError:             # Указываем базовый класс
    print "Обработали деление на 0"
```

Рассмотрим основные классы встроенных исключений:

- `BaseException` — начиная с Python 2.5, является классом самого верхнего уровня;
- `Exception` — именно этот класс, а не `BaseException`, необходимо наследовать при создании пользовательских классов исключений;
- `AssertionError` — возбуждается инструкцией `assert`;
- `AttributeError` — попытка обращения к несуществующему атрибуту объекта;

- `EOFError` — возбуждается функциями `input()` и `raw_input()` при достижении конца файла;
- `IOError` — ошибка доступа к файлу;
- `ImportError` — невозможно подключить модуль или пакет;
- `IndentationError` — неправильно расставлены отступы в программе;
- `IndexError` — указанный индекс не существует в последовательности;
- `KeyError` — указанный ключ не существует в словаре;
- `KeyboardInterrupt` — нажата комбинация клавиш <Ctrl>+<C>;
- `NameError` — попытка обращения к идентификатору до его определения;
- `StopIteration` — возбуждается методом `next()` как сигнал об окончании итераций;
- `SyntaxError` — синтаксическая ошибка;
- `TypeError` — тип объекта не соответствует ожидаемому;
- `UnboundLocalError` — внутри функции переменной присваивается значение после обращения к одноименной глобальной переменной;
- `UnicodeDecodeError` — ошибка преобразования обычной строки в Unicode-строку;
- `UnicodeEncodeError` — ошибка преобразования Unicode-строки в обычную строку;
- `ValueError` — переданный параметр не соответствует ожидаемому значению;
- `ZeroDivisionError` — попытка деления на ноль.

14.4. Пользовательские исключения

Для возбуждения пользовательских исключений предназначены две инструкции:

- `raise` — возбуждает указанное исключение. Инструкция имеет несколько форматов:

```
raise <Экземпляр класса>
raise <Название класса>[, <Данные>]
raise <Название класса>, <Экземпляр класса>
raise
```

В первом формате инструкции `raise` указывается экземпляр класса возбуждаемого исключения. При создании экземпляра можно передать данные конструктору класса. Эти данные будут доступны через второй параметр в инструкции `except`. Пример возбуждения встроенного исключения `ValueError`:

```
>>> raise ValueError("Описание исключения")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise ValueError("Описание исключения")
ValueError: Описание исключения
```

Пример обработки этого исключения:

```
try:
    raise ValueError("Описание исключения")
except ValueError, msg:
    print msg # Выведет: Описание исключения
```

В качестве исключения можно указать экземпляр пользовательского класса:

```
class MyError(Exception):
    def __init__(self, value):
        self.msg = value
    def __str__(self):
        return self.msg
# Обработка пользовательского исключения
try:
    raise MyError("Описание исключения")
except MyError, err:
    print err        # Вызывается метод __str__()
    print err.msg    # Обращение к атрибуту класса
# Повторно возбуждаем исключение
raise MyError("Описание исключения")
```

Результат выполнения:

```
Описание исключения
Описание исключения
Traceback (most recent call last):
  File "C:\test.py", line 14, in <module>
    raise MyError("Описание исключения")
MyError: Описание исключения
```

Класс `Exception` содержит все необходимые методы для вывода сообщения об ошибке. Поэтому в большинстве случаев достаточно создать пустой класс, который наследует класс `Exception`:

```
class MyError(Exception): pass
try:
    raise MyError("Описание исключения")
except MyError, err:
    print err        # Выведет: Описание исключения
```

Во втором формате инструкции `raise` в первом параметре задается объект класса, а не экземпляр. Значения, указанные в параметре `<Данные>`, передаются конструктору класса. Если необходимо передать несколько значений, то они указываются внутри кортежа. Пример:

```
try:
    raise ValueError # Эквивалентно: raise ValueError()
except ValueError:
    print "Сообщение об ошибке"
```

```
try:
    raise ValueError, "Сообщение об ошибке"
    # Эквивалентно: raise ValueError("Сообщение об ошибке")
except ValueError, err:
    print err      # Выведет: Сообщение об ошибке
class MyError(Exception):
    def __init__(self, value1, value2):
        self.arg1 = value1
        self.arg2 = value2
try:
    raise MyError, ("Значение1", "Значение2")
    # Эквивалентно: raise MyError("Значение1", "Значение2")
except MyError, err:
    print err.arg1 # Выведет: Значение1
    print err.arg2 # Выведет: Значение2
```

В третьем формате инструкции `raise` в первом параметре задается объект класса, а во втором параметре указывается экземпляр этого или производного от него класса. В этом случае экземпляр класса используется для возбуждения исключения. Пример:

```
try:
    raise ValueError, ValueError("Сообщение об ошибке")
except ValueError, err:
    print err      # Выведет: Сообщение об ошибке
```

Четвертый формат инструкции `raise` позволяет повторно возбудить последнее исключение. Пример:

```
class MyError(Exception): pass
try:
    raise MyError("Сообщение об ошибке")
except MyError, err:
    print err
    raise              # Повторно возбуждаем исключение
```

Результат выполнения:

```
Сообщение об ошибке
Traceback (most recent call last):
  File "C:\test.py", line 4, in <module>
    raise MyError("Сообщение об ошибке")
MyError: Сообщение об ошибке
```

- `assert` — возбуждает исключение `AssertionError`, если логическое выражение возвращает значение `False`. Инструкция имеет следующий формат:

```
assert <Логическое выражение>[, <Данные>]
```

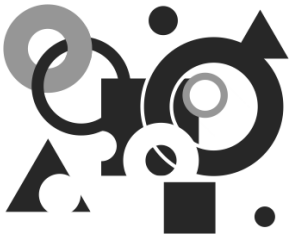
Инструкция `assert` эквивалентна следующему коду:

```
if __debug__:
    if not <Логическое выражение>:
        raise AssertionError, <Данные>
```

Если при запуске программы используется флаг `-O`, то переменная `__debug__` будет иметь ложное значение. Таким образом можно удалить все инструкции `assert` из байт-кода. Пример использования инструкции `assert`:

```
try:
    x = -3
    assert x >= 0, "Сообщение об ошибке"
except AssertionError, err:
    print err # Выведет: Сообщение об ошибке
```

Глава 15



Работа с файлами и каталогами

Очень часто нужно сохранить какие-либо данные. Для этого существуют два способа: сохранение в файл и сохранение в базу данных. Первый способ используется при сохранении информации небольшого объема. Если объем велик, то лучше (и удобнее) воспользоваться базой данных.

15.1. Открытие файла

Прежде чем работать с файлом, необходимо создать объект файла с помощью функции `open()` или функции `file()`, которая является конструктором класса. Из этих двух функций предпочтительнее использовать функцию `open()`. Функции имеют следующий формат:

```
open(<Путь к файлу>[, <Режим>[, <Размер буфера>]])
file(<Путь к файлу>[, <Режим>[, <Размер буфера>]])
```

В первом параметре указывается путь к файлу. Путь может быть абсолютным или относительным. При указании абсолютного пути в Windows следует учитывать, что слеш является специальным символом. По этой причине слеш необходимо удваивать или вместо обычных строк использовать неформатированные строки. Пример:

```
>>> "C:\\temp\\new\\file.txt"    # Правильно
'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt"      # Правильно
'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt"       # Неправильно!!!
'C:\temp\new\x0cile.txt'
```

Обратите внимание на последний пример. В этом пути присутствуют сразу три специальных символа: `\t`, `\n` и `\f`. После преобразования специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Если такую строку передать в функцию `open()`, то это приведет к исключению `IOError`:

```
>>> open("C:\temp\new\file.txt")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    open("C:\temp\new\file.txt")
IOError: [Errno 22] invalid mode ('r') or filename:
'C:\temp\new\x0cile.txt'
```

Вместо абсолютного пути к файлу можно указать относительный путь. В этом случае путь определяется с учетом местоположения текущего рабочего каталога. Относительный путь будет автоматически преобразован в абсолютный путь с помощью функции `abspath()` из модуля `os.path`. Возможны следующие варианты:

- если открываемый файл находится в текущем рабочем каталоге, то можно указать только название файла. Пример:

```
>>> import os.path # Подключаем модуль
>>> # Файл в текущем рабочем каталоге (C:\book\)
>>> os.path.abspath(r"file.txt")
'C:\\book\\file.txt'
```

- если открываемый файл расположен во вложенной папке, то перед названием файла перечисляются названия вложенных папок через слеш. Пример:

```
>>> # Открываемый файл в C:\book\folder1\
>>> os.path.abspath(r"folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> # Открываемый файл в C:\book\folder1\folder2\
>>> os.path.abspath(r"folder1/folder2/file.txt")
'C:\\book\\folder1\\folder2\\file.txt'
```

- если папка с файлом расположена ниже уровнем, то перед названием файла указываются две точки и слеш (`"../"`). Пример:

```
>>> # Открываемый файл в C:\
>>> os.path.abspath(r"../file.txt")
'C:\\file.txt'
```

- если в начале пути расположен слеш, то путь отсчитывается от корня диска. В этом случае местоположение текущего рабочего каталога не имеет значения. Пример:

```
>>> # Открываемый файл в C:\book\folder1\
>>> os.path.abspath(r"/book/folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> # Открываемый файл в C:\book\folder1\folder2\
>>> os.path.abspath(r"/book/folder1/folder2/file.txt")
'C:\\book\\folder1\\folder2\\file.txt'
```

В абсолютном и относительном пути можно указать как прямые, так и обратные слеши. Все слеши будут автоматически преобразованы с учетом значения ат-

рибута `sep` из модуля `os.path`. Значение этого атрибута зависит от используемой операционной системы. Выведем значение в операционной системе Windows:

```
>>> os.path.sep
'\\'
>>> os.path.abspath(r"C:/book/folder1/file.txt")
'C:\\book\\folder1\\file.txt'
```

При использовании относительного пути необходимо учитывать местоположение текущего рабочего каталога, т. к. рабочий каталог не всегда совпадает с каталогом, в котором находится исполняемый файл. Если файл запускается с помощью двойного щелчка на его значке, то каталоги будут совпадать. Если же файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл. Рассмотрим все на примере. В каталоге C:\book создадим следующую структуру файлов:

```
C:\book\
    test.py
    folder1\
        __init__.py
        module1.py
```

Содержимое файла C:\book\test.py приведено в листинге 15.1.

Листинг 15.1. Содержимое файла C:\book\test.py

```
# -*- coding: cp866 -*-
import os, sys
print "%-25s%s" % ("Файл:", __file__)
print "%-25s%s" % ("Текущий рабочий каталог:", os.getcwd())
print "%-25s%s" % ("Каталог для импорта:", sys.path[0])
print "%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt"))
print "-" * 40
import folder1.module1 as m
m.f_getcwd()
```

Файл C:\book\folder1__init__.py создаем пустым. Как вы уже знаете, этот файл позволяет преобразовать каталог в пакет с модулями. Содержимое файла C:\book\folder1\module1.py приведено в листинге 15.2.

Листинг 15.2. Содержимое файла C:\book\folder1\module1.py

```
# -*- coding: cp866 -*-
import os, sys
def f_getcwd():
    print "%-25s%s" % ("Файл:", __file__)
    print "%-25s%s" % ("Текущий рабочий каталог:", os.getcwd())
    print "%-25s%s" % ("Каталог для импорта:", sys.path[0])
    print "%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt"))
```

Запускаем командную строку, переходим в каталог C:\book и запускаем файл test.py:

```
C:\>cd C:\book
C:\book>test.py
Файл:                       C:\book\test.py
Текущий рабочий каталог: C:\book
Каталог для импорта:     C:\book
Путь к файлу:             C:\book\file.txt
----------------------------------------
Файл:                       C:\book\folder1\module1.py
Текущий рабочий каталог: C:\book
Каталог для импорта:     C:\book
Путь к файлу:             C:\book\file.txt
```

В этом примере текущий рабочий каталог совпадает с каталогом, в котором расположен файл test.py. Однако обратите внимание на текущий рабочий каталог внутри модуля module1.py. Если внутри этого модуля в функции `open()` указать название файла без пути, то поиск файла будет произведен в каталоге C:\book, а не C:\book\folder1.

Теперь перейдем в корень диска C: и опять запустим файл test.py:

```
C:\book>cd C:\
C:\>C:\book\test.py
Файл:                       C:\book\test.py
Текущий рабочий каталог: C:\
Каталог для импорта:     C:\book
Путь к файлу:             C:\file.txt
----------------------------------------
Файл:                       C:\book\folder1\module1.py
Текущий рабочий каталог: C:\
Каталог для импорта:     C:\book
Путь к файлу:             C:\file.txt
```

В этом случае текущий рабочий каталог не совпадает с каталогом, в котором расположен файл test.py. Если внутри файлов test.py и module1.py в функции `open()` указать название файла без пути, то поиск файла будет производиться в корне диска C:, а не в каталогах с этими файлами.

Чтобы поиск файла всегда производился в каталоге с исполняемым файлом, необходимо этот каталог сделать текущим с помощью функции `chdir()` из модуля `os`. В качестве примера изменим содержимое файла test.py (листинг 15.3).

Листинг 15.3. Пример использования функции `chdir()`

```
# -*- coding: cp866 -*-
import os, sys
# Делаем каталог с исполняемым файлом текущим
os.chdir(os.path.dirname(__file__))
```

```
print "%-25s%s" % ("Файл:", __file__)
print "%-25s%s" % ("Текущий рабочий каталог:", os.getcwd())
print "%-25s%s" % ("Каталог для импорта:", sys.path[0])
print "%-25s%s" % ("Путь к файлу:", os.path.abspath("file.txt"))
```

Обратите внимание на четвертую строку. С помощью атрибута `__file__` мы получаем полный путь к исполняемому файлу, вместе с названием файла. Далее извлекаем путь (без названия файла) с помощью функции `dirname()` и передаем его функции `chdir()`. Теперь, если в функции `open()` указать название файла без пути, то поиск будет производиться в каталоге с этим файлом. Запустим файл test.py с помощью командной строки:

```
C:\>C:\book\test.py
Файл:                      C:\book\test.py
Текущий рабочий каталог: C:\book
Каталог для импорта:       C:\book
Путь к файлу:              C:\book\file.txt
```

Функции, предназначенные для работы с каталогами, мы еще рассмотрим подробно в следующих разделах. Сейчас важно запомнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог, в котором расположен исполняемый файл. Кроме того, пути поиска файлов не имеют никакого отношения к путям поиска модулей.

Необязательный параметр `<Режим>` в функции `open()` может принимать следующие значения:

- `r` — только чтение (значение по умолчанию). После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение `IOError`;
- `r+` — чтение и запись. После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение `IOError`;
- `w` — запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `w+` — чтение и запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `a` — запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется;
- `a+` — чтение и запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется.

Кроме того, после режима может следовать модификатор:

- `b` — файл будет открыт в бинарном режиме;
- `t` — файл будет открыт в текстовом режиме (значение по умолчанию в Windows). В этом режиме в Windows при чтении символ `\r` будет удален, а при записи, наоборот, добавлен.

В качестве примера создадим файл file.txt и запишем в него две строки:

```
>>> f = open(r"file.txt", "w")  # Открываем файл на запись
>>> f.write("String1\nString2") # Записываем две строки в файл
>>> f.close()                   # Закрываем файл
```

Так как мы указали режим `w`, если файл не существует, то он будет создан, а если существует, то файл будет перезаписан. Теперь выведем содержимое файла в бинарном и текстовом режимах:

```
>>> # Бинарный режим (символ \r остается)
>>> for line in open(r"file.txt", "rb"): print repr(line)

'String1\r\n'
'String2'
>>> # Текстовый режим (символ \r удаляется)
>>> for line in open(r"file.txt", "r"): print repr(line)

'String1\n'
'String2'
```

Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла. В необязательном параметре `<Размер буфера>` можно указать размер буфера. Если в качестве значения указан `0`, то данные будут сразу записываться в файл. Значение `1` используется при построчной записи в файл, другое положительное число задает примерный размер буфера, а отрицательное значение (или отсутствие значения) означает установку размера, применяемого в системе по умолчанию.

15.2. Методы для работы с файлами

После открытия файла функция `open()` возвращает объект, с помощью которого производится дальнейшая работа с файлом. Этот объект поддерживает следующие методы:

- `close()` — закрывает файл. Так как интерпретатор автоматически удаляет объект, когда отсутствуют ссылки на него, можно явно не закрывать файл в небольших программах. Тем не менее, явное закрытие файла является признаком хорошего стиля программирования. Если ссылку на файл нигде не сохранять, то объект будет удален сразу. Таким образом, записать строку в файл и сразу неявно закрыть его можно так:

  ```
  >>> open(r"file.txt", "w").write("Строка")
  ```

  Начиная с версии 2.6, язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует закрытие файла вне зависимости от того, произошло исключение внутри блока кода или нет. Пример:

  ```
  #from __future__ import with_statement # Для Python 2.5
  with open(r"file.txt", "w") as f:
      f.write("Строка") # Записываем строку в файл
  ```

- `write(<Строка>)` — записывает обычную строку в файл. Если в качестве параметра указана Unicode-строка, то производится попытка преобразовать ее в обычную строку. Так как по умолчанию используется кодировка ASCII, попытка преобразовать Unicode-строку (содержащую русские буквы) в обычную строку приведет к исключению `UnicodeEncodeError`. Пример записи в файл:

```
>>> f = open(r"file.txt", "w")  # Открываем файл на запись
>>> f.write("Строка1\nСтрока2") # Записываем строку в файл
>>> f.close()                   # Закрываем файл
```

- `writelines(<Последовательность>)` — записывает последовательность в файл. Все элементы последовательности должны быть строками. Пример записи элементов списка и кортежа:

```
>>> f = open(r"file.txt", "w")
>>> f.writelines(["String1\n", "String2\n"]) # Список
>>> f.writelines(("String3\n", "String4"))   # Кортеж
>>> f.close()
```

- `read([<Количество байт>])` — считывает данные из файла. Если параметр не указан, то возвращается содержимое файла от текущей позиции указателя до конца файла:

```
>>> open(r"file.txt", "r").read()
'String1\nString2\nString3\nString4'
```

  Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество байтов. Когда достигается конец файла, метод возвращает пустую строку. Пример:

```
>>> f = open(r"file.txt", "r")
>>> f.read(20)               # Считываем 20 байт
'String1\nString2\nStri'
>>> f.read(20)               # Считываем 20 байт
'ng3\nString4'
>>> f.read(20)               # Достигнут конец файла
''
>>> f.close()
```

- `readline([<Количество байт>])` — считывает из файла одну строку при каждом вызове. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. При достижении конца файла возвращается пустая строка. Пример:

```
>>> f = open(r"file.txt", "r")
>>> f.readline(), f.readline(), f.readline(), f.readline()
('String1\n', 'String2\n', 'String3\n', 'String4')
>>> f.readline() # Достигнут конец файла
''
>>> f.close()
```

Если в необязательном параметре указано число, то считывание будет выполняться до тех пор, пока не встретится символ новой строки (\n), символ конца файла или из файла не будет прочитано указанное количество байтов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество байтов. Если количество символов в строке больше, то возвращается указанное количество байтов. Пример:

```
>>> f = open(r"file.txt", "r")
>>> f.readline(5), f.readline(5)
('Strin', 'g1\n')
>>> f.readline(100) # Возвращается одна строка, а не 100 байт
'String2\n'
>>> f.close()
```

- `readlines()` — считывает все содержимое файла в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. Пример:

```
>>> open(r"file.txt", "r").readlines()
['String1\n', 'String2\n', 'String3\n', 'String4']
```

- `xreadlines()` — возвращает итератор, с помощью которого можно построчно считывать файл. При достижении конца файла возбуждается исключение `StopIteration`. Пример:

```
>>> f = open(r"file.txt", "r")
>>> i = f.xreadlines()
>>> i.next(), i.next(), i.next(), i.next()
('String1\n', 'String2\n', 'String3\n', 'String4')
>>> i.next() # Достигнут конец файла
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    i.next() # Достигнут конец файла
StopIteration
>>> f.close()
```

  В современных версиях Python файлы напрямую поддерживают итерационный протокол, поэтому вместо метода `xreadlines()` лучше использовать файловый метод `next()`;

- `next()` — считывает одну строку при каждом вызове. При достижении конца файла возбуждается исключение `StopIteration`. Пример:

```
>>> f = open(r"file.txt", "r")
>>> f.next(), f.next(), f.next(), f.next()
('String1\n', 'String2\n', 'String3\n', 'String4')
>>> f.next() # Достигнут конец файла
Traceback (most recent call last):
  File "<pyshell#87>", line 1, in <module>
    f.next() # Достигнут конец файла
StopIteration
>>> f.close()
```

Благодаря методу `next()` мы можем перебирать файл построчно с помощью цикла `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `next()`. В качестве примера выведем все строки, предварительно удалив символ перевода строки:

```
>>> f = open(r"file.txt", "r")
>>> for line in f: print line.rstrip(),

String1 String2 String3 String4
>>> f.close()
```

Если после перебора файла он больше не нужен, то функцию `open()` можно указать в цикле `for`. В этом случае файл будет закрыт автоматически после выхода из цикла. Пример:

```
>>> for line in open(r"file.txt", "r"): print line.rstrip(),

String1 String2 String3 String4
```

- `flush()` — записывает данные из буфера на диск;
- `fileno()` — возвращает целочисленный дескриптор файла. Возвращаемое значение всегда будет больше числа 2, т. к. число 0 закреплено за стандартным вводом `stdin`, 1 — за стандартным выводом `stdout`, а 2 — за стандартным выводом сообщений об ошибках `stderr`. Пример:

```
>>> f = open(r"file.txt", "r")
>>> f.fileno()                    # Дескриптор файла
3
>>> f.close()
```

- `truncate([<Количество байт>])` — обрезает файл до указанного количества байтов. Пример:

```
>>> f = open(r"file.txt", "r+")
>>> f.read()
'String1\nString2\nString3\nString4'
>>> f.truncate(25)
>>> f.close()
>>> open(r"file.txt", "r").read()
'String1\nString2\nString3'
```

Если параметр не указан, то файл обрезается до текущей позиции указателя файла:

```
>>> f = open(r"file.txt", "r+")
>>> f.readline(), f.readline()
('String1\n', 'String2\n')
>>> f.truncate()
>>> f.close()
>>> open(r"file.txt", "r").read()
'String1\nString2\n'
```

- `tell()` — возвращает позицию указателя относительно начала файла в виде длинного целого числа. Обратите внимание на то, что в Windows метод `tell()` считает символ `\r` как дополнительный байт, хотя этот символ удаляется при открытии файла в текстовом режиме. Пример:

```
>>> f = open(r"file.txt", "r")
>>> f.tell()      # Указатель расположен в начале файла
0L
>>> f.readline()  # Перемещаем указатель
'String1\n'
>>> f.tell()      # Возвращает 9 (8 + '\r'), а не 8 !!!
9L
>>> f.close()
```

  Чтобы избежать этого несоответствия, следует открывать файл в бинарном режиме, а не текстовом:

```
>>> f = open(r"file.txt", "rb")
>>> f.readline()  # Перемещаем указатель
'String1\r\n'
>>> f.tell()      # Теперь значение соответствует
9L
>>> f.close()
```

- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель в позицию, имеющую смещение `<Смещение>` относительно позиции `<Позиция>`. В параметре `<Позиция>` могут быть указаны следующие атрибуты из модуля `os` или соответствующие им значения:
  - `os.SEEK_SET` или `0` — начало файла (значение по умолчанию);
  - `os.SEEK_CUR` или `1` — текущая позиция указателя;
  - `os.SEEK_END` или `2` — конец файла.

  Выведем значения этих атрибутов:

```
>>> import os
>>> os.SEEK_SET, os.SEEK_CUR, os.SEEK_END
(0, 1, 2)
```

  Пример использования метода `seek()`:

```
>>> import os
>>> f = open(r"file.txt", "rb")
>>> f.seek(9, os.SEEK_CUR) # 9 байт от указателя
>>> f.tell()
9L
>>> f.seek(0, os.SEEK_SET) # Перемещаем указатель в начало
>>> f.tell()
0L
>>> f.seek(-9, os.SEEK_END) # -9 байт от конца файла
>>> f.tell()
9L
>>> f.close()
```

Помимо методов объекты файлов поддерживают несколько атрибутов:

- `name` — содержит название файла;
- `mode` — режим, в котором был открыт файл;
- `closed` — возвращает `True`, если файл был закрыт, и `False` в противном случае.

Пример:

```
>>> f = open(r"file.txt", "r+b")
>>> f.name, f.mode, f.closed
('file.txt', 'r+b', False)
>>> f.close()
>>> f.closed
True
```

- `encoding` — название кодировки, которая будет использоваться для преобразования Unicode-строк перед записью в файл. Если атрибут содержит значение `None`, то будет использована кодировка, заданная в настройках системы. Так как значение в настройках системы равно `"ascii"`, попытка записи Unicode-строки с русскими буквами приведет к исключению `UnicodeEncodeError`. Обратите внимание на то, что изменить значение атрибута нельзя, т. к. атрибут доступен только для чтения. Перед записью в файл лучше явно преобразовывать Unicode-строку в обычную строку. Пример:

```
>>> f = open(r"file.txt", "a")
>>> print f.encoding
None
>>> f.write(unicode("Строка", "cp1251"))
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    f.write(unicode("Строка", "cp1251"))
UnicodeEncodeError: 'ascii' codec can't encode characters in
position 0-5: ordinal not in range(128)
>>> f.close()
```

Стандартный вывод `stdout` также является файловым объектом. Атрибут `encoding` этого объекта всегда содержит кодировку устройства вывода, поэтому Unicode-строка преобразуется в обычную строку в правильной кодировке. Например, при запуске с помощью двойного щелчка на значке файла атрибут `encoding` будет иметь значение `"cp866"`, а при запуске в окне **Python Shell** редактора IDLE — значение `"cp1251"`. Пример:

```
>>> import sys
>>> sys.stdout.encoding
'cp1251'
>>> sys.stdout.write(unicode("Строка", "cp1251"))
Строка
```

15.3. Доступ к файлам с помощью модуля *os*

Модуль `os` содержит дополнительные низкоуровневые функции, позволяющие работать с файлами. Функциональность этого модуля зависит от используемой операционной системы. Получить название используемой версии модуля можно с помощью атрибута `name`. В операционной системе Windows XP атрибут имеет значение `"nt"`:

```
>>> import os
>>> os.name                # Значение в ОС Windows XP
'nt'
```

Для доступа к файлам предназначены следующие функции из модуля `os`:

- `open(<Путь к файлу>, <Режим>[, mode=0777])` — открывает файл и возвращает целочисленный дескриптор, с помощью которого производится дальнейшая работа с файлом. Если файл открыть не удалось, возбуждается исключение `OSError`. В параметре `<Режим>` в операционной системе Windows могут быть указаны следующие флаги (или их комбинация через символ `|`):
  - `os.O_RDONLY` — чтение;
  - `os.O_WRONLY` — запись;
  - `os.O_RDWR` — чтение и запись;
  - `os.O_APPEND` — добавление в конец файла;
  - `os.O_CREAT` — создать файл, если он не существует;
  - `os.O_TRUNC` — очистить содержимое файла;
  - `os.O_BINARY` — файл будет открыт в бинарном режиме;
  - `os.O_TEXT` — файл будет открыт в текстовом режиме. В Windows файлы по умолчанию открываются в текстовом режиме.

  Рассмотрим несколько примеров. Откроем файл на запись и запишем в него одну строку. Если файл не существует, то создадим его. Если файл существует, то очистим его:

```
>>> import os                # Подключаем модуль
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_TRUNC
>>> f = os.open(r"file.txt", mode)
>>> os.write(f, "String1\n") # Записываем строку
8
>>> os.close(f)             # Закрываем файл
```

  Добавим еще одну строку в конец файла:

```
>>> mode = os.O_WRONLY | os.O_CREAT | os.O_APPEND
>>> f = os.open(r"file.txt", mode)
>>> os.write(f, "String2\n") # Записываем строку
8
>>> os.close(f)             # Закрываем файл
```

Прочитаем содержимое файла в текстовом режиме:

```
>>> f = os.open(r"file.txt", os.O_RDONLY)
>>> os.read(f, 50)           # Читаем 50 байт
'String1\nString2\n'
>>> os.close(f)              # Закрываем файл
```

Теперь прочитаем содержимое файла в бинарном режиме:

```
>>> f = os.open(r"file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.read(f, 50)           # Читаем 50 байт
'String1\r\nString2\r\n'
>>> os.close(f)              # Закрываем файл
```

- `read(<Дескриптор>, <Количество байтов>)` — читает из файла указанное количество байтов. При достижении конца файла возвращается пустая строка. Пример:

```
>>> f = os.open(r"file.txt", os.O_RDONLY)
>>> os.read(f, 5), os.read(f, 5), os.read(f, 5), os.read(f, 5)
('Strin', 'g1\nS', 'tring', '2\n')
>>> os.read(f, 5)            # Достигнут конец файла
''
>>> os.close(f)              # Закрываем файл
```

- `write(<Дескриптор>, <Строка>)` — записывает строку в файл. Возвращает количество записанных байтов;
- `close(<Дескриптор>)` — закрывает файл;
- `lseek(<Дескриптор>, <Смещение>, <Позиция>)` — устанавливает указатель в позицию, имеющую смещение `<Смещение>` относительно позиции `<Позиция>`. В качестве значения функция возвращает новую позицию указателя. В параметре `<Позиция>` могут быть указаны следующие атрибуты или соответствующие им значения:
  - `os.SEEK_SET` или `0` — начало файла;
  - `os.SEEK_CUR` или `1` — текущая позиция указателя;
  - `os.SEEK_END` или `2` — конец файла.

  Пример:

```
>>> f = os.open(r"file.txt", os.O_RDONLY | os.O_BINARY)
>>> os.lseek(f, 0, os.SEEK_END) # Перемещение в конец файла
18L
>>> os.lseek(f, 0, os.SEEK_SET) # Перемещение в начало файла
0L
>>> os.lseek(f, 9, os.SEEK_CUR) # Относительно указателя
9L
>>> os.lseek(f, 0, os.SEEK_CUR) # Текущее положение указателя
9L
>>> os.close(f)                 # Закрываем файл
```

- `dup(<Дескриптор>)` — возвращает дубликат файлового дескриптора;

- `fdopen(<Дескриптор>[, <Режим>[, <Размер буфера>]])` — возвращает файловый объект по указанному дескриптору. Параметры `<Режим>` и `<Размер буфера>` имеют тот же смысл, что и в функции `open()`. Пример:

```
>>> fd = os.open(r"file.txt", os.O_RDONLY)
>>> fd
3
>>> f = os.fdopen(fd, "r")
>>> f.fileno() # Объект имеет тот же дескриптор
3
>>> f.read()
'String1\nString2\n'
>>> f.close()
```

В этом примере мы воспользовались методом `fileno()`, который возвращает целочисленный дескриптор файла. Это значение можно передать функциям модуля `os` и манипулировать файловым объектом средствами этого модуля. Создадим файловый объект, запишем строку в файл, а затем закроем его с помощью функций модуля `os`:

```
>>> f = open(r"file.txt", "a")
>>> os.write(f.fileno(), "String3\n")
8
>>> os.close(f.fileno())
>>> open(r"file.txt").read()
'String1\nString2\nString3\n'
```

15.4. Модуль *StringIO*

Модуль `StringIO` позволяет работать со строкой как с файловым объектом. Все операции с файловым объектом производятся в оперативной памяти. Для создания нового объекта предназначен класс `StringIO`. Формат конструктора класса:

```
StringIO([<Начальное значение>])
```

Если параметр не указан, то начальным значением будет пустая строка. После создания объекта указатель текущей позиции устанавливается на начало "файла". Объект, возвращаемый конструктором класса, имеет следующие методы:

- `close()` — закрывает "файл". Проверить, открыт "файл" или закрыт, позволяет атрибут `closed`. Атрибут возвращает `True`, если "файл" был закрыт, и `False` в противном случае;
- `getvalue()` — возвращает содержимое "файла" в виде строки:

```
>>> import StringIO          # Подключаем модуль
>>> f = StringIO.StringIO("String1\n")
>>> f.getvalue()             # Получаем содержимое файла
'String1\n'
>>> f.close()                # Закрываем файл
```

- `tell()` — возвращает текущую позицию указателя относительно начала "файла";
- `seek(<Смещение>[, <Позиция>])` — устанавливает указатель в позицию, имеющую смещение `<Смещение>` относительно позиции `<Позиция>`. В параметре `<Позиция>` могут быть указаны следующие значения:
  - `0` — начало "файла" (значение по умолчанию);
  - `1` — текущая позиция указателя;
  - `2` — конец "файла".

  Пример использования методов `seek()` и `tell()`:

```
>>> f = StringIO.StringIO("String1\n")
>>> f.tell()        # Позиция указателя
0
>>> f.seek(0, 2)    # Перемещаем указатель в конец файла
>>> f.tell()        # Позиция указателя
8
>>> f.seek(0)       # Перемещаем указатель в начало файла
>>> f.tell()        # Позиция указателя
0
>>> f.close()       # Закрываем файл
```

- `write(<Строка>)` — записывает строку в "файл":

```
>>> f = StringIO.StringIO("String1\n")
>>> f.seek(0, 2)         # Перемещаем указатель в конец файла
>>> f.write("String2\n") # Записываем строку в файл
>>> f.getvalue()         # Получаем содержимое файла
'String1\nString2\n'
>>> f.close()            # Закрываем файл
```

- `writelines(<Последовательность>)` — записывает последовательность в "файл":

```
>>> f = StringIO.StringIO()
>>> f.writelines(["String1\n", "String2\n"])
>>> f.getvalue()         # Получаем содержимое файла
'String1\nString2\n'
>>> f.close()            # Закрываем файл
```

- `read([<Количество байтов>])` — считывает данные из "файла". Если параметр не указан, то возвращается содержимое "файла" от текущей позиции указателя до конца "файла". Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество байтов. Когда достигается конец "файла", метод возвращает пустую строку. Пример:

```
>>> f = StringIO.StringIO("String1\nString2\n")
>>> f.read()
'String1\nString2\n'
>>> f.seek(0) # Перемещаем указатель в начало файла
>>> f.read(5), f.read(5), f.read(5), f.read(5), f.read(5)
('Strin', 'g1\nSt', 'ring2', '\n', '')
>>> f.close() # Закрываем файл
```

- `readline([<Количество байтов>])` — считывает из "файла" одну строку при каждом вызове. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. При достижении конца "файла" возвращается пустая строка. Пример:

```
>>> f = StringIO.StringIO("String1\nString2")
>>> f.readline(), f.readline(), f.readline()
('String1\n', 'String2', '')
>>> f.close() # Закрываем файл
```

  Если в необязательном параметре указано число, то считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца "файла" или из "файла" не будет прочитано указанное количество байтов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество байтов. Если количество символов в строке больше, то возвращается указанное количество байтов. Пример:

```
>>> f = StringIO.StringIO("String1\nString2\nString3\n")
>>> f.readline(5), f.readline(5)
('Strin', 'g1\n')
>>> f.readline(100) # Возвращается одна строка, а не 100 байт
'String2\n'
>>> f.close()       # Закрываем файл
```

- `readlines([<Примерное количество байтов>])` — считывает все содержимое "файла" в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. Пример:

```
>>> f = StringIO.StringIO("String1\nString2\nString3")
>>> f.readlines()
['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем файл
```

  Если в необязательном параметре указано число, то считывается указанное количество байтов плюс фрагмент до символа конца строки `\n`. Затем эта строка разбивается и добавляется построчно в список. Пример:

```
>>> f = StringIO.StringIO("String1\nString2\nString3")
>>> f.readlines(16)
['String1\n', 'String2\n']
>>> f.seek(0) # Перемещаем указатель в начало файла
>>> f.readlines(17)
['String1\n', 'String2\n', 'String3']
>>> f.close() # Закрываем файл
```

- `next()` — считывает одну строку при каждом вызове. При достижении конца "файла" возбуждается исключение `StopIteration`. Пример:

```
>>> f = StringIO.StringIO("String1\nString2")
>>> f.next(), f.next()
('String1\n', 'String2')
>>> f.next()
Traceback (most recent call last):
... Фрагмент опущен ...
StopIteration
>>> f.close() # Закрываем файл
```

  Благодаря методу `next()` мы можем перебирать файл построчно с помощью цикла `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `next()`. Пример:

```
>>> f = StringIO.StringIO("String1\nString2")
>>> for line in f: print line.rstrip()

String1
String2
>>> f.close() # Закрываем файл
```

- `flush()` — сбрасывает данные из буфера в "файл";
- `truncate([<Количество байтов>])` — обрезает "файл" до указанного количества байтов. Пример:

```
>>> f = StringIO.StringIO("String1\nString2\nString3")
>>> f.truncate(15)     # Обрезаем файл
>>> f.getvalue()       # Получаем содержимое файла
'String1\nString2'
>>> f.close()          # Закрываем файл
```

  Если параметр не указан, то "файл" обрезается до текущей позиции указателя:

```
>>> f = StringIO.StringIO("String1\nString2\nString3")
>>> f.seek(15)         # Перемещаем указатель
>>> f.truncate()       # Обрезаем файл до указателя
>>> f.getvalue()       # Получаем содержимое файла
'String1\nString2'
>>> f.close()          # Закрываем файл
```

15.5. Права доступа к файлам и каталогам

В операционной системе семейства UNIX для каждого объекта (файла или каталога) назначаются права доступа для каждой разновидности пользователей — владельца, группы и прочих. Могут быть назначены следующие права доступа:

- чтение;
- запись;
- выполнение.

Права доступа обозначаются буквами:

- `r` — файл можно читать, а содержимое каталога можно просматривать;
- `w` — файл можно модифицировать, удалять и переименовывать, а в каталоге можно создавать или удалять файлы. Каталог можно переименовать или удалить;
- `x` — файл можно выполнять, а в каталоге можно выполнять операции над файлами, в том числе производить поиск файлов в нем.

Права доступа к файлу определяются записью типа:

```
-rw-r--r--
```

Первый символ `-` означает, что это файл, и не задает никаких прав доступа. Далее три символа (`rw-`) задают права доступа для владельца (чтение и запись). Символ `-` означает, что права доступа на выполнение нет. Следующие три символа задают права доступа для группы (`r--`) — только чтение. Ну и последние три символа (`r--`) задают права для всех остальных пользователей (только чтение).

Права доступа к каталогу определяются такой строкой:

```
drwxr-xr-x
```

Первая буква (`d`) означает, что это каталог. Владелец может выполнять в каталоге любые действия (`rwx`), а группа и все остальные пользователи — только читать и выполнять поиск (`r-x`). Для того чтобы каталог можно было просматривать, должны быть установлены права на выполнение (`x`).

Кроме того, права доступа обозначаются числом. Такие числа называются *маской прав доступа*. Число состоит из трех цифр от 0 до 7. Первая цифра задает права для владельца, вторая — для группы, а третья — для всех остальных пользователей. Например, права доступа `-rw-r--r--` соответствуют числу 644. Сопоставим числам, входящим в маску прав доступа, двоичную и буквенную записи (табл. 15.1).

Таблица 15.1. *Права доступа в разных записях*

Восьмеричная цифра	Двоичная запись	Буквенная запись	Восьмеричная цифра	Двоичная запись	Буквенная запись
0	000	`---`	4	100	`r--`
1	001	`--x`	5	101	`r-x`
2	010	`-w-`	6	110	`rw-`
3	011	`-wx`	7	111	`rwx`

Например, права доступа `rw-r--r--` можно записать так: `110 100 100`, что переводится в число `6 4 4`. Таким образом, если право предоставлено, то в соответствующей позиции стоит 1, а если нет — то 0.

Для определения прав доступа к файлу или каталогу предназначена функция `access()` из модуля `os`. Функция имеет следующий формат:

```
access(<Путь>, <Режим>)
```

Функция возвращает `True`, если проверка прошла успешно, или `False` в противном случае. В параметре `<Режим>` могут быть указаны следующие константы, определяющие тип проверки:

- `os.F_OK` — проверка наличия пути или файла:

```
>>> import os                          # Подключаем модуль os
>>> os.access(r"file.txt", os.F_OK) # Файл существует
True
>>> os.access(r"C:\book", os.F_OK)  # Каталог существует
True
>>> os.access(r"C:\book2", os.F_OK) # Каталог не существует
False
```

- `os.R_OK` — проверка на возможность чтения файла или каталога;
- `os.W_OK` — проверка на возможность записи в файл или каталог;
- `os.X_OK` — определение, является ли файл или каталог выполняемым.

Чтобы изменить права доступа из программы, необходимо воспользоваться функцией `chmod()` из модуля `os`. Функция имеет следующий формат:

```
chmod(<Путь>, <Права доступа>)
```

Права доступа задаются в виде числа, перед которым следует указать 0 (это соответствует восьмеричной записи числа):

```
>>> os.chmod(r"file.txt", 0777) # Полный доступ к файлу
```

Вместо числа можно указать комбинацию констант из модуля `stat`. За дополнительной информацией обращайтесь к документации по модулю.

15.6. Функции для манипулирования файлами

Для копирования и перемещения файлов предназначены следующие функции из модуля `shutil`:

- `copyfile(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать содержимое файла в другой файл. Никакие метаданные (например, права доступа) не копируются. Если файл существует, то он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `IOError`. Пример:

```
>>> import shutil      # Подключаем модуль
>>> shutil.copyfile(r"file.txt", r"file2.txt")
>>> # Путь не существует:
>>> shutil.copyfile(r"file.txt", r"C:\book2\file2.txt")
... Фрагмент опущен ...
IOError: [Errno 2] No such file or directory:
'C:\\book2\\file2.txt'
```

- `copy(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать файл. Копируются также права доступа. Если файл существует, то он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `IOError`. Пример:

```
>>> shutil.copy(r"file.txt", r"file3.txt")
```

- `copy2(<Копируемый файл>, <Куда копируем>)` — позволяет скопировать файл вместе с метаданными. Если файл существует, то он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение `IOError`. Пример:

```
>>> shutil.copy2(r"file.txt", r"file4.txt")
```

- `move(<Путь к файлу>, <Куда перемещаем>)` — копирует файл в указанное место, а затем удаляет исходный файл. Если файл существует, то он будет перезаписан. Если файл не удалось переместить, возбуждается исключение `IOError`. Если файл удалить нельзя, то в операционной системе Windows возбуждается исключение `WindowsError`. Пример перемещения файла file4.txt в каталог C:\book\test:

```
>>> shutil.move(r"file4.txt", r"C:\book\test")
```

Для переименования и удаления файлов предназначены следующие функции из модуля `os`:

- `rename(<Старое имя>, <Новое имя>)` — переименовывает файл. Если исходный файл отсутствует или новое имя файла уже существует, то в операционной системе Windows возбуждается исключение `WindowsError`. Пример переименования файла с обработкой исключений:

```
import os # Подключаем модуль
try:
    os.rename(r"file3.txt", "file4.txt")
except OSError: # WindowsError наследует OSError
    print "Файл не удалось переименовать"
else:
    print "Файл успешно переименован"
```

- `remove(<Путь к файлу>)` и `unlink(<Путь к файлу>)` — позволяют удалить файл. Если файл удалить нельзя, то в операционной системе Windows возбуждается исключение `WindowsError`. Пример:

```
>>> os.remove(r"file2.txt")
>>> os.unlink(r"file4.txt")
```

Модуль `os.path` содержит дополнительные функции, позволяющие проверить наличие файла, получить размер файла и др. Перечислим эти функции:

- `exists(<Путь>)` — проверяет указанный путь на существование. Значением функции будет `True`, если путь существует, и `False` в противном случае:

```
>>> import os.path
>>> os.path.exists(r"file.txt"), os.path.exists(r"file2.txt")
(True, False)
>>> os.path.exists(r"C:\book"), os.path.exists(r"C:\book2")
(True, False)
```

- `getsize(<Путь к файлу>)` — возвращает размер файла. Если файл не существует, то в Windows возбуждается исключение `WindowsError`:

```
>>> os.path.getsize(r"file.txt")  # Файл существует
27L
>>> os.path.getsize(r"file2.txt") # Файл не существует
... Фрагмент опущен ...
WindowsError: [Error 2] : 'file2.txt'
```

- `getatime(<Путь к файлу>)` — служит для определения времени последнего доступа к файлу. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример:

```
>>> import time    # Подключаем модуль time
>>> t = os.path.getatime(r"file.txt")
>>> t
1275860670.765625
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'07.06.2010 01:44:30'
```

- `getctime(<Путь к файлу>)` — позволяет узнать дату создания файла. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример:

```
>>> t = os.path.getctime(r"file.txt")
>>> t
1275146194.0
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.05.2010 19:16:34'
```

- `getmtime(<Путь к файлу>)` — возвращает время последнего изменения файла. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример:

```
>>> t = os.path.getmtime(r"file.txt")
>>> t
1275859256.53125
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'07.06.2010 01:20:56'
```

Получить размер файла и время создания, изменения и доступа к файлу, а также значения других метаданных позволяет функция `stat()` из модуля `os`. В качестве значения функция возвращает объект `stat_result`, содержащий десять атрибутов: `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime` и `st_ctime`. Пример использования функции `stat()` приведен в листинге 15.4.

Листинг 15.4. Пример использования функции `stat()`

```
>>> import os, time
>>> s = os.stat(r"file.txt")
>>> s
nt.stat_result(st_mode=33206, st_ino=0L, st_dev=0, st_nlink=0, st_uid=0,
st_gid=0, st_size=27L, st_atime=1275860670L, st_mtime=1275859256L,
st_ctime=1275146194L)
>>> s.st_size      # Размер файла
27L
>>> t = s.st_atime # Последний доступ
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'07.06.2010 01:44:30'
>>> t = s.st_ctime # Создание файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.05.2010 19:16:34'
>>> t = s.st_mtime # Изменение файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'07.06.2010 01:20:56'
```

Обновить время последнего доступа и время изменения файла позволяет функция `utime()` из модуля `os`. Функция имеет два формата:

```
utime(<Путь к файлу>, None)
utime(<Путь к файлу>, (<Последний доступ>, <Изменение файла>))
```

Если в качестве второго параметра указано значение `None`, то время доступа и изменения файла будет текущим. Во втором формате функции `utime()` указывается кортеж из новых значений в виде количества секунд, прошедших с начала эпохи. Если файл не существует, то в Windows возбуждается исключение `WindowsError`. Пример использования функции `utime()` приведен в листинге 15.5.

Листинг 15.5. Пример использования функции `utime()`

```
>>> import os, time
>>> os.stat(r"file.txt")          # Первоначальные значения
nt.stat_result(st_mode=33206, st_ino=0L, st_dev=0, st_nlink=0, st_uid=0,
st_gid=0, st_size=27L, st_atime=1275860670L, st_mtime=1275859256L,
st_ctime=1275146194L)
>>> t = time.time() - 600
>>> os.utime(r"file.txt", (t, t)) # Текущее время минус 600 сек
>>> os.stat(r"file.txt")
nt.stat_result(st_mode=33206, st_ino=0L, st_dev=0, st_nlink=0, st_uid=0,
st_gid=0, st_size=27L, st_atime=1275860811L, st_mtime=1275860811L,
st_ctime=1275146194L)
```

```
>>> os.utime(r"file.txt", None)   # Текущее время
>>> os.stat(r"file.txt")
nt.stat_result(st_mode=33206, st_ino=0L, st_dev=0, st_nlink=0, st_uid=0,
st_gid=0, st_size=27L, st_atime=1275861469L, st_mtime=1275861469L,
st_ctime=1275146194L)
```

15.7. Преобразование пути к файлу или каталогу

Преобразовать путь к файлу или каталогу позволяют следующие функции из модуля `os.path`:

- `abspath(<Относительный путь>)` — преобразует относительный путь в абсолютный, учитывая местоположение текущего рабочего каталога. Пример:

```
>>> import os.path
>>> os.path.abspath(r"file.txt")
'C:\\book\\file.txt'
>>> os.path.abspath(r"folder1/file.txt")
'C:\\book\\folder1\\file.txt'
>>> os.path.abspath(r"../file.txt")
'C:\\file.txt'
```

В относительном пути можно указать как прямые, так и обратные слеши. Все слеши будут автоматически преобразованы с учетом значения атрибута `sep` из модуля `os.path`. Значение этого атрибута зависит от используемой операционной системы. Выведем значение в операционной системе Windows:

```
>>> os.path.sep
'\\'
```

При указании пути в Windows следует учитывать, что слеш является специальным символом. По этой причине слеш необходимо удваивать или вместо обычных строк использовать неформатированные строки. Пример:

```
>>> "C:\\temp\\new\\file.txt"    # Правильно
'C:\\temp\\new\\file.txt'
>>> r"C:\temp\new\file.txt"      # Правильно
'C:\\temp\\new\\file.txt'
>>> "C:\temp\new\file.txt"       # Неправильно!!!
'C:\temp\new\x0cile.txt'
```

Кроме того, если слеш расположен в конце строки, то его необходимо удваивать даже при использовании неформатированных строк:

```
>>> r"C:\temp\new\"              # Неправильно!!!
SyntaxError: EOL while scanning string literal
>>> r"C:\temp\new\\"
'C:\\temp\\new\\\\'
```

В первом случае последний слеш экранирует закрывающую кавычку, что приводит к синтаксической ошибке. Решить эту проблему можно, удвоив последний слеш. Однако посмотрите на результат. Два слеша превратились в четыре. От одной проблемы ушли, а к другой пришли. Поэтому в этом случае лучше использовать обычные строки:

```
>>> "C:\\temp\\new\\"            # Правильно
'C:\\temp\\new\\'
```

- `isabs(<Путь>)` — возвращает `True`, если путь является абсолютным, и `False` в противном случае:

```
>>> os.path.isabs("file.txt")
False
>>> os.path.isabs(r"C:\book\file.txt")
True
```

- `basename(<Путь>)` — возвращает имя файла без пути к нему:

```
>>> os.path.basename(r"C:\book\folder1\file.txt")
'file.txt'
>>> os.path.basename(r"C:\book\folder")
'folder'
>>> os.path.basename("C:\\book\\folder\\")
''
```

- `dirname(<Путь>)` — возвращает путь к каталогу:

```
>>> os.path.dirname(r"C:\book\folder\file.txt")
'C:\\book\\folder'
>>> os.path.dirname(r"C:\book\folder")
'C:\\book'
>>> os.path.dirname("C:\\book\\folder\\")
'C:\\book\\folder'
```

- `split(<Путь>)` — возвращает кортеж из двух элементов: пути к каталогу и названия файла:

```
>>> os.path.split(r"C:\book\folder\file.txt")
('C:\\book\\folder', 'file.txt')
>>> os.path.split(r"C:\book\folder")
('C:\\book', 'folder')
>>> os.path.split("C:\\book\\folder\\")
('C:\\book\\folder', '')
```

- `splitdrive(<Путь>)` — разделяет путь на имя диска и остальную часть пути. В качестве значения возвращается кортеж из двух элементов:

```
>>> os.path.splitdrive(r"C:\book\folder\file.txt")
('C:', '\\book\\folder\\file.txt')
```

- `splitext(<Путь>)` — возвращает кортеж из двух элементов: пути с названием файла, но без расширения, и расширения файла (фрагмент после последней точки):

```
>>> os.path.splitext(r"C:\book\folder\file.tar.gz")
('C:\\book\\folder\\file.tar', '.gz')
```

- `join(<Путь1>[, ... <ПутьN>])` — соединяет указанные элементы пути:

```
>>> os.path.join("C:\\", "book\\folder", "file.txt")
'C:\\book\\folder\\file.txt'
>>> os.path.join(r"C:\\", "book/folder/", "file.txt")
'C:\\\\book/folder/file.txt'
```

  Обратите внимание на последний пример. В пути используются разные слеши и в результате получен некорректный путь. Чтобы этот путь сделать корректным, необходимо воспользоваться функцией `normpath()`:

```
>>> p = os.path.join(r"C:\\", "book/folder/", "file.txt")
>>> os.path.normpath(p)
'C:\\book\\folder\\file.txt'
```

15.8. Перенаправление ввода/вывода

При рассмотрении методов для работы с файлами говорилось, что значение, возвращаемое методом `fileno()`, всегда будет больше числа 2, т. к. число 0 закреплено за стандартным вводом `stdin`, 1 — за стандартным выводом `stdout`, а 2 — за стандартным выводом сообщений об ошибках `stderr`. Все эти потоки имеют некоторое сходство с файловыми объектами. Например, потоки `stdout` и `stderr` имеют метод `write()`, предназначенный для вывода сообщений, а поток `stdin` имеет метод `readline()`, предназначенный для получения входящих данных. Если этим объектам присвоить ссылку на объект, поддерживающий файловые методы, то можно перенаправить стандартные потоки в другое место. В качестве примера перенаправим вывод в файл (листинг 15.6).

Листинг 15.6. Перенаправление вывода в файл

```
>>> import sys                  # Подключаем модуль sys
>>> tmp_out = sys.stdout        # Сохраняем ссылку на sys.stdout
>>> f = open(r"file.txt", "a")  # Открываем файл на дозапись
>>> sys.stdout = f              # Перенаправляем вывод в файл
>>> print "Пишем строку в файл"
>>> sys.stdout = tmp_out        # Восстанавливаем стандартный вывод
>>> print "Пишем строку в стандартный вывод"
Пишем строку в стандартный вывод
>>> f.close()                   # Закрываем файл
```

В этом примере мы вначале сохранили ссылку на стандартный вывод в переменной `tmp_out`. С помощью этой переменной можно в дальнейшем восстановить вывод в стандартный поток.

Оператор `print` напрямую поддерживает перенаправление вывода. Для этого используется следующий формат:

```
print >> <Куда пишем>, <Что пишем>
```

Например, записать строку в файл можно так:

```
>>> f = open(r"file.txt", "a")
>>> print >> f, "Пишем строку в файл"
>>> f.close()
```

Стандартный ввод `stdin` также можно перенаправить. В этом случае функция `raw_input()` будет читать одну строку из файла при каждом вызове. При достижении конца файла возбуждается исключение `EOFError`. В качестве примера выведем содержимое файла с помощью перенаправления потока ввода (листинг 15.7).

Листинг 15.7. Перенаправление потока ввода

```
import sys
tmp_in = sys.stdin          # Сохраняем ссылку на sys.stdin
f = open(r"file.txt", "r")  # Открываем файл на чтение
sys.stdin = f               # Перенаправляем ввод
while True:
    try:
        line = raw_input()  # Считываем строку из файла
        print line          # Выводим строку
    except EOFError:        # Если достигнут конец файла
        break               # выходим из цикла
sys.stdin = tmp_in          # Восстанавливаем стандартный ввод
f.close()                   # Закрываем файл
```

Если необходимо узнать, ссылается ли стандартный ввод на терминал или нет, можно воспользоваться методом `isatty()`. Метод возвращает `True`, если объект ссылается на терминал, и `False` в противном случае. Пример:

```
>>> tmp_in = sys.stdin          # Сохраняем ссылку на sys.stdin
>>> f = open(r"file.txt", "r")
>>> sys.stdin = f               # Перенаправляем ввод
>>> sys.stdin.isatty()          # Не ссылается на терминал
False
>>> sys.stdin = tmp_in          # Восстанавливаем стандартный ввод
>>> sys.stdin.isatty()          # Ссылается на терминал
True
>>> f.close()                   # Закрываем файл
```

Перенаправить стандартный ввод/вывод можно также с помощью командной строки. В качестве примера создадим файл tests.py в папке C:\book с кодом, приведенным в листинге 15.8.

Листинг 15.8. Содержимое файла tests.py

```
# -*- coding: cp866 -*-
while True:
```

```
    try:
        line = raw_input()
        print line
    except EOFError:
        break
```

Запускаем командную строку и переходим в папку со скриптом, выполнив команду `cd C:\book`. Теперь выведем содержимое файла file.txt, выполнив команду:

```
C:\Python26\python.exe tests.py < file.txt
```

Перенаправить стандартный вывод в файл можно аналогичным образом. Только в этом случае символ < необходимо заменить на >. Изменим файл tests.py следующим образом:

```
# -*- coding: cp866 -*-
print "String"            # Эта строка будет записана в файл
```

Теперь перенаправим вывод в файл file.txt, выполнив команду:

```
C:\Python26\python.exe tests.py > file.txt
```

В этом режиме файл file.txt будет перезаписан. Если необходимо добавить результат в конец файла, следует использовать символ >>. Пример дозаписи в файл:

```
C:\Python26\python.exe tests.py >> file.txt
```

С помощью стандартного вывода `stdout` можно создать индикатор выполнения процесса в окне консоли. Чтобы реализовать такой индикатор, нужно вспомнить, что символ перевода строки в Windows состоит из двух символов: `\r` (перевод каретки) и `\n` (перевод строки). Таким образом, используя только символ перевода каретки `\r`, можно перемещаться в начало строки и перезаписывать ранее выведенную информацию. Рассмотрим вывод индикатора процесса на примере (листинг 15.9).

Листинг 15.9. Индикатор выполнения процесса

```
# -*- coding: cp866 -*-
import sys, time
for i in xrange(5, 101, 5):
    sys.stdout.write("\r ... %s%%" % i) # Обновляем индикатор
    sys.stdout.flush()                  # Сбрасываем содержимое буфера
    time.sleep(1)                       # Засыпаем на 1 секунду
sys.stdout.write("\rПроцесс завершен\n")
raw_input()
```

Сохраняем код в файл и запускаем с помощью двойного щелчка на ярлыке файла. В окне консоли записи будут заменять друг друга на одной строке каждую секунду. Так как данные перед выводом могут помещаться в буфер, мы сбрасываем их явным образом с помощью метода `flush()`.

15.9. Сохранение объектов в файл

Сохранить объекты в файл и в дальнейшем восстановить объекты из файла позволяют модули `pickle` и `shelve`. Модуль `pickle` предоставляет следующие функции:

- `dump(<Объект>, <Файл>[, <Протокол>])` — производит сериализацию объекта и записывает данные в указанный файл. В параметре `<Файл>` указывается файловый объект, открытый на запись. Пример сохранения объекта в файл:

```
>>> import pickle
>>> f = open(r"file.txt", "wb")
>>> obj = ["Строка", (2, 3)]
>>> pickle.dump(obj, f)
>>> f.close()
```

- `load(<Файл>)` — читает данные из файла и преобразует их в объект. В параметре `<Файл>` указывается файловый объект, открытый на чтение. Пример восстановления объекта из файла:

```
>>> f = open(r"file.txt", "rb")
>>> obj = pickle.load(f)
>>> obj
['\xd1\xf2\xf0\xee\xea\xe0', (2, 3)]
>>> f.close()
```

В один файл можно сохранить сразу несколько объектов, последовательно вызывая функцию `dump()`. Пример сохранения нескольких объектов приведен в листинге 15.10.

Листинг 15.10. Сохранение нескольких объектов

```
>>> obj1 = ["Строка", (2, 3)]
>>> obj2 = (1, 2)
>>> f = open(r"file.txt", "wb")
>>> pickle.dump(obj1, f)          # Сохраняем первый объект
>>> pickle.dump(obj2, f)          # Сохраняем второй объект
>>> f.close()
```

Для восстановления объектов необходимо несколько раз вызвать функцию `load()` (листинг 15.11).

Листинг 15.11. Восстановление нескольких объектов

```
>>> f = open(r"file.txt", "rb")
>>> obj1 = pickle.load(f)        # Восстанавливаем первый объект
>>> obj2 = pickle.load(f)        # Восстанавливаем второй объект
>>> obj1, obj2
(['\xd1\xf2\xf0\xee\xea\xe0', (2, 3)], (1, 2))
>>> f.close()
```

Сохранить объект в файл можно также с помощью метода `dump(<Объект>)` класса `Pickler`. Конструктор класса имеет следующий формат:

```
Pickler(<Файл>[, <Протокол>])
```

Пример сохранения объекта в файл:

```
>>> f = open(r"file.txt", "wb")
>>> obj = ["Строка", (2, 3)]
>>> pkl = pickle.Pickler(f)
>>> pkl.dump(obj)
>>> f.close()
```

Восстановить объект из файла позволяет метод `load()` из класса `Unpickler`. Формат конструктора класса:

```
Unpickler(<Файл>)
```

Пример восстановления объекта из файла:

```
>>> f = open(r"file.txt", "rb")
>>> obj = pickle.Unpickler(f).load()
>>> obj
['\xd1\xf2\xf0\xee\xea\xe0', (2, 3)]
>>> f.close()
```

Модуль `pickle` позволяет также преобразовать объект в строку и восстановить объект из строки. Для этого предназначены две функции:

- `dumps(<Объект>[, <Протокол>])` — производит сериализацию объекта и возвращает строку специального формата. Формат этой строки зависит от указанного протокола (число от `0` до `2`). В качестве примера выведем результат преобразования в различных протоколах:

```
>>> obj = ["Строка", (2, 3)]
>>> pickle.dumps(obj)    # Без указания протокола
"(lp0\nS'\\xd1\\xf2\\xf0\\xee\\xea\\xe0'\np1\na(I2\nI3\ntp2\na."
>>> pickle.dumps(obj, 0) # Протокол 0
"(lp0\nS'\\xd1\\xf2\\xf0\\xee\\xea\\xe0'\np1\na(I2\nI3\ntp2\na."
>>> pickle.dumps(obj, 1) # Протокол 1
']q\x00(U\x06\xd1\xf2\xf0\xee\xea\xe0q\x01(K\x02K\x03tq\x02e.'
>>> pickle.dumps(obj, 2) # Протокол 2
'\x80\x02]q\x00(U\x06\xd1\xf2\xf0\xee\xea\xe0q\x01K\x02K\x03
\x86q\x02e.'
```

- `loads(<Строка>)` — преобразует строку специального формата обратно в объект. Пример восстановления объекта из строки:

```
>>> obj = ["Строка", (2, 3)]
>>> pkl = pickle.dumps(obj)
>>> obj = pickle.loads(pkl)
>>> obj
['\xd1\xf2\xf0\xee\xea\xe0', (2, 3)]
```

Модуль `shelve` позволяет сохранять объекты под определенным ключом (задается в виде строки) и предоставляет интерфейс доступа, сходный со словарями.

Для сериализации объекта используются возможности модуля pickle, а чтобы записать получившуюся строку по ключу в файл, применяется модуль anydbm. Все эти действия модуль shelve производит незаметно для нас.

Чтобы открыть файл с базой объектов, используется функция open(). Функция имеет следующий формат:

```
open(<Путь к файлу>[, flag="c"[, protocol=None[, writeback=False]]])
```

В необязательном параметре flag можно указать один из режимов открытия файла:

- r — только чтение;
- w — чтение и запись;
- c — чтение и запись (значение по умолчанию). Если файл не существует, он будет создан;
- n — чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан.

Функция open() возвращает объект, с помощью которого производится дальнейшая работа с базой данных. Этот объект имеет следующие методы:

- close() — закрывает файл с базой данных. В качестве примера создадим файл и сохраним в нем список и кортеж:

```
>>> import shelve                    # Подключаем модуль
>>> db = shelve.open("file2.txt")    # Открываем файл
>>> db["obj1"] = [1, 2, 3, 4, 5]     # Сохраняем список
>>> db["obj2"] = (6, 7, 8, 9, 10)    # Сохраняем кортеж
>>> db["obj1"], db["obj2"]           # Вывод значений
([1, 2, 3, 4, 5], (6, 7, 8, 9, 10))
>>> db.close()                       # Закрываем файл
```

- keys() и values() — позволяют получить список всех ключей и значений соответственно. Можно также воспользоваться методами iterkeys() и itervalues(), которые возвращают не список ключей и значений, а итератор;
- items() — возвращает список кортежей. Каждый кортеж содержит ключ и значение. Можно также воспользоваться методом iteritems(), который возвращает итератор;
- has_key(<Ключ>) — проверяет существование указанного ключа. Если ключ найден, то возвращается значение True, в противном случае — False. Пример:

```
>>> db = shelve.open("file2.txt")
>>> db.keys(), db.values()
(['obj1', 'obj2'], [[1, 2, 3, 4, 5], (6, 7, 8, 9, 10)])
>>> db.items()
[('obj1', [1, 2, 3, 4, 5]), ('obj2', (6, 7, 8, 9, 10))]
>>> db.has_key("obj1"), db.has_key("obj3")
(True, False)
>>> db.close()
```

- get(<Ключ>[, <Значение по умолчанию>]) — если ключ присутствует, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается значение None или значение, указанное во втором параметре;

- `setdefault(<Ключ>[, <Значение по умолчанию>])` — если ключ присутствует, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то вставляет новый элемент со значением, указанным во втором параметре, и возвращает это значение. Если второй параметр не указан, значением нового элемента будет `None`;
- `pop(<Ключ>[, <Значение по умолчанию>])` — удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, то возвращается значение из второго параметра. Если ключ отсутствует и второй параметр не указан, то возбуждается исключение `KeyError`;
- `popitem()` — удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если файл пустой, возбуждается исключение `KeyError`;
- `clear()` — удаляет все элементы. Метод ничего не возвращает в качестве значения;
- `update()` — добавляет элементы. Метод изменяет текущий объект и ничего не возвращает. Если элемент с указанным ключом уже присутствует, то его значение будет перезаписано. Форматы метода:

```
update(<Ключ1>=<Значение1>[, ..., <КлючN>=<ЗначениеN>])
update(<Словарь>)
update(<Список кортежей с двумя элементами>)
update(<Список списков с двумя элементами>)
```

Помимо этих методов можно воспользоваться функцией `len()` для получения количества элементов и оператором `del` для удаления определенного элемента. Пример:

```
>>> db = shelve.open("file2.txt")
>>> len(db)            # Количество элементов
2
>>> del db["obj1"]     # Удаление элемента
>>> db.close()
```

15.10. Функции для работы с каталогами

Для работы с каталогами используются следующие функции из модуля `os`:

- `getcwd()` — возвращает текущий рабочий каталог. От значения, возвращаемого этой функцией, зависит преобразование относительного пути в абсолютный. Кроме того, важно помнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог с исполняемым файлом. Пример:

```
>>> import os
>>> os.getcwd()              # Текущий рабочий каталог
'C:\\book'
```

- `chdir(<Имя каталога>)` — делает указанный каталог текущим:

```
>>> os.chdir("C:\\book\\folder1\\")
>>> os.getcwd()              # Текущий рабочий каталог
'C:\\book\\folder1'
```

- `mkdir(<Имя каталога>[, <Права доступа>])` — создает новый каталог с правами доступа, указанными во втором параметре. Права доступа задаются трехзначным числом, перед которым указывается `0` (значение второго параметра по умолчанию `0777`). Пример создания нового каталога в текущем рабочем каталоге:

```
>>> os.mkdir("newfolder")      # Создание каталога
```

- `rmdir(<Имя каталога>)` — удаляет пустой каталог. Если в каталоге есть файлы или указанный каталог не существует, то в Windows возбуждается исключение `WindowsError`. Удалим каталог `newfolder`:

```
>>> os.rmdir("newfolder") # Удаление каталога
```

- `listdir(<Путь>)` — возвращает список объектов в указанном каталоге:

```
>>> os.listdir("C:\\book\\folder1\\")
['file1.txt', 'file2.txt', 'file3.txt', 'folder1', 'folder2']
```

- `walk()` — позволяет обойти дерево каталогов. Формат функции:

```
walk(<Начальный каталог>[, topdown=True[, onerror=None
     [, followlinks=False]]])
```

  В качестве значения функция `walk()` возвращает объект-генератор. На каждой итерации через этот объект доступен кортеж из трех элементов: текущий каталог, список каталогов и список файлов. Если произвести изменения в списке каталогов во время выполнения, то это позволит изменить порядок обхода вложенных каталогов.

  Необязательный параметр `topdown` задает последовательность обхода каталогов. Если в качестве значения указано `True` (значение по умолчанию), то последовательность обхода будет такой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\"): print p

C:\book\folder1\
C:\book\folder1\folder1_1
C:\book\folder1\folder1_1\folder1_1_1
C:\book\folder1\folder1_1\folder1_1_2
C:\book\folder1\folder1_2
```

  Если в качестве значения указано `False`, то последовательность обхода будет другой:

```
>>> for (p, d, f) in os.walk("C:\\book\\folder1\\", False):
       print p

C:\book\folder1\folder1_1\folder1_1_1
C:\book\folder1\folder1_1\folder1_1_2
C:\book\folder1\folder1_1
C:\book\folder1\folder1_2
C:\book\folder1\
```

  Благодаря такой последовательности обхода каталогов можно удалить все вложенные файлы и каталоги. Это особенно важно при удалении каталога, т. к.

функция `rmdir()` позволяет удалить только пустой каталог. Пример очистки дерева каталогов:

```
import os
for (p, d, f) in os.walk("C:\\book\\folder1\\", False):
    for file_name in f: # Удаляем все файлы
        os.remove(os.path.join(p, file_name))
    for dir_name in d: # Удаляем все каталоги
        os.rmdir(os.path.join(p, dir_name))
```

Внимание!

Очень осторожно используйте этот код. Если в качестве первого параметра в функции `walk()` указать корневой каталог диска, то все файлы и каталоги будут удалены.

Удалить дерево каталогов позволяет также функция `rmtree()` из модуля `shutil`. Функция имеет следующий формат:

```
rmtree(<Путь>[, <Обработка ошибок>[, <Обработчик ошибок>]])
```

Если в параметре `<Обработка ошибок>` указано значение `True`, то ошибки будут проигнорированы. Если указано значение `False` (значение по умолчанию), то в третьем параметре можно указать ссылку на функцию-обработчик. Эта функция будет вызываться при возникновении исключения. Пример удаления дерева каталогов вместе с начальным каталогом:

```
import shutil
shutil.rmtree("C:\\book\\folder1\\")
```

Как вы уже знаете, функция `listdir()` возвращает список объектов в указанном каталоге. Проверить, на какой тип объекта ссылается элемент этого списка, можно с помощью следующих функций из модуля `os.path`:

- `isdir(<Объект>)` — возвращает `True`, если объект является каталогом, и `False` в противном случае:

```
>>> import os.path
>>> os.path.isdir(r"C:\book\file.txt")
False
>>> os.path.isdir("C:\\book\\")
True
```

- `isfile(<Объект>)` — возвращает `True`, если объект является файлом, и `False` в противном случае:

```
>>> os.path.isfile(r"C:\book\file.txt")
True
>>> os.path.isfile("C:\\book\\")
False
```

- `islink(<Объект>)` — возвращает `True`, если объект является символической ссылкой, и `False` в противном случае. Если символические ссылки не поддерживаются, функция возвращает `False`.

Функция `listdir()` возвращает список всех объектов в указанном каталоге. Если необходимо ограничить список определенными критериями, то следует воспользоваться функцией `glob(<Путь>)` из модуля `glob`. Функция `glob()` позволяет указать в пути следующие специальные символы:

- `?` — любой одиночный символ;
- `*` — любое количество символов;
- `[<Символы>]` — позволяет указать символы, которые должны быть на этом месте в пути. Можно перечислить символы или указать диапазон через тире.

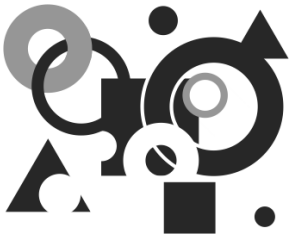
В качестве значения функция возвращает список путей к объектам, совпадающим с шаблоном. Пример использования функции `glob()` приведен в листинге 15.12.

Листинг 15.12. Пример использования функции `glob()`

```
>>> import os, glob
>>> os.listdir("C:\\book\\folder1\\")
['file.txt', 'file1.txt', 'file2.txt', 'folder1_1', 'folder1_2',
'index.html']
>>> glob.glob("C:\\book\\folder1\\*.txt")
['C:\\book\\folder1\\file.txt', 'C:\\book\\folder1\\file1.txt',
'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*.html") # Абсолютный путь
['C:\\book\\folder1\\index.html']
>>> glob.glob("folder1/*.html")            # Относительный путь
['folder1\\index.html']
>>> glob.glob("C:\\book\\folder1\\*[0-9].txt")
['C:\\book\\folder1\\file1.txt', 'C:\\book\\folder1\\file2.txt']
>>> glob.glob("C:\\book\\folder1\\*\\*.html")
['C:\\book\\folder1\\folder1_1\\index.html',
'C:\\book\\folder1\\folder1_2\\test.html']
```

Обратите внимание на последний пример. Специальные символы могут быть указаны не только в названии файла, но и в именах каталогов в пути. Это позволяет просматривать сразу несколько каталогов в поисках объектов, соответствующих шаблону.

Глава 16



Основы SQLite

В предыдущей главе мы рассмотрели работу с файлами и научились сохранять объекты с доступом по ключу с помощью модуля `shelve`. При сохранении объектов этот модуль использует возможности модуля `pickle`, для сериализации объекта и модуль `anydbm` для записи получившейся строки по ключу в файл. Если необходимо сохранять в файл просто строки, то можно сразу воспользоваться модулем `anydbm`. Однако если объем сохраняемых данных велик и требуется удобный доступ к ним, то вместо этого модуля лучше использовать базы данных.

Начиная с версии 2.5, в состав стандартной библиотеки Python входит модуль `sqlite3`, позволяющий работать с базой данных SQLite. Для использования этой базы данных нет необходимости устанавливать сервер, ожидающий запросы на каком-либо порту, т. к. SQLite напрямую работает с файлом базы данных. Все что нужно для работы с SQLite, это библиотека sqlite3.dll (расположена в папке C:\Python26\DLLs) и язык программирования, позволяющий использовать эту библиотеку (например, Python). Необходимо заметить, что база данных SQLite не предназначена для проектов, предъявляющих требования к защите данных и разграничению прав доступа для нескольких пользователей. Тем не менее, для небольших проектов SQLite является хорошей заменой полноценных баз данных.

Так как SQLite входит в состав стандартной библиотеки Python, мы на некоторое время отвлечемся от изучения языка Python и рассмотрим особенности использования языка SQL (Structured Query Language — структурированный язык запросов) применительно к базе данных SQLite. Для выполнения SQL-запросов мы воспользуемся программой sqlite3.exe, позволяющей работать с SQLite из командной строки. Со страницы **http://www.sqlite.org/download.html** загружаем архив sqlite-3_6_23.zip, а затем распаковываем его в текущую папку. Далее копируем файл sqlite3.exe в каталог, с которым будем в дальнейшем работать (например, C:\book).

16.1. Создание базы данных

Попробуем создать новую базу данных. Запускаем командную строку. Для этого в меню **Пуск** выбираем пункт **Выполнить**. В открывшемся окне набираем команду `cmd` и нажимаем кнопку **OK**. Откроется черное окно, в котором будет приглашение для ввода команд. Переходим в папку C:\book, выполнив команду:

```
cd C:\book
```

В командной строке должно быть приглашение:

```
C:\book>
```

По умолчанию в консоли используется кодировка cp866. Чтобы сменить кодировку на cp1251, в командной строке вводим команду:

```
chcp 1251
```

Теперь необходимо изменить название шрифта, т. к. точечные шрифты не поддерживают кодировку Windows-1251. Щелкаем правой кнопкой мыши на заголовке окна и из контекстного меню выбираем пункт **Свойства**. В открывшемся окне переходим на вкладку **Шрифт** и в списке выделяем пункт **Lucida Console**. На этой же вкладке можно также установить размер шрифта. Нажимаем кнопку **OK**, чтобы изменения вступили в силу. Для проверки правильности установки кодировки вводим команду `chcp`. Результат выполнения должен выглядеть так:

```
C:\book>chcp
Текущая кодовая страница: 1251
```

Для создания новой базы данных вводим команду:

```
C:\book>sqlite3.exe testdb.db
```

Если файл testdb.db не существует, то будет создана новая база данных и открыта для дальнейшей работы. Если база данных уже существует, то она просто открывается без удаления содержимого. Результат выполнения команды будет выглядеть так:

```
SQLite version 3.6.23
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Примечание

В примерах следующих разделов предполагается, что база данных была открыта указанным способом. Поэтому запомните способ изменения кодировки в консоли и способ создания (или открытия) базы данных.

Фрагмент `"sqlite> "` является приглашением для ввода SQL-команд. Каждая SQL-команда должна завершаться точкой с запятой. Если точку с запятой не указать и нажать клавишу <Enter>, то приглашение примет вид `"  ...> "`. В качестве примера получим версию SQLite:

```
sqlite> SELECT sqlite_version();
3.6.23
```

```
sqlite> SELECT sqlite_version()
   ...> ;
3.6.23
```

SQLite позволяет использовать комментарии. Однострочный комментарий начинается с двух тире и оканчивается в конце строки. В этом случае после комментария точку с запятой указывать не нужно. Многострочный комментарий начинается с комбинации символов `/*` и заканчивается комбинацией `*/`. Допускается отсутствие завершающей комбинации символов. В этом случае комментируется фрагмент до конца файла. Многострочные комментарии не могут быть вложенными. Если внутри многострочного комментария расположен однострочный комментарий, то он игнорируется. Пример использования комментариев:

```
sqlite> -- Это однострочный комментарий
sqlite> /* Это многострочный комментарий */
sqlite> SELECT sqlite_version(); -- Комментарий после SQL-команды
3.6.23
sqlite> SELECT sqlite_version(); /* Комментарий после SQL-команды */
3.6.23
```

Чтобы завершить работу с SQLite и закрыть базу данных, следует нажать комбинацию клавиш <Ctrl>+<C>.

16.2. Создание таблицы

Создать таблицу в базе данных позволяет следующая SQL-команда:

```
CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS]
[<Название базы данных>.]<Название таблицы> (
   <Название поля1> [<Тип данных>] [<Опции>],
   [...,
   <Название поляN> [<Тип данных>] [<Опции>],]
   [<Дополнительные опции>]
);
```

Если после ключевого слова `CREATE` указано слово `TEMP` или `TEMPORARY`, то будет создана временная таблица. После закрытия базы данных временные таблицы автоматически удаляются. Пример создания временных таблиц:

```
sqlite> CREATE TEMP TABLE tmp1 (pole1);
sqlite> CREATE TEMPORARY TABLE tmp2 (pole1);
sqlite> .tables
tmp1  tmp2
```

Обратите внимание на предпоследнюю строку. С помощью команды `.tables` мы получаем список всех таблиц в базе данных. Эта команда работает только в утилите sqlite3.exe и является сокращенной записью следующего SQL-запроса:

```
sqlite> SELECT name FROM sqlite_master
   ...> WHERE type IN ('table','view') AND name NOT LIKE 'sqlite_%'
```

```
   ...> UNION ALL
   ...> SELECT name FROM sqlite_temp_master
   ...> WHERE type IN ('table','view')
   ...> ORDER BY 1;
tmp1
tmp2
```

Необязательные ключевые слова `IF NOT EXISTS` означают, что если таблица уже существует, то создавать таблицу заново не нужно. Если таблица уже существует и ключевые слова `IF NOT EXISTS` не указаны, то будет выведено сообщение об ошибке. Пример:

```
sqlite> CREATE TEMP TABLE tmp1 (pole3);
Error: table tmp1 already exists
sqlite> CREATE TEMP TABLE IF NOT EXISTS tmp1 (pole3);
sqlite> PRAGMA table_info(tmp1);
0|pole1||0||0
```

В этом примере мы использовали SQL-команду `PRAGMA table_info(<Название таблицы>)`, позволяющую получить информацию о полях таблицы (название поля, тип данных, значение по умолчанию и др.). Как видно из результата, структура временной таблицы `tmp1` не изменилась после выполнения запроса на создание таблицы с таким же названием.

В параметрах `<Название таблицы>` и `<Название поля>` указывается идентификатор или строка. В идентификаторах лучше использовать только буквы латинского алфавита, цифры и символ подчеркивания. Имена, начинающиеся с префикса `"sqlite_"`, зарезервированы для служебного использования. Если в этих параметрах указывается идентификатор, то название не должно содержать пробелов и не должно совпадать с ключевыми словами SQL. Например, при попытке назвать таблицу именем `table` будет выведено сообщение об ошибке:

```
sqlite> CREATE TEMP TABLE table (pole1);
Error: near "table": syntax error
```

Если вместо идентификатора указать строку, то сообщения об ошибке не возникнет:

```
sqlite> CREATE TEMP TABLE "table" (pole1);
sqlite> .tables
table  tmp1   tmp2
```

Кроме того, идентификатор можно разместить внутри квадратных скобок:

```
sqlite> DROP TABLE "table";
sqlite> CREATE TEMP TABLE [table] (pole1);
sqlite> .tables
table  tmp1   tmp2
```

Примечание

Хотя ошибки и удается избежать, на практике не стоит использовать ключевые слова SQL в качестве названия таблицы или поля.

Обратите внимание на первую строку примера. С помощью SQL-команды `DROP TABLE <Название таблицы>` мы удаляем таблицу `table` из базы данных. Если этого не сделать, то попытка создать таблицу, при наличии уже существующей одноименной таблицы, приведет к выводу сообщения об ошибке. SQL-команда `DROP TABLE` позволяет удалить как обычную таблицу, так и временную таблицу.

В целях совместимости с другими базами данных значение, указанное в параметре `<Тип данных>`, преобразуется в один из пяти классов родства:

- `INTEGER` — класс будет назначен, если значение содержит фрагмент `"INT"` в любом месте. Этому классу родства соответствуют типы данных `INT`, `INTEGER`, `TINYINT`, `SMALLINT`, `MEDIUMINT`, `BIGINT` и др.;
- `TEXT` — если значение содержит фрагменты `"CHAR"`, `"CLOB"` или `"TEXT"`. Например, `TEXT`, `CHARACTER(30)`, `VARCHAR(250)`, `VARYING CHARACTER(100)`, `CLOB` и др. Все значения внутри круглых скобок игнорируются;
- `NONE` — если значение содержит фрагмент `"BLOB"` или тип данных не указан;
- `REAL` — если значение содержит фрагменты `"REAL"`, `"FLOA"` или `"DOUB"`. Например, `REAL`, `DOUBLE`, `DOUBLE PRECISION`, `FLOAT`;
- `NUMERIC` — если все предыдущие условия не выполняются, то назначается этот класс родства.

> ***Внимание!***
>
> Все классы указаны в порядке уменьшения приоритета определения родства. Например, если значение соответствует сразу двум классам `INTEGER` и `TEXT`, то будет назначен класс `INTEGER`, т. к. его приоритет выше.

Классы родства являются лишь обозначением предполагаемого типа данных, а не строго определенным значением. Иными словами, SQLite использует не статическую типизацию (как в большинстве баз данных), а динамическую типизацию. Например, если для поля указан класс `INTEGER`, то при вставке значения производится попытка преобразовать введенные данные в целое число. Если преобразовать не получилось, то производится попытка преобразовать введенные данные в вещественное число. Если данные нельзя преобразовать в целое или вещественное число, то будет произведена попытка преобразовать в строку и т. д. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1 INTEGER,  p2 INTEGER,
   ...>  p3 INTEGER,  p4 INTEGER,  p5 INTEGER);
sqlite> INSERT INTO tmp3 VALUES (10, "00547", 5.45, "Строка", NULL);
sqlite> SELECT * FROM tmp3;
10|547|5.45|Строка|
sqlite> SELECT typeof(p1), typeof(p2), typeof(p3), typeof(p4),
   ...> typeof(p5) FROM tmp3;
integer|integer|real|text|null
sqlite> DROP TABLE tmp3;
```

В этом примере мы воспользовались встроенной функцией `typeof()` для определения типа данных, хранящихся в ячейке таблицы. SQLite поддерживает следующие типы данных:

- `NULL` — значение `NULL`;
- `INTEGER` — целые числа;

- REAL — вещественные числа;
- TEXT — строки;
- BLOB — бинарные данные.

Если после INTEGER указаны ключевые слова PRIMARY KEY (т. е. поле является первичным ключом), то в это поле можно вставить только целые числа или значение NULL. При указании значения NULL будет вставлено число, на единицу большее максимального числа в столбце. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1 INTEGER PRIMARY KEY);
sqlite> INSERT INTO tmp3 VALUES (10);   -- Нормально
sqlite> INSERT INTO tmp3 VALUES (5.78); -- Ошибка
Error: datatype mismatch
sqlite> INSERT INTO tmp3 VALUES ("Строка"); -- Ошибка
Error: datatype mismatch
sqlite> INSERT INTO tmp3 VALUES (NULL);
sqlite> SELECT * FROM tmp3;
10
11
sqlite> DROP TABLE tmp3;
```

Класс NUMERIC аналогичен классу INTEGER. Различие между этими классами проявляется только при явном преобразовании типов с помощью инструкции CAST. Если строку, содержащую вещественное число, преобразовать в класс INTEGER, то дробная часть будет отброшена. Если строку, содержащую вещественное число, преобразовать в класс NUMERIC, то возможны два варианта:

- если преобразование в целое число возможно без потерь, то данные будут иметь тип INTEGER;
- в противном случае — тип REAL.

Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1 TEXT);
sqlite> INSERT INTO tmp3 VALUES ("00012.86");
sqlite> INSERT INTO tmp3 VALUES ("52.0");
sqlite> SELECT p1, typeof(p1) FROM tmp3;
00012.86|text
52.0|text
sqlite> SELECT CAST (p1 AS INTEGER) FROM tmp3;
12
52
sqlite> SELECT CAST (p1 AS NUMERIC) FROM tmp3;
12.86
52
sqlite> DROP TABLE tmp3;
```

В параметре <Опции> могут быть указаны следующие конструкции:

- NOT NULL [<Обработка ошибок>] — означает, что поле обязательно должно иметь значение при вставке новой записи. Если опция не указана, то поле может содержать значение NULL;

- `DEFAULT <Значение>` — задает для поля значение по умолчанию, которое будет использовано, если при вставке записи для этого поля не было явно указано значение. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1, p2 INTEGER DEFAULT 0);
sqlite> INSERT INTO tmp3 (p1) VALUES (800);
sqlite> INSERT INTO tmp3 VALUES (5, 1204);
sqlite> SELECT * FROM tmp3;
800|0
5|1204
sqlite> DROP TABLE tmp3;
```

  В параметре `<Значение>` можно указать специальные значения:

  - `CURRENT_TIME` — текущее время UTC в формате `ЧЧ:ММ:СС`;
  - `CURRENT_DATE` — текущая дата UTC в формате `ГГГГ-ММ-ДД`;
  - `CURRENT_TIMESTAMP` — текущая дата и время UTC в формате `ГГГГ-ММ-ДД ЧЧ:ММ:СС`.

  Пример указания специальных значений:

```
sqlite> CREATE TEMP TABLE tmp3 (id INTEGER,
   ...> t TEXT DEFAULT CURRENT_TIME,
   ...> d TEXT DEFAULT CURRENT_DATE,
   ...> dt TEXT DEFAULT CURRENT_TIMESTAMP);
sqlite> INSERT INTO tmp3 (id) VALUES (1);
sqlite> SELECT * FROM tmp3;
1|17:04:01|2010-06-07|2010-06-07 17:04:01
sqlite> /* Текущая дата на компьютере: 2010-06-07 21:04:01 */
sqlite> DROP TABLE tmp3;
```

- `COLLATE <Функция>` — задает функцию сравнения для класса `TEXT`. Могут быть указаны функции `BINARY` (значение по умолчанию), `NOCASE` (без учета регистра) и `RTRIM`. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (p1, p2 TEXT COLLATE NOCASE);
sqlite> INSERT INTO tmp3 VALUES ("abcd", "abcd");
sqlite> SELECT p1 = "ABCD" FROM tmp3; -- Не найдено
0
sqlite> SELECT p2 = "ABCD" FROM tmp3; -- Найдено
1
sqlite> DROP TABLE tmp3;
```

Примечание

При использовании `NOCASE` возможны проблемы с регистром русских букв.

- `UNIQUE [<Обработка ошибок>]` — указывает, что поле может содержать только уникальные значения;
- `CHECK(<Условие>)` — значение, вставляемое в поле, должно удовлетворять указанному условию. В качестве примера ограничим значения числами 10 и 20:

```
sqlite> CREATE TEMP TABLE tmp3 (
   ...> p1 INTEGER CHECK(p1 IN (10, 20)));
```

```
sqlite> INSERT INTO tmp3 VALUES (10); -- OK
sqlite> INSERT INTO tmp3 VALUES (30); -- Ошибка
Error: constraint failed
sqlite> DROP TABLE tmp3;
```

- `PRIMARY KEY [ASC | DESC] [<Обработка ошибок>] [AUTOINCREMENT]` — указывает, что поле является первичным ключом таблицы. Записи в таком поле должны быть уникальными. Если полю назначен класс `INTEGER`, то в это поле можно вставить только целые числа или значение `NULL`. При указании значения `NULL` будет вставлено число, на единицу большее максимального числа в столбце. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (id INTEGER PRIMARY KEY, t TEXT);
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка1");
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка2");
sqlite> SELECT * FROM tmp3;
1|Строка1
2|Строка2
sqlite> DELETE FROM tmp3 WHERE id=2;
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка3");
sqlite> SELECT * FROM tmp3;
1|Строка1
2|Строка3
sqlite> DROP TABLE tmp3;
```

В этом примере мы вставили две записи. Так как при вставке для первого поля указано значение `NULL`, новая запись всегда будет иметь значение на единицу больше максимального числа в поле. Если удалить последнюю запись, а затем вставить новую запись, то запись будет иметь такой же индекс, что и удаленная. Чтобы индекс всегда был уникальным, необходимо дополнительно указать ключевое слово `AUTOINCREMENT`. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (
   ...> id INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> t TEXT);
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка1");
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка2");
sqlite> SELECT * FROM tmp3;
1|Строка1
2|Строка2
sqlite> DELETE FROM tmp3 WHERE id=2;
sqlite> INSERT INTO tmp3 VALUES (NULL, "Строка3");
sqlite> SELECT * FROM tmp3;
1|Строка1
3|Строка3
sqlite> DROP TABLE tmp3;
```

Обратите внимание на индекс последней вставленной записи. Индекс имеет значение 3, а не 2, как это было в предыдущем примере. Таким образом, индекс новой записи всегда будет уникальным.

Если в таблице не существует поля с первичным ключом, то получить индекс записи можно с помощью специальных названий полей: ROWID, OID или _ROWID_. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (t TEXT);
sqlite> INSERT INTO tmp3 VALUES ("Строка1");
sqlite> INSERT INTO tmp3 VALUES ("Строка2");
sqlite> SELECT ROWID, OID, _ROWID_, t FROM tmp3;
1|1|1|Строка1
2|2|2|Строка2
sqlite> DELETE FROM tmp3 WHERE OID=2;
sqlite> INSERT INTO tmp3 VALUES ("Строка3");
sqlite> SELECT ROWID, OID, _ROWID_, t FROM tmp3;
1|1|1|Строка1
2|2|2|Строка3
sqlite> DROP TABLE tmp3;
```

В необязательном параметре <Дополнительные опции> могут быть указаны следующие конструкции:

- PRIMARY KEY (<Список полей через запятую>) [<Обработка ошибок>] — позволяет задать первичный ключ для нескольких полей таблицы;
- UNIQUE (<Список полей через запятую>) [<Обработка ошибок>] — указывает, что заданные поля могут содержать только уникальные значения;
- CHECK(<Условие>) — значение должно удовлетворять указанному условию.

Необязательный параметр <Обработка ошибок> во всех рассмотренных в этом разделе конструкциях задает способ разрешения конфликтных ситуаций. Формат конструкции:

```
ON CONFLICT <Алгоритм>
```

В параметре <Алгоритм> указываются следующие значения:

- ROLLBACK — при ошибке транзакция завершается с откатом всех измененных ранее записей, дальнейшее выполнение прерывается и выводится сообщение об ошибке. Если активной транзакции нет, то используется алгоритм ABORT;
- ABORT — при возникновении ошибки аннулируются все изменения, произведенные текущей командой, и выводится сообщение об ошибке. Все изменения, сделанные предыдущими командами в транзакции, сохраняются. Алгоритм ABORT используется по умолчанию;
- FAIL — при возникновении ошибки все изменения, произведенные текущей командой, сохраняются, а не аннулируются как в алгоритме ABORT. Дальнейшее выполнение команды прерывается и выводится сообщение об ошибке. Все изменения, сделанные предыдущими командами в транзакции, сохраняются;
- IGNORE — проигнорировать ошибку и продолжить выполнение без вывода сообщения об ошибке;

- REPLACE — при нарушении условия UNIQUE существующая запись удаляется, а новая вставляется. Сообщение об ошибке не выводится. При нарушении условия NOT NULL значение NULL заменяется значением по умолчанию. Если значение по умолчанию не задано для поля, то используется алгоритм ABORT. Если нарушено условие CHECK, применяется алгоритм IGNORE. Пример обработки условия UNIQUE:

```
sqlite> CREATE TEMP TABLE tmp3 (
   ...> id UNIQUE ON CONFLICT REPLACE, t TEXT);
sqlite> INSERT INTO tmp3 VALUES (10, "s1");
sqlite> INSERT INTO tmp3 VALUES (10, "s2");
sqlite> SELECT * FROM tmp3;
10|s2
sqlite> DROP TABLE tmp3;
```

16.3. Вставка записей

Для добавления записей в таблицу используется инструкция INSERT. Формат инструкции:

```
INSERT [OR <Алгоритм>] INTO [<Название базы данных>.]<Название таблицы>
[(<Поле1>, <Поле2>, ...)] VALUES (<Значение1>, <Значение2>, ...);
```

Необязательный параметр OR <Алгоритм> задает алгоритм обработки ошибок (ROLLBACK, ABORT, FAIL, IGNORE или REPLACE). Все эти алгоритмы мы уже рассматривали в предыдущем разделе. После названия таблицы внутри круглых скобок могут быть перечислены поля, которым будут присваиваться значения, указанные в круглых скобках после ключевого слова VALUES. Количество параметров должно совпадать. Если в таблице существуют поля, которым в инструкции INSERT не присваивается значение, то они получат значения по умолчанию. Если список полей не указан, то значения задаются в том порядке, в котором поля перечислены в инструкции CREATE TABLE.

Создадим три таблицы user (данные о пользователе), rubr (название рубрики) и site (описание сайта):

```
sqlite> CREATE TABLE user (
   ...> id_user INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> email TEXT,
   ...> passw TEXT);
sqlite> CREATE TABLE rubr (
   ...> id_rubr INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> name_rubr TEXT);
sqlite> CREATE TABLE site (
   ...> id_site INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> id_user INTEGER,
   ...> id_rubr INTEGER,
```

```
   ...> url TEXT,
   ...> title TEXT,
   ...> msg TEXT);
```

Такая структура таблиц характерна для реляционных баз данных и позволяет избежать дублирования данных в таблицах, ведь одному пользователю может принадлежать несколько сайтов, а в одной рубрике можно зарегистрировать множество сайтов. Если в таблице `site` каждый раз указывать название рубрики, то при необходимости переименовать рубрику придется изменять названия во всех записях, где встречается старое название. Если же название рубрик расположено в отдельной таблице, то изменить название можно будет только в одном месте. Все остальные записи будут связаны целочисленным идентификатором. Как получить данные сразу из нескольких таблиц мы рассмотрим по мере изучения SQLite.

Теперь заполним таблицы связанными данными:

```
sqlite> INSERT INTO user (email, passw)
   ...> VALUES ('unicross@mail.ru', 'password1');
sqlite> INSERT INTO rubr VALUES (NULL, 'Программирование');
sqlite> SELECT * FROM user;
1|unicross@mail.ru|password1
sqlite> SELECT * FROM rubr;
1|Программирование
sqlite> INSERT INTO site (id_user, id_rubr, url, title, msg)
   ...> VALUES (1, 1, 'http://wwwadmin.ru', 'Название', 'Описание');
```

В первом примере перечислены только поля `email` и `passw`. Так как поле `id_user` не указано, то ему присваивается значение по умолчанию. В таблице `user` поле `id_user` объявлено как первичный ключ, поэтому будет вставлено значение на единицу большее максимального значения в поле. Такого же эффекта можно достичь, если в качестве значения передать `NULL`. Это демонстрируется во втором примере. В третьем примере вставляется запись в таблицу `site`. Поля `id_user` и `id_rubr` в этой таблице должны содержать идентификаторы соответствующих записей из таблиц `user` и `rubr`. Поэтому вначале мы делаем запросы на выборку данных и смотрим, какой идентификатор был присвоен вставленным записям в таблицы `user` и `rubr`. Обратите внимание на то, что мы опять указываем названия полей явным образом. Хотя перечислять поля и необязательно, но лучше всегда так делать. В этом случае в дальнейшем можно будет изменить структуру таблицы (например, добавить поле) без необходимости изменять все SQL-запросы. Достаточно для нового поля указать значение по умолчанию и все старые запросы будут по-прежнему рабочими.

Во всех этих примерах строковые значения указываются внутри одинарных кавычек. Однако бывают ситуации, когда внутри строки уже содержится одинарная кавычка. Попытка вставить такую строку приведет к ошибке:

```
sqlite> INSERT INTO rubr VALUES (NULL, 'Название 'в кавычках'');
Error: near "в": syntax error
```

Чтобы избежать этой ошибки, можно заключить строку в двойные кавычки или удвоить каждую одинарную кавычку внутри строки:

```
sqlite> INSERT INTO rubr VALUES (NULL, "Название 'в кавычках'");
sqlite> INSERT INTO rubr VALUES (NULL, 'Название ''в кавычках''');
sqlite> SELECT * FROM rubr;
1|Программирование
2|Название 'в кавычках'
3|Название 'в кавычках'
```

Если предпринимается попытка вставить запись, а в таблице уже есть запись с таким же значением первичного ключа (или значение индекса UNIQUE не уникально), то такая SQL-команда приводит к ошибке. Если необходимо, чтобы такие неуникальные записи обновлялись без вывода сообщения об ошибке, можно указать алгоритм обработки ошибок REPLACE после ключевого слова OR. Заменим название рубрики с идентификатором 2:

```
sqlite> INSERT OR REPLACE INTO rubr
   ...> VALUES (2, 'Музыка');
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Название 'в кавычках'
```

Вместо алгоритма REPLACE можно использовать инструкцию REPLACE INTO. Инструкция имеет следующий формат:

```
REPLACE INTO [<Название базы данных>.]<Название таблицы>
[(<Поле1>, <Поле2>, ...)] VALUES (<Значение1>, <Значение2>, ...);
```

Заменим название рубрики с идентификатором 3:

```
sqlite> REPLACE INTO rubr VALUES (3, 'Игры');
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Игры
```

16.4. Обновление и удаление записей

Обновление записи осуществляется с помощью инструкции UPDATE. Формат инструкции:

```
UPDATE [OR <Алгоритм>] [<Название базы данных>.]<Название таблицы>
SET <Поле1>='<Значение>', <Поле2>='<Значение2>', ...
[WHERE <Условие>];
```

Необязательный параметр OR <Алгоритм> задает алгоритм обработки ошибок (ROLLBACK, ABORT, FAIL, IGNORE или REPLACE). Все эти алгоритмы мы уже рассматривали при изучении создания таблицы. После ключевого слова SET указываются названия полей и их новые значения после знака равенства. Чтобы ограничить на-

бор изменяемых записей применяется инструкция WHERE. Обратите внимание на то, что если не указано <Условие>, то будут обновлены все записи в таблице. Какие выражения можно указать в параметре <Условие> мы рассмотрим немного позже.

В качестве примера изменим название рубрики с идентификатором 3:

```
sqlite> UPDATE rubr SET name_rubr='Кино' WHERE id_rubr=3;
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Кино
```

Удаление записи осуществляется с помощью инструкции DELETE. Формат инструкции:

```
DELETE FROM [<Название базы данных>.]<Название таблицы>
[WHERE <Условие>];
```

Если условие не указано, то будут удалены все записи из таблицы. В противном случае удаляются только записи, соответствующие условию. В качестве примера удалим рубрику с идентификатором 3:

```
sqlite> DELETE FROM rubr WHERE id_rubr=3;
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
```

Частое обновление и удаление записей приводит к дефрагментации таблицы. Чтобы освободить неиспользуемое пространство, можно воспользоваться SQL-командой VACUUM. Обратите внимание на то, что SQL-команда может изменить порядок нумерации в специальных полях (ROWID, OID и _ROWID_).

16.5. Изменение свойств таблицы

В некоторых случаях необходимо изменить структуру уже созданной таблицы. Для этого используется инструкция ALTER TABLE. В SQLite инструкция ALTER TABLE позволяет выполнить лишь ограниченное количество операций. Например, нельзя изменить свойство поля или удалить его из таблицы. Формат инструкции:

```
ALTER TABLE [<Название базы данных>.]<Название таблицы>
<Преобразование>;
```

В параметре <Преобразование> могут быть указаны следующие конструкции:

- RENAME TO <Новое имя таблицы> — переименовывает таблицу. Изменим название таблицы user на users:

```
sqlite> .tables
rubr            sqlite_sequence  tmp1            user
site            table            tmp2
sqlite> ALTER TABLE user RENAME TO users;
sqlite> .tables
rubr            sqlite_sequence  tmp1            users
site            table            tmp2
```

- `ADD [COLUMN] <Имя нового поля> [<Тип данных>] [<Опции>]` — добавляет новое поле после всех имеющихся полей. Обратите внимание на то, в новом поле нужно задать значение по умолчанию или значение `NULL` должно быть допустимым, т. к. в таблице уже есть записи. Кроме того, поле не может быть объявлено как `PRIMARY KEY` или `UNIQUE`. Добавим поле `iq` в таблицу `site`:

```
sqlite> ALTER TABLE site ADD COLUMN iq INTEGER DEFAULT 0;
sqlite> PRAGMA table_info(site);
0|id_site|INTEGER|0||1
1|id_user|INTEGER|0||0
2|id_rubr|INTEGER|0||0
3|url|TEXT|0||0
4|title|TEXT|0||0
5|msg|TEXT|0||0
6|iq|INTEGER|0|0|0
sqlite> SELECT * FROM site;
1|1|1|http://wwwadmin.ru|Название|Описание|0
```

ВНИМАНИЕ!

При использовании SQLite версии 3.1.3 и ниже после добавления нового поля необходимо выполнить инструкцию `VACUUM`.

16.6. Выбор записей

Для извлечения данных из таблицы предназначена инструкция `SELECT`. Инструкция имеет следующий формат:

```
SELECT [ALL | DISTINCT]
[<Название таблицы>.]<Поле>[, ...]
[ FROM <Название таблицы> [AS <Псевдоним>][, ...] ]
[ WHERE <Условие> ]
[ [ GROUP BY <Название поля> ] [ HAVING <Условие> ] ]
[ ORDER BY <Название поля> [COLLATE BINARY | NOCASE] [ASC | DESC][, ...] ]
[ LIMIT <Ограничение> ]
```

SQL-команда `SELECT` ищет все записи в указанной таблице, которые удовлетворяют условию в инструкции `WHERE`. Если инструкция `WHERE` не указана, то будут возвращены все записи из таблицы. Получим все записи из таблицы `rubr`:

```
sqlite> SELECT id_rubr, name_rubr FROM rubr;
1|Программирование
2|Музыка
```

Теперь выведем только запись с идентификатором 1:

```
sqlite> SELECT id_rubr, name_rubr FROM rubr WHERE id_rubr=1;
1|Программирование
```

Вместо перечисления полей можно указать символ `*`. В этом случае будут возвращены значения всех полей. Получим все записи из таблицы `rubr`:

```
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
```

SQL-команда `SELECT` позволяет вместо перечисления полей указать выражение. Это выражение будет вычислено, и возвращен результат:

```
sqlite> SELECT 10 + 5;
15
```

Чтобы из программы было легче обратиться к результату выполнения выражения, можно назначить псевдоним, указав его после выражения через ключевое слово `AS`:

```
sqlite> SELECT (10 + 5) AS expr1, (70 * 2) AS expr2;
15|140
```

Псевдоним можно назначить также таблицам. Это особенно полезно при выборке из нескольких таблиц сразу. В качестве примера заменим индекс рубрики в таблице `site` на соответствующее название из таблицы `rubr`:

```
sqlite> SELECT s.url, r.name_rubr FROM site AS s, rubr AS r
   ...> WHERE s.id_rubr = r.id_rubr;
http://wwwadmin.ru|Программирование
```

В этом примере мы назначили псевдонимы сразу двум таблицам. Теперь при указании списка полей достаточно задать псевдоним перед названием поля через точку, а не указывать полные названия таблиц. Более подробно выбор записей сразу из нескольких таблиц мы рассмотрим в следующем разделе.

После ключевого слова `SELECT` можно указать слово `ALL` или `DISTINCT`. Слово `ALL` является значением по умолчанию и означает, что возвращаются все записи. Если указано слово `DISTINCT`, то в результат попадут только уникальные значения.

Инструкция `GROUP BY` позволяет сгруппировать несколько записей. Эта инструкция особенно полезна при использовании агрегатных функций. В качестве примера добавим одну рубрику и два сайта:

```
sqlite> INSERT INTO rubr VALUES (3, 'Поисковые порталы');
sqlite> INSERT INTO site (id_user, id_rubr, url, title, msg, iq)
   ...> VALUES (1, 1, 'http://python.org', 'Python', '', 1000);
sqlite> INSERT INTO site (id_user, id_rubr, url, title, msg, iq)
   ...> VALUES (1, 3, 'http://google.ru', 'Гугль', '', 3000);
```

Теперь выведем количество сайтов в каждой рубрике:

```
sqlite> SELECT id_rubr, COUNT(id_rubr) FROM site
   ...> GROUP BY id_rubr;
1|2
3|1
```

Если необходимо ограничить сгруппированный набор записей, то следует воспользоваться инструкцией `HAVING`. Эта инструкция выполняет те же функции, что и

инструкция WHERE, но только для сгруппированного набора. В качестве примера выведем номера рубрик, в которых зарегистрировано более одного сайта:

```
sqlite> SELECT id_rubr FROM site
   ...> GROUP BY id_rubr HAVING COUNT(id_rubr)>1;
1
```

В этих примерах мы воспользовались агрегатной функцией COUNT(), которая возвращает количество записей. Перечислим агрегатные функции, используемые наиболее часто:

- COUNT(<Поле> | *) — количество записей в указанном поле. Выведем количество зарегистрированных сайтов:

  ```
  sqlite> SELECT COUNT(*) FROM site;
  3
  ```

- MIN(<Поле>) — минимальное значение в указанном поле. Выведем минимальный коэффициент релевантности:

  ```
  sqlite> SELECT MIN(iq) FROM site;
  0
  ```

- MAX(<Поле>) — максимальное значение в указанном поле. Выведем максимальный коэффициент релевантности:

  ```
  sqlite> SELECT MAX(iq) FROM site;
  3000
  ```

- AVG(<Поле>) — средняя величина значений в указанном поле. Выведем среднее значение коэффициента релевантности:

  ```
  sqlite> SELECT AVG(iq) FROM site;
  1333.33333333333
  ```

- SUM(<Поле>) — сумма значений в указанном поле. Выведем сумму значений коэффициентов релевантности:

  ```
  sqlite> SELECT SUM(iq) FROM site;
  4000
  ```

Найденные записи можно отсортировать с помощью инструкции ORDER BY. Допустимо производить сортировку сразу по нескольким полям. По умолчанию записи сортируются по возрастанию (значение ASC). Если в конце указано слово DESC, то записи будут отсортированы в обратном порядке. После ключевого слова COLLATE может быть указана функция сравнения (BINARY или NOCASE). Выведем названия рубрик по возрастанию и убыванию:

```
sqlite> SELECT * FROM rubr ORDER BY name_rubr;
2|Музыка
3|Поисковые порталы
1|Программирование
sqlite> SELECT * FROM rubr ORDER BY name_rubr DESC;
1|Программирование
3|Поисковые порталы
2|Музыка
```

Если требуется, чтобы при поиске выводились не все найденные записи, а лишь их часть, то следует использовать инструкцию `LIMIT`. Например, если таблица `site` содержит много описаний сайтов, то вместо того чтобы выводить все сайты за один раз, можно выводить их частями, скажем, по 10 сайтов за раз. Инструкция имеет следующие форматы:

```
LIMIT <Количество записей>
LIMIT <Начальная позиция>, <Количество записей>
LIMIT <Количество записей> OFFSET <Начальная позиция>
```

Первый формат задает количество записей от начальной позиции. Обратите внимание на то, что начальная позиция имеет индекс 0. Второй и третий форматы позволяют явно указать начальную позицию и количество записей. Пример:

```
sqlite> CREATE TEMP TABLE tmp3 (id INTEGER);
sqlite> INSERT INTO tmp3 VALUES(1);
sqlite> INSERT INTO tmp3 VALUES(2);
sqlite> INSERT INTO tmp3 VALUES(3);
sqlite> INSERT INTO tmp3 VALUES(4);
sqlite> INSERT INTO tmp3 VALUES(5);
sqlite> SELECT * FROM tmp3 LIMIT 3; -- Эквивалентно LIMIT 0, 3
1
2
3
sqlite> SELECT * FROM tmp3 LIMIT 2, 3;
3
4
5
sqlite> SELECT * FROM tmp3 LIMIT 3 OFFSET 2;
3
4
5
sqlite> DROP TABLE tmp3;
```

16.7. Выбор записей из нескольких таблиц

SQL-команда `SELECT` позволяет выбирать записи сразу из нескольких таблиц одновременно. Для этого используются следующие форматы инструкции `FROM`:

```
FROM <Название таблицы1> [AS <Псевдоним>]
, | [NATURAL] [LEFT] [OUTER | INNER | CROSS] JOIN
<Название таблицы2> [AS <Псевдоним>]
[ON <Выражение>] [USING ( <Поле> )]
```

В первом формате таблицы перечисляются через запятую в инструкции `FROM`, а в инструкции `WHERE` через запятую указываются пары полей, являющиеся связуемыми для таблиц. Причем в условии и перечислении полей вначале указывается

название таблицы (или псевдоним), а затем через точку название поля. В качестве примера выведем сайты из таблицы site, но вместо индекса пользователя укажем его e-mail, а вместо индекса рубрики ее название:

```
sqlite> SELECT site.url, rubr.name_rubr, users.email
   ...> FROM rubr, users, site
   ...> WHERE site.id_rubr=rubr.id_rubr AND
   ...> site.id_user=users.id_user;
http://wwwadmin.ru|Программирование|unicross@mail.ru
http://python.org|Программирование|unicross@mail.ru
http://google.ru|Поисковые порталы|unicross@mail.ru
```

Вместо названия таблиц можно использовать псевдоним. Кроме того, если поля в таблицах имеют разные названия, то название таблицы можно не указывать:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr AS r, users AS u, site AS s
   ...> WHERE s.id_rubr=r.id_rubr AND
   ...> s.id_user=u.id_user;
```

Объединить таблицы позволяет также оператор JOIN, который имеет два синонима: CROSS JOIN и INNER JOIN. Переделаем наш предыдущий пример и используем оператор JOIN:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr JOIN users JOIN site
   ...> WHERE site.id_rubr=rubr.id_rubr AND
   ...> site.id_user=users.id_user;
```

Инструкцию WHERE можно заменить инструкцией ON, а в инструкции WHERE указать дополнительное условие. В качестве примера выведем сайты, зарегистрированные в рубрике с идентификатором 1:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr JOIN users JOIN site
   ...> ON site.id_rubr=rubr.id_rubr AND
   ...> site.id_user=users.id_user
   ...> WHERE site.id_rubr=1;
```

Если названия связующих полей в таблицах являются одинаковыми, то вместо инструкции ON можно использовать инструкцию USING:

```
sqlite> SELECT url, name_rubr, email
   ...> FROM rubr JOIN site USING (id_rubr) JOIN users USING (id_user);
```

Оператор JOIN объединяет все записи, которые существуют во всех связующих полях. Например, если попробовать вывести количество сайтов в каждой рубрике, то мы не получим рубрики без зарегистрированных сайтов:

```
sqlite> SELECT name_rubr, COUNT(id_site)
   ...> FROM rubr JOIN site USING (id_rubr)
   ...> GROUP BY rubr.id_rubr;
Программирование|2
Поисковые порталы|1
```

В этом примере мы не получили количество сайтов в рубрике "Музыка", т. к. в этой рубрике нет сайтов. Чтобы получить количество сайтов во всех рубриках, необходимо использовать левостороннее объединение. Формат левостороннего объединения:

```
<Таблица1> LEFT [OUTER] JOIN <Таблица2>
ON <Таблица1>.<Поле1>=<Таблица2>.<Поле2> | USING (<Поле>)
```

При левостороннем объединении возвращаются записи, соответствующие условию, а также записи из таблицы `<Таблица1>`, которым нет соответствия в таблице `<Таблица2>` (при этом поля из таблицы `<Таблица2>` будут иметь значение `NULL`). Выведем количество сайтов в рубриках и отсортируем по названию рубрики:

```
sqlite> SELECT name_rubr, COUNT(id_site)
   ...> FROM rubr LEFT JOIN site USING (id_rubr)
   ...> GROUP BY rubr.id_rubr
   ...> ORDER BY rubr.name_rubr;
Музыка|0
Поисковые порталы|1
Программирование|2
```

16.8. Условия в инструкции *WHERE*

В предыдущих разделах мы оставили без внимания рассмотрение выражений в инструкциях `WHERE` и `HAVING`. Эти инструкции позволяют ограничить набор выводимых, изменяемых или удаляемых записей с помощью некоторого условия. Внутри условий можно использовать следующие операторы сравнения:

- `=` или `==` — проверка на равенство. Пример:

  ```
  sqlite> SELECT * FROM rubr WHERE id_rubr=1;
  1|Программирование
  sqlite> SELECT 10 = 10, 5 = 10, 10 == 10, 5 == 10;
  1|0|1|0
  ```

  Как видно из примера, выражения можно разместить не только в инструкциях `WHERE` и `HAVING`, но и после ключевого слова `SELECT`. В этом случае результатом операции сравнения являются следующие значения:

  - `0` — ложь;
  - `1` — истина;
  - `NULL`.

  Результат сравнения двух строк зависит от используемой функции сравнения. Задать функцию можно при создании таблицы с помощью инструкции `COLLATE <Функция>`. В параметре `<Функция>` указывается функция `BINARY` (значение по умолчанию), `NOCASE` (без учета регистра) или `RTRIM`. Пример:

  ```
  sqlite> CREATE TEMP TABLE tmp3 (p1, p2 TEXT COLLATE NOCASE);
  sqlite> INSERT INTO tmp3 VALUES ("abcd", "abcd");
  ```

```
sqlite> SELECT p1 = "ABCD" FROM tmp3; -- Не найдено
0
sqlite> SELECT p2 = "ABCD" FROM tmp3; -- Найдено
1
sqlite> DROP TABLE tmp3;
```

Указать функцию сравнения можно также после выражения:

```
sqlite> SELECT 's' = 'S', 's' = 'S' COLLATE NOCASE;
0|1
```

Функция NOCASE не учитывает регистр только латинских букв. При использовании русских букв возможны проблемы с регистром. Пример:

```
sqlite> SELECT 'ы' = 'Ы', 'ы' = 'Ы' COLLATE NOCASE;
0|0
```

- != или <> — не равно:

```
sqlite> SELECT 10 != 10, 5 != 10, 10 <> 10, 5 <> 10;
0|1|0|1
```

- < — меньше;
- > — больше;
- <= — меньше или равно;
- >= — больше или равно;
- IS NOT NULL, NOT NULL или NOTNULL — проверка на наличие значения;
- IS NULL или ISNULL — проверка на отсутствие значения;
- BETWEEN <Начало> AND <Конец> — проверка на вхождение в диапазон значений. Пример:

```
sqlite> SELECT 100 BETWEEN 1 AND 100;
1
sqlite> SELECT 101 BETWEEN 1 AND 100;
0
```

- IN (<Список значений>) — проверка на наличие значения в определенном наборе. Сравнение зависит от регистра букв. Пример:

```
sqlite> SELECT 'один' IN ('один', 'два', 'три');
1
sqlite> SELECT 'Один' IN ('один', 'два', 'три');
0
```

- LIKE <Шаблон> [ESCAPE <Символ>] — проверка на соответствие шаблону. В шаблоне используются следующие специальные символы:
  - % — любое количество символов;
  - _ — любой одиночный символ.

  Специальные символы могут быть расположены в любом месте шаблона. Например, чтобы найти все вхождения, необходимо указать символ % в начале и в конце шаблона:

```
sqlite> SELECT 'test word test' LIKE '%word%';
1
```

Можно установить привязку или только к началу строки, или только к концу:

```
sqlite> SELECT 'test word test' LIKE 'test%';
1
sqlite> SELECT 'test word test' LIKE 'word%';
0
```

Кроме того, шаблон для поиска может иметь очень сложную структуру:

```
sqlite> SELECT 'test word test' LIKE '%es_%wo_d%';
1
sqlite> SELECT 'test word test' LIKE '%wor%d%';
1
```

Обратите внимание на последнюю строку поиска. Этот пример демонстрирует, что специальный символ `%` соответствует не только любому количеству символов, но и полному их отсутствию.

Что же делать, если необходимо найти символы `%` и _? Ведь они являются специальными. В этом случае специальные символы необходимо экранировать с помощью символа, указанного в инструкции `ESCAPE <Символ>`:

```
sqlite> SELECT '10$' LIKE '10%';
1
sqlite> SELECT '10$' LIKE '10\%' ESCAPE '\';
0
sqlite> SELECT '10%' LIKE '10\%' ESCAPE '\';
1
```

Следует учитывать, что сравнение с шаблоном для латинских букв производится без учета регистра символов. Чтобы учитывался регистр, необходимо присвоить значение `true` (или `1`, `yes`, `on`) параметру `case_sensitive_like` в SQL-команде `PRAGMA`. Пример:

```
sqlite> PRAGMA case_sensitive_like = true;
sqlite> SELECT 's' LIKE 'S';
0
sqlite> PRAGMA case_sensitive_like = false;
sqlite> SELECT 's' LIKE 'S';
1
```

Теперь посмотрим, учитывается ли регистр русских букв при поиске по шаблону:

```
sqlite> SELECT 'ы' LIKE 'Ы', 'ы' LIKE 'ы';
1|1
```

Результат выполнения примера показывает, что поиск производится без учета регистра. Однако это далеко не так. Попробуем сравнить две разные буквы и два разных слова:

```
sqlite> SELECT 'г' LIKE 'Ы', 'слово' LIKE 'текст';
1|1
```

Этот пример показывает, что буква "г" равна букве "Ы", а "слово" равно "текст". Иными словами, производится сравнение длины строк, а не символов в строке. Такой странный результат был получен при использовании кодировки Windows-1251. Если изменить кодировку на ср866, то результат выполнения примера будет другим:

```
C:\book>chcp 866
Текущая кодовая страница: 866

C:\book>sqlite3.exe testdb.db
SQLite version 3.6.23
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT 'г' LIKE 'Ы', 'слово' LIKE 'текст';
0|0
sqlite> SELECT 'ы' LIKE 'Ы', 'ы' LIKE 'ы';
0|1
```

Результат выполнения становится более логичным. Таким образом, поиск русских букв зависит от кодировки. По умолчанию в SQLite используется кодировка UTF-8. С помощью инструкции `PRAGMA encoding = <Кодировка>` можно указать другую кодировку. Поддерживаются кодировки UTF-8, UTF-16, UTF-16le и UTF-16be. В этот список не входят кодировки ср866 и Windows-1251, поэтому результат сравнения строк может быть некорректным. С кодировкой UTF-8 мы еще поработаем в следующей главе, а на данный момент следует запомнить, что результат сравнения русских букв зависит от регистра символов. Кроме того, если поиск сравнивает только длину строк, то необходимо проверить кодировку данных. В рабочих проектах данные должны быть в кодировке UTF-8.

Результат логического выражения можно изменить на противоположный. Для этого необходимо перед выражением разместить оператор `NOT`. Пример:

```
sqlite> SELECT 's' = 'S', NOT ('s' = 'S');
0|1
sqlite> SELECT NOT 'один' IN ('один', 'два', 'три');
0
```

Кроме того, допустимо проверять сразу несколько условий, указав между выражениями следующие операторы:

- `AND` — логическое И;
- `OR` — логическое ИЛИ.

16.9. Индексы

Все записи в полях таблицы расположены в произвольном порядке. Чтобы найти какие-либо данные, необходимо каждый раз просматривать все записи. Для ус-

корения выполнения запросов применяются *индексы (ключи)*. Индексированные поля всегда поддерживаются в отсортированном состоянии, что позволяет быстро найти необходимую запись, не просматривая все записи. Надо сразу заметить, что применение индексов приводит к увеличению размера базы данных, а также к затратам времени на поддержание индекса в отсортированном состоянии при каждом добавлении данных. По этой причине индексировать следует поля, которые очень часто используются в запросах типа:

```
SELECT <Список полей> FROM <Таблица> WHERE <Поле>=<Значение>;
```

В SQLite существуют следующие виды индексов:

- первичный ключ;
- уникальный индекс;
- обычный индекс.

Первичный ключ служит для однозначной идентификации каждой записи в таблице. Для создания индекса в инструкции `CREATE TABLE` используется ключевое слово `PRIMARY KEY`. Ключевое слово можно указать после описания поля или после перечисления всех полей. Второй вариант позволяет указать сразу несколько полей в качестве первичного ключа.

Посмотреть, каким образом будет выполняться запрос и какие индексы будут использоваться, позволяет SQL-команда `EXPLAIN`. Формат SQL-команды:

```
EXPLAIN [QUERY PLAN] <SQL-запрос>
```

Если ключевые слова `QUERY PLAN` не указаны, то выводится полный список параметров и их значений. Если ключевые слова указаны, то выводится информация об используемых индексах. В качестве примера попробуем выполнить запрос на извлечение записей из таблицы `site`. В первом случае поиск произведем в поле, являющемся первичным ключом, а во втором случае — в обычном поле:

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_site=1;
0|0|TABLE site USING PRIMARY KEY
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|TABLE site
```

В первом случае фраза "USING PRIMARY KEY" означает, что при поиске будет использован первичный ключ, а во втором случае никакие индексы не используются.

В одной таблице не может быть более одного первичного ключа. А вот обычных и уникальных индексов допускается создать несколько. Для создания индекса применяется SQL-команда `CREATE INDEX`. Формат команды:

```
CREATE [UNIQUE] INDEX [IF NOT EXISTS]
[<Название базы данных>.]<Название индекса>
ON <Название таблицы>
(<Название поля> [ COLLATE <Функция сравнения>] [ ASC | DESC ][, ...])
```

Если между ключевыми словами `CREATE` и `INDEX` указано слово `UNIQUE`, то создается уникальный индекс. В этом случае дублирование данных в поле не допускается. Если слово `UNIQUE` не указано, то создается обычный индекс.

Все сайты в нашем каталоге распределяются по рубрикам. Это означает, что при выводе сайтов, зарегистрированных в определенной рубрике, в инструкции WHERE будет постоянно выполняться условие:

```
WHERE id_rubr=<Номер рубрики>
```

Чтобы ускорить выборку сайтов по номеру рубрики, создадим обычный индекс для этого поля и проверим с помощью SQL-команды EXPLAIN, задействуется ли этот индекс:

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|TABLE site
sqlite> CREATE INDEX index_rubr ON site (id_rubr);
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|TABLE site WITH INDEX index_rubr
```

Обратите внимание на то, что после создания индекса добавилась фраза "WITH INDEX index_rubr". Это означает, что теперь при поиске будет задействован индекс и поиск будет выполняться быстрее. При выполнении запроса название индекса явным образом указывать нет необходимости. Использовать индекс или нет, SQLite решает самостоятельно. Таким образом, SQL-запрос будет выглядеть обычным образом:

```
sqlite> SELECT * FROM site WHERE id_rubr=1;
1|1|1|http://wwwadmin.ru|Название|Описание|0
2|1|1|http://python.org|Python||1000
```

В некоторых случаях необходимо пересоздать индексы. Для этого применяется SQL-команда REINDEX. Формат команды:

```
REINDEX [<Название базы данных>.]<Название таблицы или индекса>
```

Если указано название таблицы, то пересоздаются все существующие индексы в таблице. При задании названия индекса пересоздается только указанный индекс.

Удалить обычный и уникальный индексы позволяет SQL-команда DROP INDEX. Формат команды:

```
DROP INDEX [IF EXISTS] [<Название базы данных>.]<Название индекса>
```

Удаление индекса приводит к дефрагментации файла с базой данных. Чтобы освободить неиспользуемое свободное пространство, можно воспользоваться SQL-командой VACUUM.

Вся статистическая информация об индексах хранится в специальной таблице sqlite_stat1. В данный момент в таблице нет никакой информации. Чтобы собрать статистическую информацию и поместить ее в эту таблицу, предназначена SQL-команда ANALYZE. Формат команды:

```
ANALYZE [[<Название базы данных>.]<Название таблицы>];
```

Выполним SQL-команду ANALYZE и выведем содержимое таблицы sqlite_stat1:

```
sqlite> SELECT * FROM sqlite_stat1; -- Нет записей
Error: no such table: sqlite_stat1
sqlite> ANALYZE;
sqlite> SELECT * FROM sqlite_stat1;
site|index_rubr|3 2
```

16.10. Вложенные запросы

Результаты выполнения инструкции SELECT можно использовать в других инструкциях, создавая вложенные запросы. Для создания таблицы с помощью вложенного запроса используется следующий формат:

```
CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS]
[<Название базы данных>.]<Название таблицы> AS <Запрос SELECT>;
```

В качестве примера создадим временную копию таблицы rubr и выведем ее содержимое:

```
sqlite> CREATE TEMP TABLE tmp_rubr AS SELECT * FROM rubr;
sqlite> SELECT * FROM tmp_rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
```

В результате выполнения вложенного запроса создается таблица с полями, перечисленными после ключевого слова SELECT, и сразу заполняется данными.

Использовать вложенные запросы можно и в инструкции INSERT. Для этого предназначен следующий формат:

```
INSERT [OR <Алгоритм>] INTO [<Название базы данных>.]<Название таблицы>
[(<Поле1>, <Поле2>, ...)] <Запрос SELECT>;
```

Очистим временную таблицу tmp_rubr, а затем опять заполним ее с помощью вложенного запроса:

```
sqlite> DELETE FROM tmp_rubr;
sqlite> INSERT INTO tmp_rubr SELECT * FROM rubr WHERE id_rubr<3;
sqlite> SELECT * FROM tmp_rubr;
1|Программирование
2|Музыка
```

Если производится попытка вставить повторяющееся значение и не указан <Алгоритм>, то это приведет к ошибке. С помощью алгоритмов ROLLBACK, ABORT, FAIL, IGNORE или REPLACE можно указать, как следует обрабатывать записи с дублированными значениями. При использовании алгоритма IGNORE повторяющиеся записи отбрасываются, а при использовании REPLACE — новые записи заменяют существующие.

Использовать вложенные запросы можно также в инструкции WHERE. В этом случае вложенный запрос размещается в операторе IN. Для примера выведем сайты, зарегистрированные в рубрике с названием "Программирование":

```
sqlite> SELECT * FROM site WHERE id_rubr IN (
   ...> SELECT id_rubr FROM rubr
   ...> WHERE name_rubr='Программирование');
1|1|1|http://wwwadmin.ru|Название|Описание|0
2|1|1|http://python.org|Python||1000
```

16.11. Транзакции

Очень часто несколько инструкций выполняются последовательно. Например, при совершении покупки деньги списываются со счета клиента и сразу добавляются на счет магазина. Если во время добавления денег на счет магазина произойдет ошибка, то деньги будут списаны со счета клиента, но не попадут на счет магазина. Чтобы гарантировать успешное выполнение группы инструкций, предназначены транзакции. После запуска транзакции группа инструкций выполняется как единое целое. Если во время транзакции произойдет ошибка, например, отключится компьютер, все операции с начала транзакции будут отменены.

В SQLite каждая инструкция, производящая изменения в базе данных, автоматически запускает транзакцию, если транзакция не была запущена ранее. После завершения выполнения инструкции транзакция автоматически завершается. Для явного запуска транзакции предназначена инструкция `BEGIN`. Формат инструкции:

```
BEGIN [DEFERRED | IMMEDIATE | EXCLUSIVE] [TRANSACTION];
```

Для нормального завершения транзакции предназначены инструкции `COMMIT` и `END`. Эти инструкции сохраняют все изменения и завершают транзакцию. Инструкции имеют следующий формат:

```
COMMIT [TRANSACTION];
END [TRANSACTION];
```

Чтобы отменить изменения, выполненные с начала транзакции, используется инструкция `ROLLBACK`. Формат инструкции:

```
ROLLBACK [TRANSACTION] [TO [SAVEPOINT] <Название метки>];
```

В качестве примера запустим транзакцию, вставим две записи, а затем отменим все произведенные изменения и выведем содержимое таблицы:

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO rubr VALUES (NULL, 'Кино');
sqlite> INSERT INTO rubr VALUES (NULL, 'Разное');
sqlite> ROLLBACK TRANSACTION; -- Отменяем вставку
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
```

Как видно из результата, новые записи не были вставлены в таблицу. Аналогичные действия будут выполнены автоматически, если соединение с базой данных будет закрыто или отключится компьютер.

Если ошибка возникает в одной из инструкций внутри транзакции, то используется алгоритм обработки ошибок, указанный в конструкции `ON CONFLICT <Алгоритм>` при создании таблицы, или в конструкции `OR <Алгоритм>` при вставке или обновлении записей. По умолчанию используется алгоритм `ABORT`. Согласно этому алгоритму при возникновении ошибки аннулируются все изменения, произведенные текущей командой, и выводится сообщение об ошибке. Все изменения, сде-

ланные предыдущими командами в транзакции, сохраняются. Запустим транзакцию и попробуем вставить две записи. При вставке второй записи укажем индекс, который уже существует в таблице:

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT INTO rubr VALUES (NULL, 'Кино');
sqlite> INSERT INTO rubr VALUES (3, 'Разное'); -- Ошибка
Error: PRIMARY KEY must be unique
sqlite> COMMIT TRANSACTION;
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

Как видно из примера, первая запись успешно добавлена в таблицу. Если необходимо отменить все изменения внутри транзакции, то при вставке следует указать алгоритм `ROLLBACK`. Согласно этому алгоритму при ошибке транзакция завершается с откатом всех измененных ранее записей, дальнейшее выполнение прерывается и выводится сообщение об ошибке. Рассмотрим это на примере:

```
sqlite> BEGIN TRANSACTION;
sqlite> INSERT OR ROLLBACK INTO rubr VALUES (NULL, 'Мода');
sqlite> INSERT OR ROLLBACK INTO rubr VALUES (3, 'Разное');
Error: PRIMARY KEY must be unique
sqlite> COMMIT TRANSACTION; -- Транзакция уже завершена!
Error: cannot commit - no transaction is active
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

Вместо запуска транзакции с помощью инструкции `BEGIN` можно создать именованную метку. Метка создается с помощью инструкции `SAVEPOINT`. Формат инструкции:

```
SAVEPOINT <Название метки>;
```

Для нормального завершения транзакции и сохранения всех изменений предназначена инструкция `RELEASE`. Формат инструкции:

```
RELEASE [SAVEPOINT] <Название метки>;
```

Чтобы отменить изменения, выполненные после метки, используется инструкция `ROLLBACK`. В качестве примера запустим транзакцию, вставим две записи, а затем отменим все произведенные изменения и выведем содержимое таблицы:

```
sqlite> SAVEPOINT metka1;
sqlite> INSERT INTO rubr VALUES (NULL, 'Мода');
sqlite> INSERT INTO rubr VALUES (NULL, 'Разное');
sqlite> ROLLBACK TO SAVEPOINT metka1;
```

```
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

16.12. Удаление таблицы и базы данных

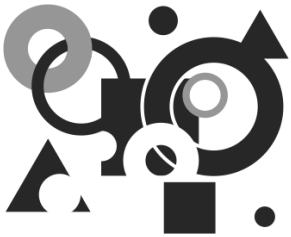
Удалить таблицу позволяет инструкция `DROP TABLE`. Удалить можно как обычную таблицу, так и временную. Все индексы и триггеры, связанные с таблицей, также удаляются. Формат инструкции:

```
DROP TABLE [IF EXISTS] [<Название базы данных>.]<Название таблицы>;
```

Так как SQLite напрямую работает с файлом, не существует инструкции для удаления базы данных. Чтобы удалить базу, достаточно просто удалить файл.

В этой главе мы рассмотрели лишь основные возможности SQLite. Остались не рассмотренными триггеры, представления, виртуальные таблицы, внешние ключи, операторы, встроенные функции и некоторые другие возможности. За подробной информацией обращайтесь к документации по SQLite.

Глава 17



Доступ к базе данных SQLite из Python

Итак, изучение основ SQLite закончено, и мы возвращаемся к изучению языка Python. В этой главе мы рассмотрим возможности модуля `sqlite3`, позволяющего работать с базой данных SQLite. Модуль `sqlite3` входит в состав стандартной библиотеки Python, начиная с версии 2.5, и в дополнительной установке не нуждается. Если необходимо получить доступ к SQLite в предыдущих версиях Python, то следует воспользоваться модулем `pysqlite`. Этот модуль не входит в состав стандартной библиотеки, поэтому его придется устанавливать отдельно.

Для работы с базами данных в языке Python существует единый интерфейс доступа. Все разработчики модулей, осуществляющих связь базы данных с Python, должны придерживаться спецификации DB-API (DataBase Application Program Interface). Эта спецификация более интересна для разработчиков модулей, чем для прикладных программистов, поэтому мы не будем ее подробно рассматривать. Получить полное описание спецификации DB-API 2.0 можно в документе PEP 249, расположенном по адресу **http://www.python.org/dev/peps/pep-0249**.

Модуль `sqlite3` поддерживает спецификацию DB-API 2.0, а также предоставляет некоторые нестандартные возможности. Поэтому, изучив методы и атрибуты этого модуля, вы получите достаточно полное представление о спецификации DB-API 2.0 и сможете в дальнейшем работать с другой базой данных. Получить номер спецификации, поддерживаемой модулем, можно с помощью атрибута `apilevel`:

```
>>> import sqlite3           # Подключаем модуль
>>> sqlite3.apilevel         # Получаем номер спецификации
'2.0'
```

Получить номер версии используемого модуля `sqlite3` можно с помощью атрибутов `sqlite_version` и `sqlite_version_info`. Атрибут `sqlite_version` возвращает номер версии в виде строки, а атрибут `sqlite_version_info` в виде кортежа из трех чисел. Пример:

```
>>> sqlite3.sqlite_version
'3.5.9'
```

```
>>> sqlite3.sqlite_version_info
(3, 5, 9)
```

Согласно спецификации DB-API 2.0 последовательность работы с базой данных выглядит следующим образом:

1. Производится подключение к базе данных с помощью функции `connect()`. Функция возвращает объект соединения, с помощью которого осуществляется дальнейшая работа с базой данных.
2. Создается объект-курсор.
3. Выполняются SQL-запросы и обрабатываются результаты. Перед выполнением первого запроса, который изменяет записи (`INSERT`, `REPLACE`, `UPDATE` и `DELETE`), автоматически запускается транзакция.
4. Завершается транзакция или отменяются все изменения в рамках транзакции.
5. Закрывается объект-курсор.
6. Закрывается соединение с базой данных.

17.1. Создание и открытие базы данных

Для создания и открытия базы данных используется функция `connect()`. Функция имеет следующий формат:

```
connect(database[, timeout][, isolation_level][, detect_types]
        [, factory])
```

В параметре `database` указывается абсолютный или относительный путь к базе данных. Если база данных не существует, то она будет создана и открыта для работы. Если база данных уже существует, то она просто открывается без удаления имеющихся данных. Вместо пути к базе данных можно указать значение `:memory:`, которое означает, что база данных будет создана в оперативной памяти. После закрытия такой базы все данные будут удалены.

Все остальные параметры являются необязательными и могут быть указаны в произвольном порядке путем присвоения значения названию параметра. Необязательный параметр `timeout` задает время ожидания снятия блокировки с открываемой базы данных. По умолчанию значение параметра `timeout` равно пяти секундам. Предназначение остальных параметров мы рассмотрим немного позже.

Функция `connect()` возвращает объект соединения, с помощью которого осуществляется вся дальнейшая работа с базой данных. Если открыть базу данных не удалось, то возбуждается исключение. Соединение закрывается, когда вызывается метод `close()` объекта соединения. В качестве примера откроем и сразу закроем базу данных testdb.db, расположенную в текущем рабочем каталоге:

```
>>> import sqlite3                          # Подключаем модуль sqlite3
>>> con = sqlite3.connect("testdb.db")      # Открываем базу данных
>>>                                         # Работаем с базой данных
>>> con.close()                             # Закрываем базу данных
```

17.2. Выполнение запроса

Согласно спецификации DB-API 2.0 после создания объекта соединения необходимо создать объект-курсор. Все дальнейшие запросы должны производиться через этот объект. Создание объекта-курсора производится с помощью метода `cursor()`. Для выполнения запроса к базе данных предназначены следующие методы объекта-курсора:

- `close()` — закрывает объект-курсор;
- `executescript(<SQL-запросы через точку с запятой>)` — выполняет несколько SQL-запросов за один раз. Если в процессе выполнения запросов возникает ошибка, то метод возбуждает исключение. В качестве примера создадим базу данных и три таблицы в ней:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()        # Создаем объект-курсор
sql = """\
CREATE TABLE user (
   id_user INTEGER PRIMARY KEY AUTOINCREMENT,
   email TEXT,
   passw TEXT
);
CREATE TABLE rubr (
   id_rubr INTEGER PRIMARY KEY AUTOINCREMENT,
   name_rubr TEXT
);
CREATE TABLE site (
   id_site INTEGER PRIMARY KEY AUTOINCREMENT,
   id_user INTEGER,
   id_rubr INTEGER,
   url TEXT,
   title TEXT,
   msg TEXT,
   iq INTEGER
);
"""
try:                        # Обрабатываем исключения
    cur.executescript(sql) # Выполняем SQL-запросы
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()                # Закрываем объект-курсор
con.close()                # Закрываем соединение
raw_input()
```

Сохраняем код в файл, а затем запускаем его с помощью двойного щелчка на значке файла. Обратите внимание на то, что мы работаем с кодировкой UTF-8. Эта кодировка по умолчанию используется в SQLite;

- `execute(<SQL-запрос>[, <Значения>])` — выполняет один SQL-запрос. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Добавим пользователя в таблицу `user`:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()        # Создаем объект-курсор
sql = """\
INSERT INTO user (email, passw)
VALUES ('unicross@mail.ru', 'password1')
"""
try:
    cur.execute(sql)      # Выполняем SQL-запрос
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
    con.commit()          # Завершаем транзакцию
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
raw_input()
```

В этом примере мы использовали метод `commit()` объекта соединения. Метод `commit()` позволяет завершить транзакцию, которая запускается автоматически. Если метод не вызвать и при этом закрыть соединение с базой данных, то все произведенные изменения будут автоматически отменены. Более подробно управление транзакциями мы рассмотрим далее в этой главе, а сейчас следует запомнить, что запросы, изменяющие записи (`INSERT`, `REPLACE`, `UPDATE` и `DELETE`), необходимо завершать вызовом метода `commit()`.

В некоторых случаях в SQL-запрос необходимо подставлять данные, полученные от пользователя. Если данные не обработать и подставить в SQL-запрос, то пользователь получает возможность видоизменить запрос и, например, зайти в закрытый раздел без ввода пароля. Чтобы значения были правильно подставлены, необходимо их передавать в виде кортежа или словаря во втором параметре метода `execute()`. В этом случае в SQL-запросе указываются следующие специальные заполнители:

- `?` — при указании значения в виде кортежа;
- `:<Ключ>` — при указании значения в виде словаря.

В качестве примера заполним таблицу с рубриками этими способами:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
```

```
cur = con.cursor()              # Создаем объект-курсор
t1 = (u"Программирование",)
t2 = (2, u"Музыка")
d  = {"id": 3, "name": u"""Поисковые ' " порталы"""}
sql_t1 = "INSERT INTO rubr (name_rubr) VALUES (?)"
sql_t2 = "INSERT INTO rubr VALUES (?, ?)"
sql_d  = "INSERT INTO rubr VALUES (:id, :name)"
try:
    cur.execute(sql_t1, t1)     # Кортеж из 1-го элемента
    cur.execute(sql_t2, t2)     # Кортеж из 2-х элементов
    cur.execute(sql_d,  d)      # Словарь
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
    con.commit()                # Завершаем транзакцию
cur.close()                     # Закрываем объект-курсор
con.close()                     # Закрываем соединение
raw_input()
```

Обратите внимание на значение переменной `t1`. Перед закрывающей круглой скобкой запятая указана не по ошибке. Если запятую убрать, то вместо кортежа мы получим строку. Не скобки создают кортеж, а запятые. Поэтому при создании кортежа из одного элемента в конце необходимо добавить запятую. Как показывает практика, новички постоянно забывают указать запятую и при этом получают сообщение об ошибке.

В значении ключа `name` переменной `d` апостроф и двойная кавычка также указаны не случайно. Это значение показывает, что при подстановке все специальные символы экранируются, поэтому никакой ошибки при вставке значения в таблицу не будет.

Внимание!

Никогда напрямую не передавайте в SQL-запрос данные, полученные от пользователя. Это потенциальная угроза безопасности. Данные следует передавать через второй параметр методов `execute()` и `executemany()`.

- `executemany(<SQL-запрос>, <Последовательность>)` — выполняет SQL-запрос несколько раз, при этом подставляя значения из последовательности. Каждый элемент последовательности должен быть кортежем (при использовании заполнителя `"?"`) или словарем (при использовании заполнителя `":<Ключ>"`). Вместо последовательности можно указать объект-итератор или объект-генератор. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Заполним таблицу `site` с помощью метода `executemany()`:

```
# -*- coding: utf-8 -*-
import sqlite3
```

```
    con = sqlite3.connect("catalog.db")
    cur = con.cursor()              # Создаем объект-курсор
    arr = [
             (1, 1, u"http://wwwadmin.ru", u"Название", u"", 100),
             (1, 1, u"http://python.org", u"Python", u"", 1000),
             (1, 3, u"http://google.ru", u"Гугль", u"", 3000)
          ]
    sql = """\
    INSERT INTO site (id_user, id_rubr, url, title, msg, iq)
    VALUES (?, ?, ?, ?, ?, ?)
    """
    try:
        cur.executemany(sql, arr)
    except sqlite3.DatabaseError, err:
        print u"Ошибка:", err
    else:
        print u"Запрос успешно выполнен"
        con.commit()                # Завершаем транзакцию
    cur.close()                     # Закрываем объект-курсор
    con.close()                     # Закрываем соединение
    raw_input()
```

Модуль `sqlite3` содержит также методы `execute()`, `executemany()` и `executescript()` объекта соединения, которые позволяют выполнить запрос без создания объекта-курсора. Эти методы не входят в спецификацию DB-API 2.0. В качестве примера изменим название рубрики с идентификатором `3` (листинг 17.1).

Листинг 17.1. Использование метода `execute()`

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
try:
    con.execute("""UPDATE rubr SET name_rubr='Поисковые порталы'
                   WHERE id_rubr=3""")
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    con.commit()                # Завершаем транзакцию
    print u"Запрос успешно выполнен"
con.close()                     # Закрываем соединение
raw_input()
```

Объект-курсор поддерживает несколько атрибутов:

- `lastrowid` — индекс последней добавленной записи с помощью инструкции `INSERT` и метода `execute()`. Если индекс не определен, то атрибут будет содер-

жать значение `None`. В качестве примера добавим новую рубрику и выведем ее индекс:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()                  # Создаем объект-курсор
try:
    cur.execute("""INSERT INTO rubr (name_rubr)
                   VALUES ('Кино')""")
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    con.commit()                    # Завершаем транзакцию
    print u"Запрос успешно выполнен"
    print u"Индекс:", cur.lastrowid
cur.close()                         # Закрываем объект-курсор
con.close()                         # Закрываем соединение
raw_input()
```

- `rowcount` — количество измененных или удаленных записей. Если количество не определено, то атрибут имеет значение `-1`;
- `description` — содержит кортеж кортежей с именами полей в результате выполнения инструкции `SELECT`. Каждый внутренний кортеж состоит из семи элементов. Первый элемент содержит название поля, а остальные элементы всегда имеют значение `None`. Например, если выполнить SQL-запрос `SELECT * FROM rubr`, то атрибут будет содержать следующее значение:

```
(('id_rubr', None, None, None, None, None, None),
 ('name_rubr', None, None, None, None, None, None))
```

17.3. Обработка результата запроса

Для обработки результата запроса применяются следующие методы объекта-курсора:

- `fetchone()` — при каждом вызове возвращает одну запись из результата запроса в виде кортежа, а затем перемещает указатель текущей позиции. Если записей больше нет, метод возвращает значение `None`. Выведем все записи из таблицы `user`:

```
>>> import sqlite3
>>> con = sqlite3.connect("catalog.db")
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.fetchone()
(1, u'unicross@mail.ru', u'password1')
>>> print cur.fetchone()
None
```

- `next()` — при каждом вызове возвращает одну запись из результата запроса в виде кортежа, а затем перемещает указатель текущей позиции. Если записей больше нет, метод возбуждает исключение `StopIteration`. Выведем все записи из таблицы `user` с помощью метода `next()`:

```
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.next()
(1, u'unicross@mail.ru', u'password1')
>>> cur.next()
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    cur.next()
StopIteration
```

Цикл `for` на каждой итерации вызывает метод `next()` автоматически. Поэтому для перебора записей достаточно указать объект-курсор в качестве параметра цикла. Выведем все записи из таблицы `rubr`:

```
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x0150E2F0>
>>> for id_rubr, name in cur: print "%s|%s" % (id_rubr, name)

1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

- `fetchmany([size=cursor.arraysize])` — при каждом вызове возвращает список записей из результата запроса, а затем перемещает указатель текущей позиции. Каждый элемент списка является кортежем. Количество элементов, выбираемых за один раз, задается с помощью необязательного параметра или значения атрибута `arraysize` объекта-курсора. Если количество записей в результате запроса меньше указанного количества элементов списка, то количество элементов списка будет соответствовать оставшемуся количеству записей. Если записей больше нет, метод возвращает пустой список. Пример:

```
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.arraysize
1
>>> cur.fetchmany()
[(1, u'\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c
\u0438\u0440\u043e\u0432\u0430\u043d\u0438\u0435')]
>>> cur.fetchmany(2)
[(2, u'\u041c\u0443\u0437\u044b\u043a\u0430'), (3,
u'\u041f\u043e\u0438\u0441\u043a\u043e\u0432\u044b\u0435
\u043f\u043e\u0440\u0442\u0430\u043b\u044b')]
>>> cur.fetchmany(3)
```

```
[(4, u'\u041a\u0438\u043d\u043e')]
>>> cur.fetchmany()
[]
```

- `fetchall()` — возвращает список всех (или всех оставшихся) записей из результата запроса. Каждый элемент списка является кортежем. Если записей больше нет, метод возвращает пустой список. Пример:

```
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.fetchall()
[(1, u'\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c
\u0438\u0440\u043e\u0432\u0430\u043d\u0438\u0435'), (2,
u'\u041c\u0443\u0437\u044b\u043a\u0430'), (3,
u'\u041f\u043e\u0438\u0441\u043a\u043e\u0432\u044b\u0435
\u043f\u043e\u0440\u0442\u0430\u043b\u044b'), (4,
u'\u041a\u0438\u043d\u043e')]
>>> cur.fetchall()
[]
```

Все рассмотренные методы возвращают запись в виде кортежа. Если необходимо изменить такое поведение и, например, получить записи в виде словаря, то следует воспользоваться атрибутом `row_factory` объекта соединения. В качестве значения атрибут принимает ссылку на функцию обратного вызова, имеющую следующий формат:

```
def <Название функции>(<Объект-курсор>, <Запись>)
    # Обработка записи
    return <Новый объект>
```

В качестве примера выведем записи из таблицы `user` в виде словаря (листинг 17.2).

Листинг 17.2. Атрибут `row_factory`

```
# -*- coding: utf-8 -*-
import sqlite3
def my_factory(c, r):
    d = {}
    for i, name in enumerate(c.description):
        d[name[0]] = r[i] # Ключи в виде названий полей
        d[i] = r[i]       # Ключи в виде индексов полей
    return d
con = sqlite3.connect("catalog.db")
con.row_factory = my_factory
cur = con.cursor()        # Создаем объект-курсор
cur.execute("SELECT * FROM user")
arr = cur.fetchall()
print arr                 # Результат:
"""[{0: 1, 1: u'unicross@mail.ru', 2: u'password1', 'id_user': 1,
```

```
'passw': u'password1', 'email': u'unicross@mail.ru'}]"""
print arr[0][1]           # Доступ по индексу
print arr[0]["email"]     # Доступ но названию поля
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
raw_input()
```

Функция `my_factory()` будет вызываться для каждой записи. Обратите внимание на то, что название функции в операции присваивания атрибуту `row_factory` указывается без круглых скобок. Если скобки указать, то смысл операции будет совсем иным.

Атрибуту `row_factory` можно присвоить ссылку на объект `Row` из модуля `sqlite3`. Этот объект позволяет получить доступ к значению поля как по индексу, так и по названию поля. Причем название не зависит от регистра символов. Начиная с Python 2.6, объект `Row` поддерживает итерации и метод `keys()`, который возвращает список с названиями полей. Переделаем наш предыдущий пример и используем объект `Row` (листинг 17.3).

Листинг 17.3. Объект `Row`

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
con.row_factory = sqlite3.Row
cur = con.cursor()
cur.execute("SELECT * FROM user")
arr = cur.fetchall()
print type(arr[0])        # <type 'sqlite3.Row'>
print len(arr[0])         # 3
print arr[0][1]           # Доступ по индексу
print arr[0]["email"]     # Доступ по названию поля
print arr[0]["EMAIL"]     # Не зависит от регистра символов
for elem in arr[0]:
    print elem
print arr[0].keys()       # ['id_user', 'email', 'passw']
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
raw_input()
```

Как видно из результатов предыдущих примеров, все данные, имеющие в SQLite тип `TEXT`, возвращаются в виде Unicode-строк. В предыдущей главе мы создали базу данных testdb.db и сохраняли данные в полях таблицы в кодировке Windows-1251. Попробуем отобразить записи из таблицы с рубриками:

```
>>> con = sqlite3.connect("testdb.db")
>>> cur = con.cursor()
```

```
>>> cur.execute("SELECT * FROM rubr")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    cur.execute("SELECT * FROM rubr")
OperationalError: Could not decode to UTF-8 column 'name_rubr' with
text 'Программирование'
```

При осуществлении преобразования обычной строки в Unicode-строку предполагается, что строка хранится в кодировке UTF-8. Так как в нашем примере мы используем другую кодировку, то при преобразовании возникает ошибка и возбуждается исключение `OperationalError`. Обойти это исключение позволяет атрибут `text_factory` объекта соединения. В качестве значения атрибута указывается ссылка на функцию, которая будет использоваться для осуществления преобразования значения текстовых полей. Например, чтобы вернуть обычную строку следует указать ссылку на функцию `str()` (листинг 17.4).

Листинг 17.4. Атрибут `text_factory`

```
>>> con = sqlite3.connect("testdb.db")
>>> con.text_factory = str # Название функции без круглых скобок!
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x014FE380>
>>> cur.fetchone()
(1, '\xcf\xf0\xee\xe3\xf0\xe0\xec\xec\xe8\xf0\xee\xe2\xe0\xed\xe8\xe5')
```

Если необходимо вернуть Unicode-строку, то внутри функции обратного вызова следует вызвать функцию `unicode()` и явно указать кодировку данных. Функция обратного вызова должна принимать один параметр и возвращать преобразованную строку. Выведем текстовые данные в виде Unicode-строк (листинг 17.5).

Листинг 17.5. Указание пользовательской функции преобразования

```
>>> con.text_factory = lambda s: unicode(s, "cp1251")
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x014FE380>
>>> cur.fetchone()
(1, u'\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c\u0438
\u0440\u043e\u0432\u0430\u043d\u0438\u0435')
```

Атрибуту `text_factory` можно также присвоить значение `sqlite3.OptimizedUnicode` (листинг 17.6). В этом случае, если строка состоит только из ASCII-символов, то возвращается обычная строка, в противном случае возвращается Unicode-строка.

Листинг 17.6. Указание объекта `sqlite3.OptimizedUnicode`

```
>>> con = sqlite3.connect("catalog.db")
>>> con.text_factory = sqlite3.OptimizedUnicode
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x01508AD0>
>>> type(cur.fetchone()[1])
<type 'unicode'>
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x01508AD0>
>>> type(cur.fetchone()[1])
<type 'str'>
>>> con.close()
```

17.4. Управление транзакциями

Перед выполнением первого запроса автоматически запускается транзакция. Поэтому все запросы, изменяющие записи (`INSERT`, `REPLACE`, `UPDATE` и `DELETE`), необходимо завершать вызовом метода `commit()` объекта соединения. Если метод не вызвать и при этом закрыть соединение с базой данных, то все произведенные изменения будут отменены. Транзакция может автоматически завершаться при выполнении запросов `CREATE TABLE`, `VACUUM` и некоторых других. После выполнения этих запросов транзакция запускается снова.

Если необходимо отменить изменения, то следует вызвать метод `rollback()` объекта соединения. В качестве примера добавим нового пользователя, а затем отменим транзакцию и выведем содержимое таблицы (листинг 17.7).

Листинг 17.7. Отмена изменений с помощью метода `rollback()`

```
>>> con = sqlite3.connect("catalog.db")
>>> cur = con.cursor()
>>> cur.execute("INSERT INTO user VALUES (Null, 'user@mail.ru', '')")
<sqlite3.Cursor object at 0x01508CB0>
>>> con.rollback()                          # Отмена изменений
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x01508CB0>
>>> cur.fetchall()
[(1, u'unicross@mail.ru', u'password1')]
>>> con.close()
```

Управлять транзакцией можно с помощью параметра `isolation_level` в функции `connect()`, а также с помощью атрибута `isolation_level` объекта соединения. Допустимые значения: `"DEFERRED"`, `"IMMEDIATE"`, `"EXCLUSIVE"`, пустая строка и

None. Первые три значения передаются в инструкцию BEGIN. Если в качестве значения указать None, то транзакция запускаться не будет. В этом случае нет необходимости вызывать метод commit(). Все изменения будут сразу сохраняться в базе данных. Отключим автоматический запуск транзакции с помощью параметра isolation_level, добавим нового пользователя, а затем подключимся заново и выведем все записи из таблицы (листинг 17.8).

Листинг 17.8. Управление транзакциями

```
>>> con = sqlite3.connect("catalog.db", isolation_level=None)
>>> cur = con.cursor()
>>> cur.execute("INSERT INTO user VALUES (Null, 'user@mail.ru', '')")
<sqlite3.Cursor object at 0x01508CE0>
>>> con.close()
>>> con = sqlite3.connect("catalog.db")
>>> con.isolation_level = None # Отключение запуска транзакции
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM user")
<sqlite3.Cursor object at 0x01508530>
>>> cur.fetchall()
[(1, u'unicross@mail.ru', u'password1'), (2, u'user@mail.ru', u'')]
>>> con.close()
```

17.5. Создание пользовательской сортировки

По умолчанию сортировка с помощью инструкции ORDER BY зависит от регистра символов. Например, если сортировать слова "единица1", "Единица2" и "Единый", то в результате мы получим неправильную сортировку ("Единица2", "Единый" и лишь затем "единица1"). Модуль sqlite3 позволяет создать пользовательскую функцию сортировки и связать ее с названием функции в SQL-запросе. В дальнейшем это название можно указать в инструкции ORDER BY после ключевого слова COLLATE.

Связать название функции в SQL-запросе с пользовательской функцией в программе позволяет метод create_collation() объекта соединения. Формат метода:

```
create_collation(<Название функции в SQL-запросе в виде строки>,
                 <Ссылка на функцию сортировки>)
```

Функция сортировки принимает две строки (обычно в кодировке UTF-8) и должна возвращать:

- 1 — если первая строка больше второй;
- -1 — если вторая больше первой;
- 0 — если строки равны.

Обратите внимание на то, что функция сортировки будет вызываться только при сравнении текстовых значений. При сравнении чисел функция вызвана не будет.

В качестве примера создадим новую таблицу с одним полем, вставим три записи, а затем произведем сортировку стандартным методом и с помощью пользовательской функции (листинг 17.9).

Листинг 17.9. Сортировка записей

```
# -*- coding: utf-8 -*-
import sqlite3
def myfunc(s1, s2): # Пользовательская функция сортировки
    s1 = s1.decode("utf-8").lower()
    s2 = s2.decode("utf-8").lower()
    return cmp(s1, s2)
con = sqlite3.connect("catalog.db", isolation_level=None)
# Связываем имя "myfunc" с функцией myfunc()
con.create_collation("myfunc", myfunc)
cur = con.cursor()
cur.execute("CREATE TABLE words (word TEXT)")
cur.execute("INSERT INTO words VALUES('единица1')")
cur.execute("INSERT INTO words VALUES('Единый')")
cur.execute("INSERT INTO words VALUES('Единица2')")
# Стандартная сортировка
cur.execute("SELECT * FROM words ORDER BY word")
for line in cur:
    print line[0], # Результат: Единица2 Единый единица1
print
# Пользовательская сортировка
cur.execute("""SELECT * FROM words
               ORDER BY word COLLATE myfunc""")
for line in cur:
    print line[0], # Результат: единица1 Единица2 Единый
cur.close()
con.close()
raw_input()
```

17.6. Поиск без учета регистра символов

Как уже говорилось в предыдущей главе, сравнение строк и поиск с помощью оператора `LIKE` для русских букв производятся с учетом регистра символов. Поэтому следующие выражения вернут значение `0`:

```
cur.execute("SELECT 'строка' = 'Строка'")
print cur.fetchone()[0]                 # Результат: 0 (не равно)
cur.execute("SELECT 'строка' LIKE 'Строка'")
print cur.fetchone()[0]                 # Результат: 0 (не найдено)
```

Одним из вариантов решения проблемы является преобразование символов обоих строк к верхнему или нижнему регистру. Но встроенные функции SQLite `UPPER()` и `LOWER()` с русскими буквами опять работают некорректно. Модуль `sqlite3` позволяет создать пользовательскую функцию и связать ее с названием функции в SQL-запросе. Таким образом, можно создать пользовательскую функцию преобразования регистра символов, а затем указать связанное с ней имя в SQL-запросе.

Связать название функции в SQL-запросе с пользовательской функцией в программе позволяет метод `create_function()` объекта соединения. Формат метода:

```
create_function(<Название функции в SQL-запросе в виде строки>,
                <Количество параметров>, <Ссылка на функцию>)
```

В первом параметре указывается название функции в виде строки. Количество параметров, принимаемых функцией, задается во втором параметре. Параметры могут быть любого типа. Если функция принимает строку, то ее типом данных будет `unicode`. В третьем параметре указывается ссылка на пользовательскую функцию в программе. Для примера произведем поиск рубрики без учета регистра символов (листинг 17.10).

Листинг 17.10. Поиск без учета регистра символов

```
# -*- coding: utf-8 -*-
import sqlite3
# Пользовательская функция изменения регистра
def myfunc(s):
    return s.lower()
con = sqlite3.connect("catalog.db")
# Связываем имя "mylower" с функцией myfunc()
con.create_function("mylower", 1, myfunc)
cur = con.cursor()
string = u"%МуЗЫка%"                          # Строка для поиска
# Поиск без учета регистра символов
sql = """SELECT * FROM rubr
         WHERE mylower(name_rubr) LIKE ?"""
cur.execute(sql, (string.lower(),))
print cur.fetchone()[1]                       # Результат: Музыка
cur.close()
con.close()
raw_input()
```

В этом примере предполагается, что значение переменной `string` получено от пользователя. Обратите внимание на то, что строку для поиска в метод `execute()` мы передаем в нижнем регистре. Если этого не сделать и указать преобразование в SQL-запросе, то будет производиться лишнее преобразование регистра при каждом сравнении.

Метод `create_function()` используется не только для создания функции изменения регистра символов, но и для других целей. Например, в SQLite нет специального типа данных для хранения даты и времени. Дату и время можно хранить разными способами, например, как количество секунд, прошедших с начала эпохи, в числовом поле. Для преобразования количества секунд в другой формат следует создать пользовательскую функцию форматирования (листинг 17.11).

Листинг 17.11. Преобразование даты и времени

```
# -*- coding: utf-8 -*-
import sqlite3
import time
def myfunc(d):
    return time.strftime("%d.%m.%Y", time.localtime(d))
con = sqlite3.connect(":memory:")
# Связываем имя "mytime" с функцией myfunc()
con.create_function("mytime", 1, myfunc)
cur = con.cursor()
cur.execute("SELECT mytime(1275762391)")
print cur.fetchone()[0] # Результат: 05.06.2010
cur.close()
con.close()
raw_input()
```

17.7. Создание агрегатных функций

При изучении SQLite мы рассматривали встроенные агрегатные функции `COUNT()`, `MIN()`, `MAX()`, `AVG()` и `SUM()`. Если возможностей этих функций окажется недостаточно, то можно определить пользовательскую агрегатную функцию. Связать название функции в SQL-запросе с пользовательским классом в программе позволяет метод `create_aggregate()` объекта соединения. Формат метода:

```
create_aggregate(<Название функции в SQL-запросе в виде строки>,
                 <Количество параметров>, <Ссылка на класс>)
```

В первом параметре указывается название агрегатной функции в виде строки. В третьем параметре передается ссылка на класс (название класса без круглых скобок). Этот класс должен иметь два метода `step()` и `finalize()`. Метод `step()` вызывается несколько раз и ему передаются параметры. Количество параметров задается во втором параметре метода `create_aggregate()`. Если метод принимает строку, то ее типом данных будет `unicode`. Метод `finalize()` должен возвращать результат выполнения. В качестве примера выведем все названия рубрик в алфавитном порядке через разделитель (листинг 17.12).

Листинг 17.12. Создание агрегатной функции

```
# -*- coding: utf-8 -*-
import sqlite3
class MyClass:
    def __init__(self):
        self.result = []
    def step(self, value):
        self.result.append(value)
    def finalize(self):
        self.result.sort()
        return " - ".join(self.result)
con = sqlite3.connect("catalog.db")
# Связываем имя "myfunc" с классом MyClass
con.create_aggregate("myfunc", 1, MyClass)
cur = con.cursor()
cur.execute("SELECT myfunc(name_rubr) FROM rubr")
print cur.fetchone()[0]
# Результат: Кино - Музыка - Поисковые порталы - Программирование
cur.close()
con.close()
raw_input()
```

17.8. Преобразование типов данных

SQLite поддерживает пять типов данных. Для каждого типа SQLite в модуле `sqlite3` определено соответствие с типом данных в языке Python:

- `NULL` — значение `NULL`. Значение соответствует типу `None` в Python;
- `INTEGER` — целые числа. Соответствует типу `int`. Если число превышает максимально допустимое для `int` значение, то преобразуется в тип `long`;
- `REAL` — вещественные числа. Соответствует типу `float`;
- `TEXT` — строки. По умолчанию преобразуется в тип `unicode`. Предполагается, что строка в базе данных хранится в кодировке UTF-8. Соответствие можно изменить с помощью атрибута `text_factory`;
- `BLOB` — бинарные данные. Соответствует типу `buffer`.

Если необходимо сохранить в таблице данные, которые имеют тип, не поддерживаемый SQLite, то следует преобразовать тип самостоятельно. Для этого с помощью функции `register_adapter()` можно зарегистрировать пользовательскую функцию, которая будет вызываться при попытке вставки объекта в SQL-запрос. Функция имеет следующий формат:

```
register_adapter(<Тип данных или класс>, <Ссылка на функцию>)
```

В первом параметре указывается тип данных или ссылка на класс нового стиля. Во втором параметре задается ссылка на функцию, которая будет вызываться для

преобразования типа. Функция принимает один параметр и должна возвращать значение, имеющее тип данных, поддерживаемый SQLite. В качестве примера создадим новую таблицу и сохраним в ней значения атрибутов класса нового стиля (листинг 17.13).

Листинг 17.13. Сохранение в базе атрибутов класса

```
# -*- coding: utf-8 -*-
import sqlite3
class Car(object):
    def __init__(self, model, color):
        self.model, self.color = model, color
def my_adapter(car):
    return u"%s|%s" % (car.model, car.color)
# Регистрируем функцию для преобразования типа
sqlite3.register_adapter(Car, my_adapter)
# Создаем экземпляр класса Car
car = Car(u"ВАЗ-2109", u"красный")
con = sqlite3.connect("catalog.db")
cur = con.cursor()
try:
    cur.execute("CREATE TABLE cars1 (model TEXT)")
    cur.execute("INSERT INTO cars1 VALUES (?)", (car,))
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
    con.commit()
cur.close()
con.close()
raw_input()
```

Вместо регистрации функции преобразования типа можно внутри класса нового стиля определить метод `__conform__()`. Формат метода:

```
__conform__(self, <Протокол>)
```

Параметр `<Протокол>` будет соответствовать `PrepareProtocol`. Более подробно о протоколе можно прочитать в документе PEP 246. Метод должен возвращать значение, имеющее тип данных, поддерживаемый SQLite. Создадим таблицу `cars2` и сохраним в ней значения атрибутов, используя метод `__conform__()` (листинг 17.14).

Листинг 17.14. Использование метода `__conform__()`

```
# -*- coding: utf-8 -*-
import sqlite3
```

```
class Car(object):
    def __init__(self, model, color):
        self.model, self.color = model, color
    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return u"%s|%s" % (self.model, self.color)
# Создаем экземпляр класса Car
car = Car(u"Москвич-412", u"синий")
con = sqlite3.connect("catalog.db")
cur = con.cursor()
try:
    cur.execute("CREATE TABLE cars2 (model mycar)")
    cur.execute("INSERT INTO cars2 VALUES (?)", (car,))
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
    con.commit()
cur.close()
con.close()
raw_input()
```

Чтобы восстановить объект, следует зарегистрировать функцию преобразования типа данных SQLite в тип данных Python с помощью функции `register_converter()`. Функция имеет следующий формат:

```
register_converter(<Тип данных>, <Ссылка на функцию>)
```

В первом параметре указывается преобразуемый тип данных в виде строки, а во втором параметре задается ссылка на функцию, которая будет использоваться для преобразования типа данных. Функция должна принимать один параметр и возвращать преобразованное значение.

Чтобы интерпретатор смог определить, какую функцию необходимо вызвать для преобразования типа данных, следует явно указать местоположение метки с помощью параметра `detect_types` функции `connect()`. Параметр может принимать следующие значения (или их комбинацию через символ `|`):

- `sqlite3.PARSE_COLNAMES` — тип данных указывается в SQL-запросе в псевдониме поля внутри квадратных скобок. Пример указания типа `mycar` для поля `model`:

  ```
  SELECT model as "c [mycar]" FROM cars1
  ```

- `sqlite3.PARSE_DECLTYPES` — тип данных определяется по значению, указанному после названия поля в инструкции `CREATE TABLE`. Пример указания типа `mycar` для поля `model`:

  ```
  CREATE TABLE cars2 (model mycar)
  ```

Выведем сохраненное значение из таблицы `cars1` (листинг 17.15).

Листинг 17.15. Использование значения `sqlite3.PARSE_COLNAMES`

```
# -*- coding: utf-8 -*-
import sqlite3, sys
class Car(object):
    def __init__(self, model, color):
        self.model, self.color = model, color
    def __repr__(self):
        s = u"Модель: %s, цвет: %s" % (self.model, self.color)
        enc = sys.stdout.encoding # Кодировка терминала
        return s.encode(enc)
def my_converter(value):
    value = unicode(value, "utf-8")
    model, color = value.split("|")
    return Car(model, color)
# Регистрируем функцию для преобразования типа
sqlite3.register_converter("mycar", my_converter)
con = sqlite3.connect("catalog.db",
                      detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("""SELECT model as "c [mycar]" FROM cars1""")
print cur.fetchone()[0]
# Результат: Модель: ВАЗ-2109, цвет: красный
con.close()
raw_input()
```

Теперь выведем значение из таблицы `cars2` (листинг 17.16).

Листинг 17.16. Использование значения `sqlite3.PARSE_DECLTYPES`

```
# -*- coding: utf-8 -*-
import sqlite3, sys
class Car(object):
    def __init__(self, model, color):
        self.model, self.color = model, color
    def __repr__(self):
        s = u"Модель: %s, цвет: %s" % (self.model, self.color)
        enc = sys.stdout.encoding # Кодировка терминала
        return s.encode(enc)
def my_converter(value):
    value = unicode(value, "utf-8")
    model, color = value.split("|")
    return Car(model, color)
# Регистрируем функцию для преобразования типа
sqlite3.register_converter("mycar", my_converter)
```

```
con = sqlite3.connect("catalog.db",
                      detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("SELECT model FROM cars2")
print cur.fetchone()[0]
# Результат: Модель: Москвич-412, цвет: синий
con.close()
raw_input()
```

17.9. Сохранение в таблице даты и времени

В SQLite нет специальных типов данных для представления даты и времени. Поэтому обычно дату преобразовывают в строку или число (количество секунд, прошедших с начала эпохи) и сохраняют в соответствующих полях. При выводе данные необходимо опять преобразовывать. Используя знания, полученные в предыдущем разделе, можно зарегистрировать две функции преобразования (листинг 17.17).

Листинг 17.17. Сохранение в таблице даты и времени

```
# -*- coding: utf-8 -*-
import sqlite3, datetime, time
# Преобразование даты в число
def my_adapter(t):
    return time.mktime(t.timetuple())
# Преобразование в дату
def my_converter(t):
    return datetime.datetime.fromtimestamp(float(t))
# Регистрируем обработчики
sqlite3.register_adapter(datetime.datetime, my_adapter)
sqlite3.register_converter("mytime", my_converter)
# Получаем текущую дату и время
dt = datetime.datetime.today()
con = sqlite3.connect(":memory:", isolation_level=None,
                      detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("CREATE TABLE times (time)")
cur.execute("INSERT INTO times VALUES (?)", (dt,))
cur.execute("""SELECT time as "t [mytime]" FROM times""")
print cur.fetchone()[0] # 2010-06-08 10:35:18
con.close()
raw_input()
```

Модуль sqlite3 для типов date и datetime из модуля datetime содержит встроенные функции для преобразования типов. Для datetime.date зарегистрирован тип date, а для datetime.datetime — тип timestamp. Таким образом, создавать пользовательские функции преобразования не нужно. Пример сохранения в таблице даты и времени приведен в листинге 17.18.

Листинг 17.18. Встроенные функции для преобразования типов

```
# -*- coding: utf-8 -*-
import sqlite3, datetime
# Получаем текущую дату и время
d = datetime.date.today()
dt = datetime.datetime.today()
con = sqlite3.connect(":memory:", isolation_level=None,
                      detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("CREATE TABLE times (d date, dt timestamp)")
cur.execute("INSERT INTO times VALUES (?, ?)", (d, dt))
cur.execute("SELECT d, dt FROM times")
res = cur.fetchone()
print res[0] # 2010-06-08
print res[1] # 2010-06-08 10:37:04.453000
con.close()
raw_input()
```

17.10. Обработка исключений

Модуль sqlite3 поддерживает следующую иерархию исключений:

```
StandardError
    Warning
    Error
        InterfaceError
        DatabaseError
            DataError
            OperationalError
            IntegrityError
            InternalError
            ProgrammingError
            NotSupportedError
```

Базовым классом самого верхнего уровня является класс StandardError. Все остальные исключения определены в модуле sqlite3. Поэтому при указании исключения в инструкции except следует предварительно указать название модуля (например, sqlite3.DatabaseError).

Исключения возбуждаются в следующих случаях:

- `Warning` — при наличии важных предупреждений;
- `Error` — базовый класс для всех остальных исключений, возбуждаемых в случае ошибки. Если указать этот класс в инструкции `except`, то будут перехватываться все ошибки;
- `InterfaceError` — при ошибках, которые связаны с интерфейсом базы данных, а не с самой базой данных;
- `DatabaseError` — базовый класс для исключений, которые связаны с базой данных;
- `DataError` — при ошибках, возникающих при обработке данных;
- `OperationalError` — вызывается при ошибках, которые связаны с операциями в базе данных, например, при синтаксической ошибке в SQL-запросе, несоответствии количества полей в инструкции `INSERT`, отсутствии поля с указанным именем и т. д. Иногда не зависит от правильности SQL-запроса;
- `IntegrityError` — при наличии проблем с внешними ключами или индексами;
- `InternalError` — при внутренней ошибке в базе данных;
- `ProgrammingError` — возникает при ошибках программирования. Например, количество переменных, указанных во втором параметре метода `execute()`, не совпадает с количеством специальных символов в SQL-запросе;
- `NotSupportedError` — при использовании методов, не поддерживаемых базой данных.

В качестве примера обработки исключений напишем программу, которая позволяет пользователям вводить название базы данных и SQL-команды в консоли (листинг 17.19).

Листинг 17.19. Выполнение SQL-команд, введенных в консоли

```
# -*- coding: utf-8 -*-
import sqlite3, sys, re
def db_connect(db_name):
    try:
        db = sqlite3.connect(db_name, isolation_level=None)
    except (sqlite3.Error, sqlite3.Warning), err:
        print u"Не удалось подключиться к БД"
        raw_input()
        sys.exit(0)
    return db
enc = sys.stdin.encoding        # Кодировка терминала
print u"Введите название базы данных:",
db_name = raw_input()
con = db_connect(db_name)       # Подключаемся к базе
cur = con.cursor()
sql = ""
print u"Чтобы закончить выполнение программы введите <Q>+<Enter>"
while True:
    tmp = raw_input()
```

```
    if tmp in ["q", "Q"]:
        break
    if tmp.strip() == "":
        continue
    tmp = tmp.decode(enc).encode("utf-8")
    sql = "%s %s" % (sql, tmp)
    if sqlite3.complete_statement(sql):
        try:
            sql = sql.strip()
            cur.execute(sql)
            if re.match("SELECT ", sql, re.I):
                print cur.fetchall()
        except (sqlite3.Error, sqlite3.Warning), err:
            print u"Ошибка:", err
        else:
            print u"Запрос успешно выполнен"
        sql = ""
cur.close()
con.close()
```

В консоли Windows по умолчанию используется кодировка cp866. С помощью команды `chcp <Кодовая таблица>` можно изменить кодировку. Чтобы сделать программу более универсальной, мы получаем кодировку консоли с помощью атрибута `encoding` объекта `sys.stdin`, а затем указываем ее при преобразовании SQL-запроса в кодировку UTF-8, которая используется в базе данных SQLite. Таким образом, русские буквы будут обрабатываться правильно независимо от кодировки консоли.

Чтобы SQL-запрос можно было разместить на нескольких строках, мы производим проверку завершенности запроса с помощью функции `complete_statement(<SQL-запрос>)`. Функция возвращает `True`, если параметр содержит один или более полных SQL-запросов. Признаком завершенности запроса является точка с запятой. Никакой проверки правильности SQL-запроса не производится. Пример использования функции:

```
>>> sql = "SELECT 10 > 5;"
>>> sqlite3.complete_statement(sql)
True
>>> sql = "SELECT 10 > 5"
>>> sqlite3.complete_statement(sql)
False
>>> sql = "SELECT 10 > 5; SELECT 20 + 2;"
>>> sqlite3.complete_statement(sql)
True
```

Начиная с версии 2.6, язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует выполнение завершающих действий вне зависимо-

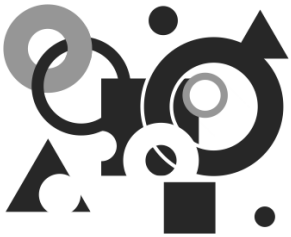
сти от того, произошло исключение внутри блока кода или нет. В модуле `sqlite3` объект соединения поддерживает этот протокол. Если внутри блока `with` не произошло исключение, то автоматически вызывается метод `commit()`. В противном случае все изменения отменяются с помощью метода `rollback()`. Для примера добавим три рубрики в таблицу `rubr`. В первом случае запрос будет без ошибок, а во втором случае выполним два запроса, последний из которых будет добавлять рубрику с уже существующим идентификатором (листинг 17.20).

Листинг 17.20. Инструкция `with...as`

```
# -*- coding: utf-8 -*-
# Работает только в версиях >= 2.6
import sqlite3
con = sqlite3.connect(r"C:\book\catalog.db")
try:
    with con:
        # Добавление новой рубрики
        con.execute("""INSERT INTO rubr VALUES (NULL, 'Мода')""")
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
try:
    with con:
        # Добавление новой рубрики
        con.execute("""INSERT INTO rubr VALUES (NULL, 'Спорт')""")
        # Рубрика с идентификатором 1 уже существует !!!
        con.execute("""INSERT INTO rubr VALUES (1, 'Казино')""")
except sqlite3.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
con.close()
raw_input()
```

Итак, в первом случае запрос не содержит ошибок и рубрика "Мода" будет успешно добавлена в таблицу. Во втором случае возникнет исключение `IntegrityError`. Поэтому ни рубрика "Спорт", ни рубрика "Казино" в таблицу добавлены не будут, т. к. все изменения автоматически отменяются с помощью вызова метода `rollback()`.

Глава 18



Доступ к базе данных MySQL

MySQL является наиболее популярной системой управления базами данных среди СУБД, не требующих вносить денежные отчисления за лицензию. Особенную популярность MySQL получила в Web-программировании. На сегодняшний день очень трудно найти платный хостинг, на котором нельзя было бы использовать MySQL. И это не удивительно. MySQL обладает простым синтаксисом, имеет высокую скорость работы и предоставляет функциональность, доступную ранее только в коммерческих базах данных.

В отличие от SQLite, работающей с файлом базы непосредственно, MySQL поддерживает архитектуру "клиент/сервер". Это означает, что MySQL запускается на определенном порту (обычно 3306) и ожидает запросы. Клиент подключается к серверу, посылает запрос, а в ответ получает результат. Сервер MySQL может быть запущен как на локальном компьютере, так и на отдельном компьютере в сети, специально предназначенном для обслуживания запросов к базам данных. MySQL обеспечивает доступ к данным одновременно сразу нескольким пользователям. При этом доступ к данным предоставляется только пользователям, имеющим на это право.

MySQL не поставляется вместе с Python. Кроме того, в состав стандартной библиотеки не входят модули, предназначенные для работы с MySQL. Все эти компоненты необходимо устанавливать отдельно. Загрузить дистрибутив MySQL можно со страницы **http://dev.mysql.com/downloads/mysql/**. Описание процесса установки и рассмотрение функциональных возможностей MySQL выходит за рамки этой книги. В дальнейшем предполагается, что сервер MySQL уже установлен на компьютере и вы умеете работать с этой базой данных. Если это не так, то вначале следует изучить специальную литературу по MySQL и лишь затем вернуться к изучению материала, описываемого в этой главе. Описание MySQL можно также найти в моих книгах "HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера"[1] и "Разработка Web-сайтов с помощью Perl и MySQL"[2].

---

[1] Прохоренок Н. HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера. (+ Видеокурс на CD). — 3-е изд. — СПб.: БХВ-Петербург, 2010.

[2] Прохоренок Н. Разработка Web-сайтов с помощью Perl и MySQL. — СПб.: БХВ-Петербург, 2009.

Для доступа к базе данных MySQL существует большое количество модулей сторонних разработчиков. Получить полный список модулей можно на странице **http://wiki.python.org/moin/MySQL**. В этой главе мы рассмотрим функциональные возможности модулей `MySQLdb` и `PyODBC`.

18.1. Модуль *MySQLdb*

Модуль `MySQLdb` используется наиболее часто. Для установки модуля со страницы **http://www.codegood.com/archives/4** скачиваем файл MySQL-python-1.2.3c1.win32-py2.6.exe, а затем запускаем его с помощью двойного щелчка на значке файла. Процесс установки предельно прост и в комментариях не нуждается.

Примечание

На официальной странице модуля (**http://sourceforge.net/projects/mysql-python/)** можно найти программы установки под Windows для Python 2.5 и более ранних версий. К сожалению, для Python 2.6 официальной программы установки под Windows нет. Вполне возможно, в будущем ситуация изменится.

Чтобы проверить работоспособность модуля, в окне **Python Shell** редактора IDLE набираем следующий код:

```
>>> import MySQLdb
>>> MySQLdb.__version__
'1.2.3c1'
```

Модуль `MySQLdb` является оберткой над модулем `_mysql` и предоставляет интерфейс доступа, совместимый со спецификацией DB-API. Получить номер поддерживаемой версии спецификации можно с помощью атрибута `apilevel`:

```
>>> MySQLdb.apilevel
'2.0'
```

18.1.1. Подключение к базе данных

Для подключения к базе данных используется функция `connect()`. Функция имеет следующий формат:

```
connect(<Параметры>)
```

Функция `connect()` возвращает объект соединения, с помощью которого осуществляется вся дальнейшая работа с базой данных. Если подключиться не удалось, возбуждается исключение. Соединение закрывается, когда вызывается метод `close()` объекта соединения. Рассмотрим наиболее важные параметры функции `connect()`:

- `host` — имя хоста. По умолчанию используется локальный хост;
- `user` — имя пользователя;
- `passwd` — пароль для авторизации пользователя. По умолчанию пустой пароль;

- `db` — название базы данных, которую необходимо выбрать для работы. По умолчанию никакая база данных не выбирается. Указать название базы данных можно также после подключения с помощью метода `select_db()` объекта соединения;
- `port` — номер порта, на котором запущен сервер MySQL. Значение по умолчанию 3306;
- `unix_socket` — местоположение сокета UNIX;
- `conv` — словарь преобразования типов.
  По умолчанию `MySQLdb.converters.conversions`;
- `compress` — включение протокола сжатия. По умолчанию нет сжатия;
- `connect_timeout` — ограничение времени соединения. По умолчанию ограничения нет;
- `named_pipe` — использовать именованный канал (применяется только в Windows). По умолчанию не используется;
- `init_command` — команда, передаваемая на сервер при подключении;
- `cursorclass` — класс курсора. По умолчанию `MySQLdb.cursors.Cursor`;
- `sql_mode` — режим SQL;
- `use_unicode` — если параметр имеет значение `True`, то значения, хранящиеся в полях `CHAR`, `VARCHAR` и `TEXT`, будут возвращаться в виде Unicode-строк;
- `read_default_file` — местоположение конфигурационного файла MySQL;
- `read_default_group` — название секции в конфигурационном файле, из которой будут считываться параметры. По умолчанию `[client]`;
- `charset` — название кодовой таблицы, которая будет использоваться при преобразовании значений в Unicode-строку.

Последние три параметра необходимо рассмотреть более подробно. Если кодировка не указана, то в большинстве случаев сервер MySQL настроен на кодировку соединения `latin1`. Получить настройки кодировки позволяет метод `get_character_set_info()` (листинг 18.1).

Листинг 18.1. Получение настроек кодировки

```
>>> import MySQLdb # Подключаем модуль MySQLdb
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456")
>>> con.get_character_set_info()
{'collation': 'latin1_swedish_ci', 'comment': '', 'mbminlen': 1,
'name': 'latin1', 'mbmaxlen': 1}
>>> con.close()
```

В MySQL-python-1.2.2 под Python 2.5 (в MySQL-python-1.2.3c1 под Python 2.6 подобной ошибки не возникает) по умолчанию поиск файлов с кодовыми таблицами производился в папке C:\mysql\share\charsets\. Поэтому попытка задать коди-

ровку в параметре `charset` без указания значения в параметре `read_default_file` приводила к ошибке:

```
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456", charset="cp1251")
Traceback (most recent call last):
... Фрагмент опущен ...
OperationalError: (2019, "Can't initialize character set cp1251
(path: C:\\mysql\\\\share\\charsets\\)")
```

Исключением является кодировка UTF-8, которая используется в MySQL по умолчанию и не требует наличия файла с кодовой таблицей:

```
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456", charset="utf8")
>>> con.get_character_set_info()
{'collation': 'utf8_general_ci', 'comment': '', 'mbminlen': 1,
'name': 'utf8', 'mbmaxlen': 3}
>>> con.close()
```

Чтобы избежать ошибки, необходимо в параметре `read_default_file` указать путь к конфигурационному файлу MySQL. Причем в этом файле должен быть задан путь к файлам с кодовыми таблицами в директиве `character-sets-dir` внутри секции `[client]`:

```
character-sets-dir="C:\\Program Files\\MySQL\\MySQL Server
5.1\\share\\charsets\\"
```

Если ранее производились попытки подключения в окне **Python Shell** редактора IDLE, то прежде чем выполнить дальнейшие примеры, следует закрыть, а затем снова открыть IDLE. В противном случае даже при указании пути к файлам с кодовыми таблицами все равно произойдет ошибка. Пример указания пути к конфигурационному файлу приведен в листинге 18.2.

Листинг 18.2. Указание пути к конфигурационному файлу

```
>>> import MySQLdb # Подключаем модуль MySQLdb
>>> ini = r"C:\Program Files\MySQL\MySQL Server 5.1\my.ini"
>>> con = MySQLdb.connect(host="localhost", user="root",
          passwd="123456", read_default_file=ini, charset="cp1251")
>>> con.get_character_set_info()
{'comment': 'Windows Cyrillic', 'name': 'cp1251', 'mbmaxlen': 1,
'mbminlen': 1, 'collation': 'cp1251_general_ci', 'dir':
'C:\\Program Files\\MySQL\\MySQL Server 5.1\\share\\charsets\\'}
>>> con.close()
```

В конфигурационном файле my.ini можно сразу указать кодировку соединения с помощью директивы `default-character-set`. В этом случае задавать кодировку с помощью параметра `charset` нет необходимости.

Указать кодировку позволяет также метод `set_character_set(<Кодировка>)` объекта соединения (листинг 18.3).

Листинг 18.3. Указание кодировки соединения

```
>>> ini = r"C:\Program Files\MySQL\MySQL Server 5.1\my.ini"
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456", read_default_file=ini)
>>> con.set_character_set("cp1251")
>>> con.get_character_set_info()
{'comment': 'Windows Cyrillic', 'name': 'cp1251', 'mbmaxlen': 1,
'mbminlen': 1, 'collation': 'cp1251_general_ci', 'dir':
'C:\\Program Files\\MySQL\\MySQL Server 5.1\\share\\charsets\\'}
>>> con.close()
```

18.1.2. Выполнение запроса

Согласно спецификации DB-API 2.0 после создания объекта соединения необходимо создать объект-курсор. Все дальнейшие запросы должны производиться через этот объект. Создание объекта-курсора осуществляется с помощью метода `cursor([<Класс курсора>])`. Для выполнения запроса к базе данных предназначены следующие методы курсора `MySQLdb.cursors.Cursor`:

- `close()` — закрывает объект-курсор;
- `execute(<SQL-запрос>[, <Значения>])` — выполняет один SQL-запрос. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Создадим новую базу данных:

```
# -*- coding: utf-8 -*-
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456")
cur = con.cursor()        # Создаем объект-курсор
sql = """CREATE DATABASE `python`
DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci"""
try:                      # Обрабатываем исключения
    cur.execute(sql)      # Выполняем SQL-запрос
except MySQLdb.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()               # Закрываем объект-курсор
con.close()               # Закрываем соединение
raw_input()
```

Теперь подключимся к новой базе данных, создадим таблицу и добавим запись:

```
# -*- coding: utf-8 -*-
import MySQLdb
```

```
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", db="python")
cur = con.cursor()
sql_1 = """\
CREATE TABLE `city` (
   `id_city` INT NOT NULL AUTO_INCREMENT,
   `name_city` CHAR(255) NOT NULL,
   PRIMARY KEY  (`id_city`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8"""
sql_2 = "INSERT INTO `city` VALUES (NULL, 'Санкт-Петербург')"
try:
    cur.execute("SET NAMES utf8") # Кодировка соединения
    cur.execute(sql_1)
    cur.execute(sql_2)
except MySQLdb.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
    con.commit()
cur.close()
con.close()
raw_input()
```

В этом примере мы использовали метод `commit()` объекта соединения. Метод `commit()` позволяет завершить транзакцию, которая запускается автоматически. При использовании транзакций в MySQL существуют нюансы. Таблица типа `MyISAM`, которую мы использовали в этом примере, не поддерживает транзакции. Поэтому вызов метода `commit()` можно опустить. Как видно из примера, указание метода не приводит в ошибке. Попытка отменить изменения с помощью метода `rollback()` не приведет к желаемому результату. В некоторых случаях использование этого метода может возбудить исключение `NotSupportedError`. Таблицы типа `InnoDB` транзакции поддерживают, поэтому все запросы, изменяющие записи (`INSERT`, `REPLACE`, `UPDATE` и `DELETE`), необходимо завершать вызовом метода `commit()`. Отменить изменения можно с помощью метода `rollback()`. Чтобы транзакции завершались без вызова метода `commit()`, следует указать значение `True` в методе `autocommit()` объекта соединения:

```
con.autocommit(True) # Автоматическое завершение транзакции
```

В некоторых случаях в SQL-запрос необходимо подставлять данные, полученные от пользователя. Если данные не обработать и подставить в SQL-запрос, то пользователь получает возможность видоизменить запрос и, например, зайти в закрытый раздел без ввода пароля. Чтобы значения были правильно подставлены, необходимо их передавать в виде кортежа или словаря во втором параметре метода `execute()`.

В этом случае в SQL-запросе указываются следующие специальные заполнители:

- `%s` — при указании значения в виде кортежа;
- `%(<Ключ>)s` — при указании значения в виде словаря.

В качестве примера заполним таблицу с городами этими способами:

```
# -*- coding: utf-8 -*-
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", db="python")
con.autocommit(True) # Автоматическое завершение транзакции
cur = con.cursor()
t1 = ("Москва",)     # Запятая в конце обязательна!
t2 = (3, "Новгород")
d  = {"id": 4, "name": """Новый ' " город"""}
sql_t1 = "INSERT INTO `city` (`name_city`) VALUES (%s)"
sql_t2 = "INSERT INTO `city` VALUES (%s, %s)"
sql_d  = "INSERT INTO `city` VALUES (%(id)s, %(name)s)"
try:
    cur.execute("SET NAMES utf8") # Кодировка соединения
    cur.execute(sql_t1, t1)     # Кортеж из 1-го элемента
    cur.execute(sql_t2, t2)     # Кортеж из 2-х элементов
    cur.execute(sql_d,  d)      # Словарь
except MySQLdb.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()
con.close()
raw_input()
```

Обратите внимание на значение переменной `t1`. Перед закрывающей круглой скобкой запятая указана не по ошибке. Если запятую убрать, то вместо кортежа мы получим строку. В значении ключа `name` переменной `d` апостроф и двойная кавычка также указаны не случайно. Это значение показывает, что при подстановке все специальные символы экранируются, поэтому никакой ошибки при вставке значения в таблицу не будет.

Внимание!

Никогда напрямую не передавайте в SQL-запрос данные, полученные от пользователя. Это потенциальная угроза безопасности. Данные следует передавать через второй параметр метода `execute()`.

- `executemany(<SQL-запрос>, <Последовательность>)` — выполняет SQL-запрос несколько раз, при этом подставляя значения из последовательности. Каждый элемент последовательности должен быть кортежем (используется заполнитель `"%s"`). Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение.

- Добавим два города с помощью метода `executemany()`:

```
# -*- coding: utf-8 -*-
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", db="python")
con.autocommit(True) # Автоматическое завершение транзакции
cur = con.cursor()
arr = [ ("Пермь",), ("Самара",) ]
sql = "INSERT INTO `city` (`name_city`) VALUES (%s)"
try:
    cur.execute("SET NAMES utf8") # Кодировка соединения
    cur.executemany(sql, arr)     # Выполняем запрос
except MySQLdb.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()
con.close()
raw_input()
```

Объект-курсор поддерживает несколько атрибутов:

- `lastrowid` — индекс последней добавленной записи с помощью инструкции `INSERT` и метода `execute()`. Вместо атрибута `lastrowid` можно воспользоваться методом `insert_id()` объекта соединения. В качестве примера добавим новый город и выведем его индекс двумя способами:

```
# -*- coding: utf-8 -*-
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", db="python")
con.autocommit(True) # Автоматическое завершение транзакции
cur = con.cursor()
sql = "INSERT INTO `city` (`name_city`) VALUES ('Омск')"
try:
    cur.execute("SET NAMES utf8") # Кодировка соединения
    cur.execute(sql)
except MySQLdb.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
    print u"Индекс:", cur.lastrowid
    print u"Индекс:", con.insert_id()
cur.close()
con.close()
raw_input()
```

- `rowcount` — количество измененных или удаленных записей, а также количество записей, возвращаемых инструкцией `SELECT`;

- description — содержит кортеж кортежей с опциями полей в результате выполнения инструкции SELECT. Каждый внутренний кортеж состоит из семи элементов. Первый элемент содержит название поля. Например, если выполнить SQL-запрос SELECT * FROM `city`, то атрибут будет содержать следующее значение:

```
(('id_city', 3, 1, 11, 11, 0, 0),
('name_city', 254, 29, 765, 765, 0, 0))
```

18.1.3. Обработка результата запроса

Для обработки результата запроса применяются следующие методы курсора MySQLdb.cursors.Cursor:

- fetchone() — при каждом вызове возвращает одну запись из результата запроса в виде кортежа, а затем перемещает указатель текущей позиции. Если записей больше нет, метод возвращает значение None. Выведем две первые записи из таблицы с городами:

```
>>> import MySQLdb
>>> con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", db="python")
>>> cur = con.cursor()
>>> cur.execute("SET NAMES utf8")
0L
>>> sql = "SELECT `name_city` FROM `city` WHERE `id_city`<3"
>>> cur.execute(sql)
2L
>>> cur.rowcount      # Количество записей
2L
>>> con.field_count() # Количество полей
1
>>> cur.fetchone()
('\xd0\xa1\xd0\xb0\xd0\xbd\xd0\xba\xd1\x82-
\xd0\x9f\xd0\xb5\xd1\x82\xd0\xb5\xd1\x80\xd0\xb1\xd1\x83\xd1
\x80\xd0\xb3',)
>>> cur.fetchone()
('\xd0\x9c\xd0\xbe\xd1\x81\xd0\xba\xd0\xb2\xd0\xb0',)
>>> print cur.fetchone()
None
```

Как видно из примера, метод execute() при выполнении запроса SELECT возвращает количество записей в виде длинного целого числа. Получить количество записей можно также с помощью атрибута rowcount объекта-курсора. Узнать количество полей в результате запроса позволяет метод field_count() объекта соединения;

- `fetchmany([size=cursor.arraysize])` — при каждом вызове возвращает кортеж записей из результата запроса, а затем перемещает указатель текущей позиции. Каждый элемент кортежа также является кортежем. Количество элементов, выбираемых за один раз, задается с помощью необязательного параметра или значения атрибута `arraysize` объекта-курсора. Если количество записей в результате запроса меньше указанного количества элементов, то количество элементов кортежа будет соответствовать оставшемуся количеству записей. Если записей больше нет, метод возвращает пустой кортеж. Пример:

```
>>> sql = "SELECT `name_city` FROM `city` WHERE `id_city`>3"
>>> cur.execute(sql)
4L
>>> cur.arraysize
1
>>> cur.fetchmany()
(('\xd0\x9d\xd0\xbe\xd0\xb2\xd1\x8b\xd0\xb9 \' "
\xd0\xb3\xd0\xbe\xd1\x80\xd0\xbe\xd0\xb4',),)
>>> cur.fetchmany(2)
(('\xd0\x9f\xd0\xb5\xd1\x80\xd0\xbc\xd1\x8c',),
('\xd0\xa1\xd0\xb0\xd0\xbc\xd0\xb0\xd1\x80\xd0\xb0',))
>>> cur.fetchmany(3)
(('\xd0\x9e\xd0\xbc\xd1\x81\xd0\xba',),)
>>> cur.fetchmany()
()
```

- `fetchall()` — возвращает кортеж всех (или всех оставшихся) записей из результата запроса. Каждый элемент кортежа также является кортежем. Если записей больше нет, метод возвращает пустой кортеж. Пример:

```
>>> sql = "SELECT `name_city` FROM `city` WHERE `id_city`>4"
>>> cur.execute(sql)
3L
>>> cur.fetchall()
(('\xd0\x9f\xd0\xb5\xd1\x80\xd0\xbc\xd1\x8c',),
('\xd0\xa1\xd0\xb0\xd0\xbc\xd0\xb0\xd1\x80\xd0\xb0',),
('\xd0\x9e\xd0\xbc\xd1\x81\xd0\xba',))
>>> cur.fetchall()
()
```

Все рассмотренные методы после возвращения результата перемещают указатель текущей позиции. Если необходимо вернуться в начало или переместить указатель к произвольной записи, то следует воспользоваться методом `scroll(<Смещение>, <Точка отсчета>)`. Во втором параметре могут быть указаны значения `"absolute"` (абсолютное положение) или `"relative"` (относительно текущей позиции указателя). Если указанное смещение выходит за диапазон, то возбуждается исключение `IndexError`. В качестве примера переместим указатель в начало, выведем все записи, а затем вернемся на одну запись назад (листинг 18.4).

Листинг 18.4. Перемещение указателя текущей позиции

```
>>> cur.scroll(0, "absolute")
>>> res = cur.fetchall()
>>> for name in res: print name[0].decode("utf-8").encode("cp1251")

Пермь
Самара
Омск
>>> cur.scroll(-1, "relative")
>>> res = cur.fetchall()
>>> for name in res: print name[0].decode("utf-8").encode("cp1251")

Омск
```

Объект-курсор поддерживает итерационный протокол. Поэтому можно перебрать записи с помощью цикла `for`, указав объект-курсор в качестве параметра:

```
>>> sql = "SELECT `name_city` FROM `city` WHERE `id_city`>5"
>>> cur.execute(sql)
2L
>>> for row in cur: print row[0].decode("utf-8").encode("cp1251")

Самара
Омск
```

Во всех предыдущих результатах названия городов выводились в виде обычных строк в кодировке соединения. Чтобы результатом была Unicode-строка, необходимо указать кодировку соединения с помощью параметра `charset` функции `connect()` или метода `set_character_set(<Кодировка>)` объекта соединения. В этом случае параметр `use_unicode` функции `connect()` автоматически получит значение `True`, и значения, хранящиеся в полях `CHAR`, `VARCHAR` и `TEXT`, будут возвращаться в виде Unicode-строк. Если кодировка соединения указана в конфигурационном файле MySQL, то достаточно указать значение `True` в параметре `use_unicode`. Пример указания кодировки соединения приведен в листинге 18.5.

Листинг 18.5. Указание кодировки соединения

```
>>> ini = r"C:\Program Files\MySQL\MySQL Server 5.1\my.ini"
>>> con = MySQLdb.connect(host="localhost", user="root",
          passwd="123456", read_default_file=ini, charset="utf8")
>>> con.select_db("python")
>>> cur = con.cursor()
>>> sql = "SELECT `name_city` FROM `city` WHERE `id_city`>5"
>>> cur.execute(sql)
2L
>>> cur.fetchone()
```

```
(u'\u0421\u0430\u043c\u0430\u0440\u0430',)
>>> print cur.fetchone()[0]
Омск
>>> con.close()
```

Обратите внимание на то, что задание кодировки требует указания пути к файлам с кодовыми таблицами в директиве `character-sets-dir` в конфигурационном файле MySQL. Путь к конфигурационному файлу задается в параметре `read_default_file`. В этом примере мы используем кодировку UTF-8, поэтому путь к конфигурационному файлу можно было и не указывать. Кодировка UTF-8 используется в MySQL по умолчанию и не требует наличия файла с кодовой таблицей.

Все рассмотренные методы возвращают запись в виде кортежа. Если необходимо изменить такое поведение и получить записи в виде словаря, то следует воспользоваться курсором `MySQLdb.cursors.DictCursor`. Этот курсор аналогичен курсору `MySQLdb.cursors.Cursor`, но возвращает записи в виде словаря, а не кортежа. В качестве примера выведем запись с идентификатором 5 в виде словаря (листинг 18.6).

Листинг 18.6. Получение записей в виде словаря

```
>>> con = MySQLdb.connect(host="localhost", user="root",
          passwd="123456", db="python", charset="utf8")
>>> cur = con.cursor(MySQLdb.cursors.DictCursor)
>>> sql = "SELECT * FROM `city` WHERE `id_city`=5"
>>> cur.execute(sql)
1L
>>> cur.fetchone()
{'name_city': u'\u041f\u0435\u0440\u043c\u044c', 'id_city': 5L}
>>> con.close()
```

18.2. Модуль *PyODBC*

Модуль `PyODBC` позволяет работать с базами данных Access, SQL Server, MySQL и с таблицами Excel. В этом разделе мы рассмотрим возможности модуля применительно к базе данных MySQL. Для установки модуля `PyODBC` со страницы **http://code.google.com/p/pyodbc/downloads/list** скачиваем файл pyodbc-2.1.7.win32-py2.6.exe, а затем запускаем его с помощью двойного щелчка на значке файла. Процесс установки предельно прост и в комментариях не нуждается. Чтобы проверить работоспособность модуля в окне **Python Shell** редактора IDLE, набираем следующий код:

```
>>> import pyodbc
>>> pyodbc.version
'2.1.7'
```

Модуль PyODBC предоставляет интерфейс доступа, совместимый со спецификацией DB-API. Получить номер поддерживаемой версии спецификации можно с помощью атрибута apilevel:

```
>>> pyodbc.apilevel
'2.0'
```

Прежде чем использовать модуль PyODBC, необходимо установить на компьютер драйвер ODBC для MySQL. Для этого переходим на страницу **http://www.mysql.com/downloads/connector/odbc/** и загружаем файл mysql-connector-odbc-5.1.6-win32.msi. Затем запускаем программу установки с помощью двойного щелчка на значке файла. После установки драйвера можно подключиться к MySQL.

18.2.1. Подключение к базе данных

Для подключения к базе данных используется функция connect(). Функция имеет следующий формат:

```
connect(<Строка подключения>[, autocommit=False]
                            [, unicode_results=False])
```

Функция connect() возвращает объект соединения, с помощью которого осуществляется вся дальнейшая работа с базой данных. Если подключиться не удалось, то возбуждается исключение. Соединение закрывается, когда вызывается метод close() объекта соединения. Рассмотрим наиболее важные параметры, указываемые в строке подключения:

- DRIVER — название драйвера. Для MySQL указывается значение "{MySQL ODBC 5.1 Driver}";
- SERVER — имя хоста. По умолчанию используется локальный хост;
- UID — имя пользователя;
- PWD — пароль для авторизации пользователя. По умолчанию пустой пароль;
- DATABASE — название базы данных, которую необходимо выбрать для работы;
- PORT — номер порта, на котором запущен сервер MySQL. Значение по умолчанию 3306;
- CHARSET — кодировка соединения.

Примечание

Более подробную информацию о параметрах подключения можно получить на странице **http://dev.mysql.com/doc/refman/5.1/en/connector-odbc-configuration-connection-parameters.html**.

В качестве примера подключимся к базе данных python, которую мы создали при изучении модуля MySQLdb:

```
>>> import pyodbc
>>> s = "DRIVER={MySQL ODBC 5.1 driver};SERVER=localhost;"
>>> s += "UID=root;PWD=123456;DATABASE=python;CHARSET=utf8"
>>> con = pyodbc.connect(s, autocommit=True, unicode_results=True)
>>> con.close()
```

Если параметр `autocommit` имеет значение `True`, то транзакции будут завершаться автоматически. Вместо параметра можно использовать метод `autocommit()` объекта соединения. Если автоматическое завершение транзакции отключено, то при использовании таблиц типа `InnoDB` все запросы, изменяющие записи (`INSERT`, `REPLACE`, `UPDATE` и `DELETE`), необходимо завершать вызовом метода `commit()`. Отменить изменения можно с помощью метода `rollback()`.

При указании в параметре `unicode_results` значения `True` значения, хранящиеся в полях `CHAR`, `VARCHAR` и `TEXT`, будут возвращаться в виде Unicode-строк. По умолчанию параметр имеет значение `False`.

18.2.2. Выполнение запроса

После подключения к базе данных необходимо создать объект-курсор с помощью метода `cursor()`. Для выполнения запроса к базе данных предназначены следующие методы объекта-курсора:

- `close()` — закрывает объект-курсор;
- `execute(<SQL-запрос>[, <Значения>])` — выполняет один SQL-запрос. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Метод возвращает объект-курсор. Создадим три таблицы в базе данных `python`:

```
# -*- coding: utf-8 -*-
import pyodbc
s = "DRIVER={MySQL ODBC 5.1 driver};SERVER=localhost;"
s += "UID=root;PWD=123456;DATABASE=python;CHARSET=utf8"
con = pyodbc.connect(s, autocommit=True, unicode_results=True)
cur = con.cursor()
sql_1 = """\
CREATE TABLE `user` (
   `id_user` INT AUTO_INCREMENT PRIMARY KEY,
   `email` VARCHAR(255),
   `passw` VARCHAR(255)
) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_general_ci
"""
sql_2 = """\
CREATE TABLE `rubr` (
   `id_rubr` INT AUTO_INCREMENT PRIMARY KEY,
   `name_rubr` VARCHAR(255)
) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_general_ci
"""
sql_3 = """\
CREATE TABLE `site` (
   `id_site` INT AUTO_INCREMENT PRIMARY KEY,
   `id_user` INT,
   `id_rubr` INT,
```

```
    `url` VARCHAR(255),
    `title` VARCHAR(255),
    `msg` TEXT,
    `iq` INT
) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_general_ci
"""
try:
    cur.execute(sql_1)
    cur.execute(sql_2)
    cur.execute(sql_3)
except pyodbc.Error, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()
con.close()
raw_input()
```

Если данные получены от пользователя, то подставлять их в SQL-запрос необходимо через второй параметр метода `execute()`. В этом случае данные проходят обработку и все специальные символы экранируются. Если данные не обработать и подставить в SQL-запрос, то пользователь получает возможность видоизменить запрос и, например, зайти в закрытый раздел без ввода пароля. В SQL-запросе место вставки обработанных данных помечается с помощью символа `?`, а сами данные передаются в виде кортежа (или просто числа, строки через запятую) во втором параметре метода `execute()`. В качестве примера заполним таблицу с рубриками и добавим нового пользователя:

```
# -*- coding: utf-8 -*-
import pyodbc
s = "DRIVER={MySQL ODBC 5.1 driver};SERVER=localhost;"
s += "UID=root;PWD=123456;DATABASE=python;CHARSET=utf8"
con = pyodbc.connect(s, autocommit=True, unicode_results=True)
cur = con.cursor()
sql_1 = "INSERT INTO `user` (`email`, `passw`) VALUES (?, ?)"
sql_2 = "INSERT INTO `rubr` (`name_rubr`) VALUES (?)"
sql_3 = "INSERT INTO `rubr` VALUES (NULL, ?)"
try:
    cur.execute(sql_1, ('unicross@mail.ru', 'password1'))
    cur.execute(sql_2, ("Программирование",))
    cur.execute(sql_3, """Поисковые ' " порталы""")
except pyodbc.Error, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()
con.close()
raw_input()
```

- `executemany(<SQL-запрос>, <Последовательность>)` — выполняет SQL-запрос несколько раз, при этом подставляя значения из последовательности. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Заполним таблицу `site` с помощью метода `executemany()`:

```
# -*- coding: utf-8 -*-
import pyodbc
s = "DRIVER={MySQL ODBC 5.1 driver};SERVER=localhost;"
s += "UID=root;PWD=123456;DATABASE=python;CHARSET=utf8"
con = pyodbc.connect(s, autocommit=True, unicode_results=True)
cur = con.cursor()
arr = [
         (1, 1, "http://wwwadmin.ru", "Название", "", 100),
         (1, 1, "http://python.org", "Python", "", 1000),
         (1, 2, "http://google.ru", "Гугль", "", 3000)
      ]
sql = """INSERT INTO `site` \
(`id_user`, `id_rubr`, `url`, `title`, `msg`, `iq`) \
VALUES (?, ?, ?, ?, ?, ?)"""
try:
    cur.executemany(sql, arr)
except pyodbc.Error, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()
con.close()
raw_input()
```

18.2.3. Обработка результата запроса

Для обработки результата запроса применяются следующие методы объекта-курсора:

- `fetchone()` — при каждом вызове возвращает одну запись из результата запроса в виде объекта `Row`, а затем перемещает указатель текущей позиции. Если записей больше нет, метод возвращает значение `None`. Выведем записи из таблицы с рубриками:

```
>>> import pyodbc
>>> s = "DRIVER={MySQL ODBC 5.1 driver};SERVER=localhost;"
>>> s += "UID=root;PWD=123456;DATABASE=python;CHARSET=utf8"
>>> con = pyodbc.connect(s,autocommit=True,unicode_results=True)
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM `rubr`")
<pyodbc.Cursor object at 0x011C8CD0>
>>> row = cur.fetchone()
```

```
>>> row.id_rubr             # Доступ по названию поля
1
>>> print row.name_rubr     # Доступ по названию поля
Программирование
>>> print row[1]            # Доступ по индексу поля
Программирование
>>> cur.fetchone()
(2, u'\u041f\u043e\u0438\0441\u043a\u043e\u0432\u044b\u0435
\' " \u043f\u043e\u0440\u0442\u0430\u043b\u044b')
>>> print cur.fetchone()
None
```

Как видно из примера, объект `Row`, возвращаемый методом `fetchone()`, позволяет получить значение как по индексу, так и по названию поля, которое указывается через точку. Если вывести полностью содержимое объекта, то возвращается кортеж со значениями. Так как при подключении мы указали значение `True` в параметре `unicode_results`, все строковые значения возвращаются в виде Unicode-строк;

- `fetchmany([size=cursor.arraysize])` — при каждом вызове возвращает список записей из результата запроса, а затем перемещает указатель текущей позиции. Каждый элемент списка является объектом `Row`. Количество элементов, выбираемых за один раз, задается с помощью необязательного параметра или значения атрибута `arraysize` объекта-курсора. Если количество записей в результате запроса меньше указанного количества элементов, то количество элементов списка будет соответствовать оставшемуся количеству записей. Если записей больше нет, метод возвращает пустой список. Пример:

```
>>> cur.execute("SELECT * FROM `rubr`")
<pyodbc.Cursor object at 0x011C8CD0>
>>> cur.arraysize
1
>>> row = cur.fetchmany()[0]
>>> print row.name_rubr
Программирование
>>> cur.fetchmany(2)
[(2, u'\u041f\u043e\u0438\0441\u043a\u043e\u0432\u044b\u0435
\' " \u043f\u043e\u0440\u0442\u0430\u043b\u044b')]
>>> cur.fetchmany()
[]
```

- `fetchall()` — возвращает список всех (или всех оставшихся) записей из результата запроса. Каждый элемент списка является объектом `Row`. Если записей больше нет, метод возвращает пустой список. Пример:

```
>>> cur.execute("SELECT * FROM `rubr`")
<pyodbc.Cursor object at 0x011C8CD0>
>>> rows = cur.fetchall()
>>> rows
```

```
[(1, u'\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c
\u0438\u0440\u043e\u0432\u0430\u043d\u0438\u0435'),
(2, u'\u041f\u043e\u0438\u0441\u043a\u043e\u0432\u044b\u0435
\' " \u043f\u043e\u0440\u0442\u0430\u043b\u044b')]
>>> print rows[0].name_rubr
Программирование
>>> cur.fetchall()
[]
```

Объект-курсор поддерживает итерационный протокол. Поэтому можно перебрать записи с помощью цикла `for`, указав объект-курсор в качестве параметра:

```
>>> cur.execute("SELECT * FROM `rubr`")
<pyodbc.Cursor object at 0x011C8CD0>
>>> for row in cur: print row.name_rubr

Программирование
Поисковые ' " порталы
```

Объект-курсор поддерживает несколько атрибутов:

- `rowcount` — количество измененных или удаленных записей. Изменим название рубрики с идентификатором 2 и выведем количество изменений:

```
>>> cur.execute("""UPDATE `rubr`
                   SET `name_rubr`='Поисковые порталы'
                   WHERE `id_rubr`=2""")
<pyodbc.Cursor object at 0x011C8CD0>
>>> cur.rowcount
1
>>> cur.execute("SELECT * FROM `rubr` WHERE `id_rubr`=2")
<pyodbc.Cursor object at 0x011C8CD0>
>>> print cur.fetchone().name_rubr
Поисковые порталы
```

- `description` — содержит кортеж кортежей с опциями полей в результате выполнения инструкции `SELECT`. Каждый внутренний кортеж состоит из семи элементов. Первый элемент содержит название поля. Пример:

```
>>> cur.execute("SELECT * FROM `rubr`")
<pyodbc.Cursor object at 0x011C8CD0>
>>> cur.description
(('id_rubr', <type 'int'>, None, 10, 10, 0, True),
('name_rubr', <type 'unicode'>, None, 255, 255, 0, True))
```

Мы уже не раз говорили, что передавать значения, введенные пользователем, необходимо через второй параметр метода `execute()`. Если данные не обработать и подставить в SQL-запрос, то пользователь получает возможность видоизменить запрос и, например, зайти в закрытый раздел без ввода пароля. В качестве примера составим SQL-запрос с помощью форматирования и зайдем под учетной записью пользователя без ввода пароля (листинг 18.7).

Листинг 18.7. Видоизменение SQL-запроса извне

```
>>> user = "unicross@mail.ru'/*"
>>> passw = "*/ '"
>>> sql = """SELECT * FROM `user`
             WHERE `email`='%s' AND `passw`='%s'""" % (user, passw)
>>> cur.execute(sql)
<pyodbc.Cursor object at 0x011C8CD0>
>>> cur.fetchone()
(1, u'unicross@mail.ru', u'password1')
```

Как видно из результата, мы получили доступ не зная пароля. После форматирования SQL-запрос будет выглядеть следующим образом:

```
SELECT * FROM `user` WHERE `email`='unicross@mail.ru'/*'
                           AND `passw`='*/ ''
```

Все, что расположено между `/*` и `*/`, является комментарием. В итоге SQL-запрос будет выглядеть так:

```
SELECT * FROM `user` WHERE `email`='unicross@mail.ru' ''
```

Никакая проверка пароля в данном случае вообще не производится. Достаточно знать логин пользователя, и можно войти без пароля. Если данные передавать через второй параметр метода `execute()`, то все специальные символы будут экранированы и пользователь не сможет видоизменить SQL-запрос (листинг 18.8).

Листинг 18.8. Правильная передача данных в SQL-запрос

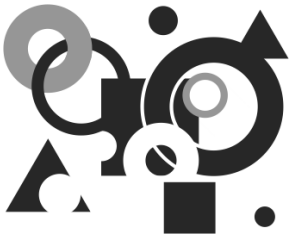
```
>>> user = "unicross@mail.ru'/*"
>>> passw = "*/ '"
>>> sql = "SELECT * FROM `user` WHERE `email`=? AND `passw`=?"
>>> cur.execute(sql, (user, passw))
<pyodbc.Cursor object at 0x011C8CD0>
>>> print cur.fetchone()
None
```

После подстановки значений, SQL-запрос будет выглядеть следующим образом:

```
SELECT * FROM `user` WHERE `email`='unicross@mail.ru\'/*'
                           AND `passw`='*/ \''
```

В результате все опасные символы были экранированы.

Глава 19



Библиотека PIL. Работа с изображениями

Для работы с изображениями в Python наиболее часто используется библиотека PIL (Python Imaging Library). В этой главе мы рассмотрим базовые возможности данной библиотеки, применяемые наиболее часто. Для установки библиотеки со страницы **http://effbot.org/downloads/#pil** скачиваем файл PIL-1.1.6.win32-py2.6.exe, а затем запускаем его с помощью двойного щелчка на значке файла. Процесс установки очень прост и в комментариях не нуждается. Чтобы проверить работоспособность библиотеки, в окне **Python Shell** редактора IDLE набираем следующий код:

```
>>> from PIL import Image
>>> Image.VERSION
'1.1.6'
```

Примечание

При работе с версией PIL-1.1.7.win32-py2.6.exe под Windows XP возникли проблемы. Поэтому в этой главе мы будем рассматривать версию 1.1.6, а не 1.1.7. Вполне возможно, что в вашей версии библиотеки ошибки были исправлены.

19.1. Загрузка готового изображения

Для открытия файла с готовым изображением применяется функция `open()`. Функция возвращает объект, с помощью которого производится дальнейшая работа с изображением. Если открыть файл с изображением не удалось, возбуждается исключение `IOError`. Формат функции:

```
open(<Путь или файловый объект>[, mode='r'])
```

В первом параметре можно указать абсолютный или относительный путь к изображению. Откроем файл foto.gif, который расположен в текущем рабочем каталоге (листинг 19.1).

Листинг 19.1. Загрузка готового изображения

```
>>> img = Image.open("foto.gif")
```

Вместо указания пути к файлу можно передать файловый объект, открытый в бинарном режиме. Пример:

```
>>> f = open("foto.gif", "rb") # Открываем файл в бинарном режиме
>>> img = Image.open(f)        # Передаем объект файла
>>> img.size                   # Получаем размер изображения
(800, 600)
>>> f.close()                  # Закрываем файл
```

Если изображение было загружено в какую-либо переменную, то можно создать файловый объект с помощью модуля `StringIO` и передать его в функцию `open()` (листинг 19.2).

Листинг 19.2. Загрузка изображения из строки

```
>>> f = open("foto.gif", "rb") # Открываем файл в бинарном режиме
>>> i = f.read()               # Сохраняем изображение в переменной
>>> f.close()                  # Закрываем файл
>>> import StringIO            # Подключаем модуль StringIO
>>> img = Image.open(StringIO.StringIO(i)) # Передаем объект
>>> img.format                 # Выводим формат изображения
'GIF'
```

Как видно из примера, формат изображения определяется автоматически. Следует также заметить, что после открытия файла с помощью функции `open()` само изображение не загружается сразу в память из файла. Загрузка изображения производится при первой операции с изображением. Загрузить изображение явным образом позволяет метод `load()`. Хотя в большинстве случаев это делать не нужно. Начиная с версии 1.1.6 метод `load()` возвращает объект, с помощью которого можно получить доступ к отдельным пикселам изображения. Указав два значения внутри квадратных скобок, можно получить или задать цвет пиксела (листинг 19.3).

Листинг 19.3. Получение и изменение цвета пиксела

```
>>> img = Image.open("foto.jpg")
>>> obj = img.load()
>>> obj[25, 45]                    # Получаем цвет пиксела
(122, 86, 62)
>>> obj[25, 45] = (255, 0, 0)      # Задаем цвет пиксела (красный)
```

Для доступа к отдельному пикселу вместо метода `load()` можно использовать методы `getpixel()` и `putpixel()`. Метод `getpixel(<Координаты>)` позволяет получить цвет указанного пиксела, а метод `putpixel(<Координаты>, <Цвет>)` изменяет

цвет пиксела. Координаты пиксела указываются в виде кортежа из двух элементов. Необходимо заметить, что эти методы работают медленнее метода `load()`. Пример использования методов `getpixel()` и `putpixel()` приведен в листинге 19.4.

Листинг 19.4. Использование методов `getpixel()` и `putpixel()`

```
>>> img = Image.open("foto.jpg")
>>> img.getpixel((25, 45))              # Получаем цвет пиксела
(122, 86, 62)
>>> img.putpixel((25, 45), (255, 0, 0)) # Изменяем цвет пиксела
>>> img.getpixel((25, 45))              # Получаем цвет пиксела
(255, 0, 0)
>>> img.show()                          # Просматриваем изображение
```

В этом примере для просмотра изображения мы воспользовались методом `show()`. Метод `show()` создает временный файл в формате BMP и запускает программу для просмотра изображений, используемую в операционной системе по умолчанию. Например, на моем компьютере запускается программа ACDSee.

Для сохранения изображения в файл предназначен метод `save()`. Формат метода:

```
save(<Путь или файловый объект>[, <Формат>[, <Опции>]])
```

В первом параметре указывается абсолютный или относительный путь. Вместо пути можно передать файловый объект, открытый в бинарном режиме. Сохраним изображение в форматах JPEG и BMP разными способами (листинг 19.5).

Листинг 19.5. Сохранение изображения

```
>>> img.save("tmp.jpg")          # В формате JPEG
>>> img.save("tmp.bmp", "BMP")   # В формате BMP
>>> f = open("tmp2.bmp", "wb")
>>> img.save(f, "BMP")           # Передаем файловый объект
>>> f.close()
```

Обратите внимание на то, что мы открыли файл в формате JPEG, а сохранили его в формате BMP. Таким образом, можно открывать изображения в одном формате и конвертировать его в другой формат. Если сохранить изображение не удалось, возбуждается исключение `IOError`. Если параметр `<Формат>` не указан, то формат изображения определяется по расширению файла.

В параметре `<Опции>` можно указать дополнительные опции. Поддерживаемые опции зависят от формата изображения. Например, по умолчанию изображения в формате JPEG сохраняются с качеством 75. С помощью опции `quality` можно указать другое значение в диапазоне от 1 до 100. Пример:

```
>>> img.save("tmp3.jpg", "JPEG", quality=100) # Указание качества
```

За дополнительной информацией по опциям обращайтесь к документации.

19.2. Создание нового изображения

Библиотека PIL позволяет работать не только с готовыми изображениями, но и создавать изображения. Создать новое изображение позволяет функция `new()`. Функция имеет следующий формат:

```
new(<Режим>, <Размер>[, <Цвет фона>])
```

В параметре `<Режим>` указывается один из режимов:

- `1` — 1 бит, черно-белое;
- `L` — 8 бит, черно-белое;
- `P` — 8 бит, цветное (256 цветов);
- `RGB` — 24 бита, цветное;
- `RGBA` — 32 бита, цветное с альфа-каналом;
- `CMYK` — 32 бита, цветное;
- `YCbCr` — 24 бита, цветное, видеоформат;
- `I` — 32 бита, целое число, цветное;
- `F` — 32 бита, вещественное число, цветное.

Во втором параметре необходимо передать размер холста в виде кортежа из двух элементов `(<Ширина>, <Высота>)`. В необязательном параметре `<Цвет фона>` задается цвет фона. Если параметр не указан, то изображение будет черного цвета. Для режима `RGB` цвет указывается в виде кортежа из трех цифр от 0 до 255 `(<Доля красного>, <Доля зеленого>, <Доля синего>)`. Кроме того, можно указать название цвета на английском языке и строки в форматах `"#RGB"` и `"#RRGGBB"`. Различные способы указания цвета приведены в листинге 19.6.

Листинг 19.6. Способы указания цвета

```
>>> img = Image.new("RGB", (100, 100))
>>> img.show()                              # Черный квадрат
>>> img = Image.new("RGB", (100, 100), (255, 0, 0))
>>> img.show()                              # Красный квадрат
>>> img = Image.new("RGB", (100, 100), "green")
>>> img.show()                              # Зеленый квадрат
>>> img = Image.new("RGB", (100, 100), "#f00")
>>> img.show()                              # Красный квадрат
>>> img = Image.new("RGB", (100, 100), "#ff0000")
>>> img.show()                              # Красный квадрат
```

19.3. Получение информации об изображении

Получить информацию об изображении позволяют следующие атрибуты:

- `size` — размер изображения в виде кортежа из двух элементов `(<Ширина>, <Высота>)`;

- `format` — формат изображения (например, GIF, JPEG и т. д.);
- `mode` — режим (например, `P`, `RGB`, `CMYK` и т. д.);
- `info` — дополнительная информация об изображении в виде словаря.

В качестве примера выведем информацию об изображениях в форматах JPEG, GIF, BMP, TIFF и PNG (листинг 19.7).

Листинг 19.7. Получение информации об изображении

```
>>> img = Image.open("foto.jpg")
>>> img.size, img.format, img.mode
((800, 600), 'JPEG', 'RGB')
>>> img.info
{'jfif': 258, 'jfif_unit': 0, 'adobe': 100, 'progression': 1,
'jfif_version': (1, 2), 'adobe_transform': 100,
'jfif_density': (100, 100)}
>>> img = Image.open("foto.gif")
>>> img.size, img.format, img.mode
((800, 600), 'GIF', 'P')
>>> img.info
{'version': 'GIF89a', 'background': 254}
>>> img = Image.open("foto.bmp")
>>> img.size, img.format, img.mode
((800, 600), 'BMP', 'RGB')
>>> img.info
{'compression': 0}
>>> img = Image.open("foto.tif")
>>> img.size, img.format, img.mode
((800, 600), 'TIFF', 'RGB')
>>> img.info
{'compression': 'raw'}
>>> img = Image.open("foto.png")
>>> img.size, img.format, img.mode
((800, 600), 'PNG', 'RGB')
>>> img.info
{'dpi': (72, 72)}
```

19.4. Манипулирование изображением

Произвести различные манипуляции с загруженным изображением позволяют следующие методы:

- `copy()` — создает копию изображения:

```
>>> from PIL import Image          # Подключаем модуль
>>> img = Image.open("foto.jpg") # Открываем файл
>>> img2 = img.copy()             # Создаем копию
>>> img2.show()                   # Просматриваем копию
```

- `thumbnail(<Размер>[, <Фильтр>])` — создает уменьшенную версию изображения указанного размера. Размер задается в виде кортежа из двух элементов `(<Ширина>, <Высота>)`. Обратите внимание на то, что изменение размера производится пропорционально. Иными словами, за основу берется минимальное значение, а второе значение вычисляется пропорционально первому. В параметре `<Фильтр>` могут быть указаны фильтры `NEAREST`, `BILINEAR`, `BICUBIC` или `ANTIALIAS`. Метод изменяет само изображение и ничего не возвращает. Пример:

```
>>> img = Image.open("foto.jpg")
>>> img.size # Исходные размеры изображения
(800, 600)
>>> img.thumbnail((400, 300), Image.ANTIALIAS)
>>> img.size # Изменяется само изображение
(400, 300)
>>> img = Image.open("foto.jpg")
>>> img.thumbnail((400, 100), Image.ANTIALIAS)
>>> img.size # Размер изменяется пропорционально
(133, 100)
```

- `resize(<Размер>[, <Фильтр>])` — изменяет размер изображения. В отличие от метода `thumbnail()` возвращает новое изображение, а не изменяет исходное изображение. Изменение размера производится не пропорционально. Иными словами, если пропорции не соблюдены, то изображение будет искажено. В параметре `<Фильтр>` могут быть указаны фильтры `NEAREST`, `BILINEAR`, `BICUBIC` или `ANTIALIAS`. Пример:

```
>>> img = Image.open("foto.jpg")
>>> img.size # Исходные размеры изображения
(800, 600)
>>> img2 = img.resize((400, 300), Image.ANTIALIAS)
>>> img2.size # Пропорциональное уменьшение
(400, 300)
>>> img3 = img.resize((400, 100), Image.ANTIALIAS)
>>> img3.size # Изображение будет искажено
(400, 100)
```

- `rotate(<Угол>[, <Фильтр>][, expand=0])` — поворачивает изображение на указанное количество градусов против часовой стрелки. Метод возвращает новое изображение. В параметре `<Фильтр>` могут быть указаны фильтры `NEAREST`, `BILINEAR` или `BICUBIC`. Если параметр `expand` имеет значение `True`, то размер изображения будет увеличен таким образом, чтобы оно полностью поместилось. По умолчанию размер изображения сохраняется. Если изображение не помещается, то оно будет обрезано. Пример:

```
>>> img = Image.open("foto.jpg")
>>> img.size # Исходные размеры изображения
(800, 600)
>>> img2 = img.rotate(90) # Поворот на 90 градусов
```

```
>>> img2.size
(600, 800)
>>> img3 = img.rotate(45, Image.NEAREST)
>>> img3.size # Размеры сохранены, изображение обрезано
(800, 600)
>>> img4 = img.rotate(45, expand=True)
>>> img4.size # Размеры увеличены, изображение полное
(991, 990)
```

- `transpose(<Преобразование>)` — возвращает зеркальный образ или поворачивает изображение. В качестве параметра можно указать значения `FLIP_LEFT_RIGHT`, `FLIP_TOP_BOTTOM`, `ROTATE_90`, `ROTATE_180` или `ROTATE_270`. Метод возвращает новое изображение. Пример:

```
>>> img = Image.open("foto.jpg")
>>> img2 = img.transpose(Image.FLIP_LEFT_RIGHT)
>>> img2.show() # Горизонтальный зеркальный образ
>>> img3 = img.transpose(Image.FLIP_TOP_BOTTOM)
>>> img3.show() # Вертикальный зеркальный образ
>>> img4 = img.transpose(Image.ROTATE_90)
>>> img4.show() # Поворот на 90 гр. против часовой стрелки
>>> img5 = img.transpose(Image.ROTATE_180)
>>> img5.show() # Поворот на 180 градусов
>>> img6 = img.transpose(Image.ROTATE_270)
>>> img6.show() # Поворот на 270 градусов
```

- `crop((<X1>, <Y1>, <X2>, <Y2>))` — считывает прямоугольную область из исходного изображения. В качестве параметра указывается кортеж из четырех элементов. Первые два элемента задают координату левого верхнего угла прямоугольной области, а вторые два элемента задают координату правого нижнего угла. Предполагается, что начало координат располагается в левом верхнем углу изображения. Положительная ось *x* направлена вправо, а положительная ось *y* — вниз. В качестве значения метод возвращает новое изображение. Обратите внимание на то, что считывание области из исходного изображения производится только при первой операции над новым изображением. Если после выполнения метода `crop()` над исходным изображением были произведены операции, то они отобразятся и на новом изображении. Чтобы явным образом произвести считывание области, необходимо сразу после метода `crop()` вызвать метод `load()`. Пример:

```
>>> img = Image.open("foto.jpg")
>>> img2 = img.crop( [0, 0, 100, 100] ) # Помечаем область
>>> img2.load() # Считываем область в новое изображение
>>> img2.size
(100, 100)
```

- `paste(<Цвет>, <Область>[, <Маска>])` — закрашивает прямоугольную область определенным цветом. Координаты области указываются в виде кортежа из четырех элементов. Первые два элемента задают координату левого верхне-

го угла прямоугольной области, а вторые два элемента — координату правого нижнего угла. Закрасим область красным цветом:

```
>>> img = Image.open("foto.jpg")
>>> img.paste( (255, 0, 0), (0, 0, 100, 100) )
>>> img.show()
```

Теперь зальем все изображение зеленым цветом:

```
>>> img = Image.open("foto.jpg")
>>> img.paste( (0, 128, 0), img.getbbox() )
>>> img.show()
```

В этом примере мы использовали метод `getbbox()`, который возвращает координаты прямоугольной области, в которую вписывается все изображение:

```
>>> img.getbbox()
(0, 0, 800, 600)
```

- `paste(<Изображение>, <Область>[, <Маска>])` — вставляет указанное изображение в прямоугольную область. Координаты области указываются в виде кортежа из двух или четырех элементов. Если указан кортеж из двух элементов, то он задает начальную точку. В качестве примера загрузим изображение, создадим уменьшенную копию, а затем вставим ее в исходное изображение, причем вокруг вставленного изображения отобразим рамку красного цвета:

```
>>> img = Image.open("foto.jpg")
>>> img2 = img.resize( (200, 150) ) # Создаем миниатюру
>>> img2.size
(200, 150)
>>> img.paste( (255, 0, 0), (9, 9, 211, 161) ) # Рамка
>>> img.paste(img2, (10, 10) )      # Вставляем миниатюру
>>> img.show()
```

Необязательный параметр `<Маска>` позволяет задать степень прозрачности вставляемого изображения или цвета. В качестве примера выведем белую полупрозрачную горизонтальную полосу высотой 100 пикселов:

```
>>> img = Image.open("foto.jpg")
>>> white = Image.new("RGB", (img.size[0],100), (255,255,255))
>>> mask = Image.new("L", (img.size[0], 100), 64) # Маска
>>> img.paste(white, (0, 0), mask)
>>> img.show()
```

- `split()` — возвращает каналы изображения в виде кортежа. Например, для изображения в режиме `RGB` возвращается кортеж из трех элементов `(R, G, B)`. Произвести обратную операцию (собрать изображение из каналов) позволяет функция `merge(<Режим>, <Каналы>)`. В качестве примера преобразуем изображение из режима `RGB` в режим `RGBA`:

```
>>> img = Image.open("foto.jpg")
>>> img.mode
'RGB'
>>> R, G, B = img.split()
```

```
>>> mask = Image.new("L", img.size, 128)
>>> img2 = Image.merge("RGBA", (R, G, B, mask) )
>>> img2.mode
'RGBA'
>>> img2.show()
```

- `convert(<Новый режим>[, <Матрица>])` — преобразует изображение в указанный режим. Возвращает новое изображение. Преобразуем изображение из режима `RGB` в режим `RGBA`:

```
>>> img = Image.open("foto.jpg")
>>> img.mode
'RGB'
>>> img2 = img.convert("RGBA")
>>> img2.mode
'RGBA'
>>> img2.show()
```

- `filter(<Фильтр>)` — применяет к изображению указанный фильтр. Метод возвращает новое изображение. В качестве параметра можно указать фильтры `BLUR`, `CONTOUR`, `DETAIL`, `EDGE_ENHANCE`, `EDGE_ENHANCE_MORE`, `EMBOSS`, `FIND_EDGES`, `SHARPEN`, `SMOOTH` и `SMOOTH_MORE` из модуля `ImageFilter`. Пример:

```
>>> import ImageFilter
>>> img = Image.open("foto.jpg")
>>> img2 = img.filter(ImageFilter.EMBOSS)
>>> img2.show()
```

19.5. Рисование линий и фигур

Чтобы на изображении можно было рисовать, необходимо создать экземпляр класса `Draw`, передав в конструктор класса ссылку на изображение. Прежде чем использовать класс, предварительно следует импортировать модуль `ImageDraw`. Пример создания экземпляра класса:

```
>>> from PIL import Image, ImageDraw # Подключаем модули
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = ImageDraw.Draw(img)       # Создаем экземпляр класса
```

Класс `Draw` предоставляет следующие методы:

- `point(<Координаты>, fill=<Цвет>)` — рисует точку. Нарисуем красную горизонтальную линию из нескольких точек:

```
>>> from PIL import Image, ImageDraw
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = ImageDraw.Draw(img)
>>> for n in xrange(5, 31):
        draw.point( (n, 5), fill=(255, 0, 0) )
>>> img.show()
```

- `line(<Координаты>, fill=<Цвет>[, width=<Ширина>])` — рисует линию между двумя координатами. Пример:

```
>>> draw.line( (0, 0, 0, 300), fill=(0, 128, 0) )
>>> draw.line( (297, 0, 297, 300), fill=(0, 128, 0), width=3 )
>>> img.show()
```

- `rectangle()` — рисует прямоугольник. Формат метода:

```
rectangle(<Координаты>[, fill=<Цвет заливки>]
                      [, outline=<Цвет линии>])
```

В параметре `<Координаты>` указываются координаты двух точек: левого верхнего угла и правого нижнего угла. Нарисуем три прямоугольника. Первый прямоугольник с рамкой и заливкой, второй — только с заливкой, а третий — только с рамкой:

```
>>> draw.rectangle( (10, 10, 30, 30), fill=(0, 0, 255),
                    outline=(0, 0, 0) )
>>> draw.rectangle( (40, 10, 60, 30), fill=(0, 0, 128))
>>> draw.rectangle( (0, 0, 299, 299), outline=(0, 0, 0))
>>> img.show()
```

- `polygon()` — рисует многоугольник. Формат метода:

```
polygon(<Координаты>[, fill=<Цвет заливки>]
                      [, outline=<Цвет линии>])
```

В параметре `<Координаты>` указываются координаты трех и более точек. Указанные точки соединяются линиями. Кроме того, проводится прямая линия между первой и последней точками. Пример:

```
>>> draw.polygon((50, 50, 150, 150, 50, 150), outline=(0,0,0),
                 fill=(255, 0, 0)) # Треугольник
>>> draw.polygon( (200, 200, 250, 200, 275, 250, 250, 300,
                  200, 300, 175, 250), fill=(255, 255, 0))
>>> img.show()
```

- `ellipse()` — рисует эллипс. Формат метода:

```
ellipse(<Координаты>[, fill=<Цвет заливки>]
                      [, outline=<Цвет линии>])
```

В параметре `<Координаты>` указываются координаты прямоугольника, в который необходимо вписать эллипс. Пример:

```
>>> draw.ellipse((100, 100, 200, 200), fill=(255, 255, 0))
>>> draw.ellipse((50, 170, 150, 300), outline=(0, 255, 255))
>>> img.show()
```

- `arc()` — рисует дугу. Формат метода:

```
arc(<Координаты>, <Начальный угол>, <Конечный угол>,
    fill=<Цвет линии>)
```

В параметре `<Координаты>` указываются координаты прямоугольника, в который необходимо вписать окружность. Второй и третий параметры задают на-

чальный и конечный угол, между которыми будет отображена дуга. Угол, равный 0, расположен в крайней правой точке. Увеличение производится по часовой стрелке от 0 до 360. Линия рисуется по часовой стрелке. Пример:

```
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = ImageDraw.Draw(img)
>>> draw.arc((10, 10, 290, 290), 180, 0, fill=(255, 0, 0))
>>> img.show()
```

- `chord()` — рисует замкнутую дугу. Формат метода:

```
chord(<Координаты>, <Начальный угол>, <Конечный угол>,
      [, fill=<Цвет заливки>][, outline=<Цвет линии>])
```

Метод `chord()` аналогичен методу `arc()`, но замыкает крайние точки дуги прямой линией. Пример:

```
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = ImageDraw.Draw(img)
>>> draw.chord((10, 10, 290, 290), 180, 0, fill=(255, 0, 0))
>>> draw.chord((10, 10, 290, 290), -90, 0, fill=(255, 255, 0))
>>> img.show()
```

- `pieslice()` — рисует замкнутый сектор. Формат метода:

```
pieslice(<Координаты>, <Начальный угол>, <Конечный угол>,
        [, fill=<Цвет заливки>][, outline=<Цвет линии>])
```

Метод `pieslice()` аналогичен методу `arc()`, но замыкает крайние точки дуги с центром окружности. Пример:

```
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = ImageDraw.Draw(img)
>>> draw.pieslice((10, 10, 290, 290), -90, 0, fill="red")
>>> img.show()
```

19.6. Модуль *aggdraw*

Если приглядеться к контурам фигур, созданных с помощью класса `ImageDraw` из библиотеки PIL, то можно заметить, что граница отображается в виде ступенек. Сделать контуры более гладкими позволяет модуль `aggdraw`. Модуль `aggdraw` не входит в состав стандартной библиотеки Python. Для установки модуля переходим на страницу **http://www.effbot.org/downloads/#aggdraw**, загружаем файл aggdraw-1.2a3-20060212.win32-py2.6.exe, а затем запускаем программу установки с помощью двойного щелчка на значке файла. Установка предельно проста и в комментариях не нуждается. Для проверки установки запускаем следующий код:

```
>>> import aggdraw
>>> aggdraw.VERSION
'1.2a3'
```

Чтобы увидеть разницу, нарисуем два круга. Первый круг с помощью метода `ellipse()` класса `ImageDraw`, а второй круг с помощью метода `ellipse()` из модуля `aggdraw` (листинг 19.8).

Листинг 19.8. Сравнение класса `ImageDraw` и модуля `aggdraw`

```
>>> import aggdraw
>>> from PIL import Image, ImageDraw
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = ImageDraw.Draw(img)
>>> draw.ellipse((0, 0, 150, 150), fill="red", outline="red")
>>> pen = aggdraw.Pen("red", 0.5)
>>> brush = aggdraw.Brush("red")
>>> draw2 = aggdraw.Draw(img)
>>> draw2.ellipse((150, 150, 300, 300), pen, brush)
>>> draw2.flush()
<PIL.Image.Image instance at 0x01666350>
>>> img.show()
```

Методика создания первого круга должна быть уже вам знакома. Вначале создаем экземпляр класса `Draw` и передаем в конструктор ссылку на изображение, а затем выводим круг с помощью метода `ellipse()`, указывая размер области, цвет границы и заливки.

При использовании модуля `aggdraw` методика немного другая. Вместо указания конкретного цвета границы и заливки необходимо создать объекты "перо" и "кисть" с помощью классов `Pen` и `Brush` соответственно. Перо используется для вывода границы фигуры, а кисть применяется для заливки. Конструкторы классов имеют следующий формат:

```
Pen(<Цвет>[, width=1][, opacity=255])
Brush(<Цвет>[, opacity=255])
```

Цвет может быть указан в виде кортежа, строки с названием цвета на английском языке или строки в форматах `"#RGB"` и `"#RRGGBB"`. Параметр `width` задает ширину линии, а параметр `opacity` — прозрачность.

Чтобы на изображении можно было рисовать, необходимо создать экземпляр класса `Draw`. Конструктор класса имеет два формата:

```
Draw(<Объект изображения>)
Draw(<Режим>, <Размер>[, <Цвет>])
```

В первом формате указывается объект изображения из модуля PIL. Именно этим форматом мы воспользовались в предыдущем примере. Второй формат позволяет создать изображение. В параметре `<Режим>` могут быть указаны режимы `L`, `RGB`, `RGBA`, `BGR` и `BGRA`. Во втором параметре передается размер холста в виде кортежа. Если третий параметр не указан, то изображение будет белого цвета. Для примера создадим новое изображение, нарисуем круг, а затем преобразуем изображение в формат библиотеки PIL и просмотрим его (листинг 19.9).

Листинг 19.9. Создание нового изображения с помощью модуля `aggdraw`

```
>>> import aggdraw
>>> from PIL import Image
>>> draw = aggdraw.Draw("RGB", (300, 300), (255, 255, 255))
>>> pen = aggdraw.Pen("red", 0.5)
>>> brush = aggdraw.Brush("red")
>>> draw.ellipse((10, 10, 290, 290), pen, brush)
>>> img = draw.tostring()
>>> img2 = Image.fromstring("RGB", (300, 300), img)
>>> img2.show()
```

Получить созданное изображение в виде строки позволяет метод `tostring()`. Преобразовать эту строку в объект изображения библиотеки PIL можно с помощью функции `fromstring()` из модуля `Image`.

Класс `Draw` из модуля `aggdraw` предоставляет следующие методы:

- `flush()` — обновляет изображение. Если изображение не обновить, то изменения отображаться не будут;
- `setantialias(<Флаг>)` — если указано значение `True`, то контуры изображения сглаживаются, а если `False`, то сглаживание не производится. Нарисуем два круга с отключенным и включенным сглаживанием:

```
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = aggdraw.Draw(img)
>>> pen = aggdraw.Pen((255, 0, 0), 0.5)
>>> brush = aggdraw.Brush("#ff0000")
>>> draw.setantialias(False) # Сглаживание отключено
>>> draw.ellipse((0, 0, 150, 150), pen, brush)
>>> draw.setantialias(True)  # Сглаживание включено
>>> draw.ellipse((150, 150, 300, 300), pen, brush)
>>> draw.flush()             # Обновляем изображение
<PIL.Image.Image instance at 0x01317378>
>>> img.show()
```

- `tostring()` — возвращает изображение в виде строки;
- `line(<Координаты>, <Перо>)` — рисует линию между координатами. Пример:

```
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> pen = aggdraw.Pen((0, 128, 0), 1)
>>> draw = aggdraw.Draw(img)
>>> draw.line((10, 10, 150, 10), pen)
>>> pen = aggdraw.Pen("green", 3)
>>> draw.line((10, 30, 150, 30), pen)
>>> draw.flush()
<PIL.Image.Image instance at 0x011C6760>
>>> img.show()
```

В параметре <Координаты> можно указать координаты сразу нескольких точек. В этом случае они соединяются линиями. Первая и последняя точки не соединяются. Пример:

```
>>> draw.line((0, 0, 20, 100, 30, 50, 40, 100, 60, 0), pen)
>>> draw.flush()
<PIL.Image.Image instance at 0x011C6760>
>>> img.show()
```

- `rectangle(<Координаты>, <Перо>, <Кисть>)` — рисует прямоугольник. В параметре <Координаты> указываются координаты двух точек: левого верхнего угла и правого нижнего угла. Перо используется для рисования контура, а кисть для заливки. Пример:

```
>>> pen = aggdraw.Pen("#000000")
>>> brush = aggdraw.Brush("orange")
>>> draw.rectangle((40, 10, 60, 30), pen, brush)
>>> draw.rectangle((0, 0, 299, 299), pen)
>>> draw.flush()
<PIL.Image.Image instance at 0x011C6760>
>>> img.show()
```

- `polygon(<Координаты>, <Перо>, <Кисть>)` — рисует многоугольник. В параметре <Координаты> указываются координаты трех и более точек. Указанные точки соединяются линиями. Кроме того, проводится прямая линия между первой и последней точками. Перо используется для рисования контура, а кисть для заливки. Пример:

```
>>> pen = aggdraw.Pen("#000000")
>>> brush = aggdraw.Brush("green")
>>> draw.polygon((50, 50, 150, 150, 50, 150), pen, brush)
>>> draw.polygon( (200, 200, 250, 200, 275, 250, 250, 300,
                   200, 300, 175, 250), None, brush)
>>> draw.flush()
<PIL.Image.Image instance at 0x011C6760>
>>> img.show()
```

- `ellipse(<Координаты>, <Перо>, <Кисть>)` — рисует окружность. В параметре <Координаты> указываются координаты прямоугольника, в который необходимо вписать окружность. Перо используется для рисования контура, а кисть для заливки. Пример:

```
>>> pen = aggdraw.Pen((0, 255, 255))
>>> brush = aggdraw.Brush((255, 255, 0))
>>> draw.ellipse((100, 100, 200, 200), None, brush)
>>> draw.ellipse((50, 170, 150, 300), pen)
>>> draw.flush()
<PIL.Image.Image instance at 0x011C6760>
>>> img.show()
```

- `arc()` — рисует дугу. Формат метода:

```
arc(<Координаты>, <Начальный угол>, <Конечный угол>, <Перо>)
```

В параметре `<Координаты>` указываются координаты прямоугольника, в который необходимо вписать окружность. Второй и третий параметры задают начальный и конечный углы, между которыми будет отображена дуга. Угол, равный 0, расположен в крайней правой точке. Линия рисуется против часовой стрелки от начального угла до конечного угла. Перо используется для рисования линии. Пример:

```
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = aggdraw.Draw(img)
>>> pen = aggdraw.Pen((255, 0, 0))
>>> draw.arc((10, 10, 290, 290), 180, 0, pen)
>>> draw.flush()
<PIL.Image.Image instance at 0x01511B70>
>>> img.show()
```

- `chord()` — рисует замкнутую дугу. Формат метода:

```
chord(<Координаты>, <Начальный угол>, <Конечный угол>,
      <Перо>, <Кисть>)
```

Метод `chord()` аналогичен методу `arc()`, но замыкает крайние точки дуги прямой линией. Перо используется для рисования контура, а кисть для заливки. Пример:

```
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = aggdraw.Draw(img)
>>> brush = aggdraw.Brush((255, 0, 0))
>>> draw.chord((10, 10, 290, 290), 180, 0, None, brush)
>>> brush = aggdraw.Brush((255, 255, 0))
>>> draw.chord((10, 10, 290, 290), -90, 0, None, brush)
>>> draw.flush()
<PIL.Image.Image instance at 0x01511DC8>
>>> img.show()
```

- `pieslice()` — рисует замкнутый сектор. Формат метода:

```
pieslice(<Координаты>, <Начальный угол>, <Конечный угол>,
         <Перо>, <Кисть>)
```

Метод `pieslice()` аналогичен методу `arc()`, но замыкает крайние точки дуги с центром окружности. Перо используется для рисования контура, а кисть для заливки. Пример:

```
>>> brush = aggdraw.Brush("green")
>>> draw.pieslice((10, 10, 290, 290), 45, 90, None, brush)
>>> draw.flush()
<PIL.Image.Image instance at 0x01511DC8>
>>> img.show()
```

19.7. Вывод текста на изображение

Вывести текст на изображение позволяет метод `text()` из модуля `ImageDraw`. Метод имеет следующий формат:

```
text(<Координаты>, <Строка>, fill=<Цвет>, font=<Объект шрифта>)
```

В первом параметре указывается кортеж из двух элементов, задающих координаты левого верхнего угла прямоугольной области, в которую вписан текст. Во втором параметре задается текст надписи. Параметр `fill` определяет цвет текста, а параметр `font` задает используемый шрифт. Для создания объекта шрифта предназначены следующие функции из модуля `ImageFont`:

- `load_default()` — шрифт по умолчанию. Вывести русские буквы таким шрифтом нельзя. Пример:

```
>>> from PIL import Image, ImageDraw, ImageFont
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = ImageDraw.Draw(img)
>>> font = ImageFont.load_default()
>>> draw.text((10, 10), "Hello", font=font, fill="red")
>>> img.show()
```

- `load(<Путь к файлу>)` — загружает шрифт из файла и возвращает объект шрифта. Если файл не найден, возбуждается исключение `IOError`. Файлы со шрифтами с расширением pil можно скачать с сайта **http://effbot.org/**. Пример:

```
>>> font = ImageFont.load("pilfonts/helvO12.pil")
>>> draw.text((10, 40), "Hello", font=font, fill="blue")
>>> img.show()
```

- `load_path(<Путь к файлу>)` — аналогичен методу `load()`, но дополнительно производит поиск файла в каталогах, указанных в `sys.path`. Если файл не найден, возбуждается исключение `IOError`;
- `truetype(<Путь к файлу>, <Размер>[, encoding])` — загружает файл с TrueType-шрифтом и возвращает объект шрифта. Если файл не найден, возбуждается исключение `IOError`. В Windows поиск файла дополнительно производится в стандартном каталоге со шрифтами. Пример вывода надписи на русском языке:

```
>>> txt = unicode("Привет мир", "cp1251")
>>> font_file = r"C:\WINDOWS\Fonts\arial.ttf"
>>> font = ImageFont.truetype(font_file, 24)
>>> draw.text((10, 80), txt, font=font, fill=(0, 0, 0))
>>> img.show()
```

Получить размеры прямоугольника, в который вписывается надпись, позволяет метод `textsize()`. Формат метода:

```
textsize(<Строка>, font=<Объект шрифта>)
```

Метод возвращает кортеж из двух элементов (<Ширина>, <Высота>). Кроме того, можно воспользоваться методом getsize(<Строка>) объекта шрифта.

Пример:

```
>>> txt = unicode("Привет мир", "cp1251")
>>> font_file = r"C:\WINDOWS\Fonts\arial.ttf"
>>> font = ImageFont.truetype(font_file, 24)
>>> draw.textsize(txt, font=font)
(133, 28)
>>> font.getsize(txt)
(133, 28)
```

Вывести текст на изображение позволяет также метод text() из модуля aggdraw. Формат метода:

```
text(<Координаты>, <Строка>, <Объект шрифта>)
```

Для создания объекта шрифта предназначена функция Font():

```
Font(<Цвет>, <Путь к файлу>[, size=12[, opacity=255]])
```

Выведем текст на русском языке с помощью модуля aggdraw (листинг 19.10).

Листинг 19.10. Вывод текста на русском языке с помощью модуля aggdraw

```
>>> import aggdraw
>>> from PIL import Image
>>> img = Image.new("RGB", (300, 300), (255, 255, 255))
>>> draw = aggdraw.Draw(img)
>>> font_file = r"C:\WINDOWS\Fonts\arial.ttf"
>>> font = aggdraw.Font("red", font_file, size=24)
>>> txt = unicode("Привет мир", "cp1251")
>>> draw.text((10, 10), txt, font)
>>> draw.flush()
<PIL.Image.Image instance at 0x015101E8>
>>> img.show()
```

Получить размеры прямоугольника, в который вписывается надпись, позволяет метод textsize(). Формат метода:

```
textsize(<Строка>, <Объект шрифта>)
```

Метод возвращает кортеж из двух элементов (<Ширина>, <Высота>). Пример:

```
>>> draw.textsize(txt, font)
(132.0, 28.0)
```

19.8. Создание скриншотов

Библиотека PIL в операционной системе Windows позволяет сделать снимок экрана (скриншот). Можно получить как полную копию экрана, так и копию опре-

деленной прямоугольной области. Для получения копии экрана предназначена функция `grab()` из модуля `ImageGrab`. Формат функции:

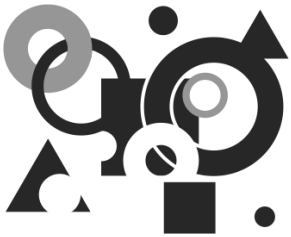
```
grab([<Координаты прямоугольной области>])
```

Если параметр не указан, то возвращается полная копия экрана в виде объекта изображения в режиме `RGB`. Для получения только определенной области необходимо указать координаты прямоугольника: левого верхнего угла и правого нижнего угла. Пример создания скриншотов приведен в листинге 19.11.

Листинг 19.11. Создание скриншотов

```
>>> from PIL import Image, ImageGrab
>>> img = ImageGrab.grab()
>>> img.save("screen.bmp", "BMP")
>>> img.mode
'RGB'
>>> img2 = ImageGrab.grab( (100, 100, 300, 300) )
>>> img2.save("screen2.bmp", "BMP")
>>> img2.size
(200, 200)
```

Глава 20



Взаимодействие с Интернетом

Интернет прочно вошел в нашу жизнь. Очень часто необходимо передать данные на Web-сервер или, наоборот, получить данные. Например, нужно получить котировки валют или прогноз погоды, проверить наличие писем в почтовом ящике и т. д. В состав стандартной библиотеки Python входит множество модулей, позволяющих работать практически со всеми протоколами Интернета. В этой главе мы рассмотрим только наиболее часто встречающиеся задачи: разбор URL-адреса и строки запроса на составляющие, преобразование гиперссылок, разбор HTML-эквивалентов, определение кодировки документа, а также обмен данными по протоколу HTTP с помощью модулей `httplib` и `urllib2`.

20.1. Разбор URL-адреса

С помощью модуля `urlparse` можно манипулировать URL-адресом. Например, разобрать его на составляющие или получить абсолютный URL-адрес, указав базовый адрес и относительный. URL-адрес состоит из следующих элементов:

```
<Протокол>://<Домен>:<Порт>/<Путь>;<Параметры>?<Запрос>#<Якорь>
```

Схема URL-адреса для протокола FTP выглядит по-другому:

```
<Протокол>://<Пользователь>:<Пароль>@<Домен>
```

Разобрать URL-адрес на составляющие позволяет функция `urlparse()`:

```
urlparse(<URL-адрес>[, <Схема>[, <Разбор якоря>]])
```

Функция возвращает объект `ParseResult` с результатами разбора URL-адреса. Получить значения можно с помощью атрибутов или индексов. Объект можно преобразовать в кортеж из следующих элементов: `(scheme, netloc, path, params, query, fragment)`. Элементы соответствуют схеме URL-адреса:

```
<scheme>://<netloc>/<path>;<params>?<query>#<fragment>
```

Обратите внимание на то, что название домена будет содержать номер порта. Кроме того, не ко всем атрибутам объекта можно получить доступ с помощью индексов. Результат разбора URL-адреса приведен в листинге 20.1.

Листинг 20.1. Разбор URL-адреса с помощью функции `urlparse()`

```
>>> from urlparse import urlparse
>>> url = urlparse("http://wwwadmin.ru:80/test.php;st?var=5#metka")
>>> url
ParseResult(scheme='http', netloc='wwwadmin.ru:80', path='/test.php',
params='st', query='var=5', fragment='metka')
>>> tuple(url) # Преобразование в кортеж
('http', 'wwwadmin.ru:80', '/test.php', 'st', 'var=5', 'metka')
```

Во втором параметре функции `urlparse()` можно указать название протокола, которое будет использоваться, если протокола нет в составе URL-адреса. По умолчанию используется пустая строка. Пример:

```
>>> urlparse("//wwwadmin.ru/test.php")
ParseResult(scheme='', netloc='wwwadmin.ru', path='/test.php',
params='', query='', fragment='')
>>> urlparse("//wwwadmin.ru/test.php", "http")
ParseResult(scheme='http', netloc='wwwadmin.ru', path='/test.php',
params='', query='', fragment='')
```

Объект `ParseResult`, возвращаемый функцией `urlparse()`, содержит следующие атрибуты:

- `scheme` — название протокола. Значение доступно также по индексу 0. По умолчанию пустая строка. Пример:

  ```
  >>> url.scheme, url[0]
  ('http', 'http')
  ```

- `netloc` — название домена вместе с номером порта. Значение доступно также по индексу 1. По умолчанию пустая строка. Пример:

  ```
  >>> url.netloc, url[1]
  ('wwwadmin.ru:80', 'wwwadmin.ru:80')
  ```

- `hostname` — название домена в нижнем регистре. Значение по умолчанию: `None`;
- `port` — номер порта. Значение по умолчанию: `None`. Пример:

  ```
  >>> url.hostname, url.port
  ('wwwadmin.ru', 80)
  ```

- `path` — путь. Значение доступно также по индексу 2. По умолчанию пустая строка. Пример:

  ```
  >>> url.path, url[2]
  ('/test.php', '/test.php')
  ```

- `params` — параметры. Значение доступно также по индексу 3. По умолчанию пустая строка. Пример:

```
>>> url.params, url[3]
('st', 'st')
```

- `query` — строка запроса. Значение доступно также по индексу 4. По умолчанию пустая строка. Пример:

```
>>> url.query, url[4]
('var=5', 'var=5')
```

- `fragment` — якорь. Значение доступно также по индексу 5. По умолчанию пустая строка. Пример:

```
>>> url.fragment, url[5]
('metka', 'metka')
```

  Если третий параметр в функции `urlparse()` имеет значение `False`, то якорь будет входить в состав значения других атрибутов, а не `fragment`. По умолчанию параметр имеет значение `True`. Пример:

```
>>> u = urlparse("http://site.ru/add.php?v=5#metka")
>>> u.query, u.fragment
('v=5', 'metka')
>>> u = urlparse("http://site.ru/add.php?v=5#metka","",False)
>>> u.query, u.fragment
('v=5#metka', '')
```

- `username` — имя пользователя. Значение по умолчанию: `None`;
- `password` — пароль. Значение по умолчанию: `None`. Пример:

```
>>> ftp = urlparse("ftp://user:123456@mysite.ru")
>>> ftp.scheme, ftp.hostname, ftp.username, ftp.password
('ftp', 'mysite.ru', 'user', '123456')
```

- `geturl()` — метод возвращает URL-адрес. Пример:

```
>>> url.geturl()
'http://wwwadmin.ru:80/test.php;st?var=5#metka'
```

Произвести обратную операцию (собрать URL-адрес из отдельных значений) позволяет функция `urlunparse(<Последовательность>)` (листинг 20.2).

Листинг 20.2. Использование функции `urlunparse()`

```
>>> import urlparse
>>> t = ('http', 'wwwadmin.ru:80', '/test.php', '', 'var=5', 'metka')
>>> urlparse.urlunparse(t)
'http://wwwadmin.ru:80/test.php?var=5#metka'
>>> l = ['http', 'wwwadmin.ru:80', '/test.php', '', 'var=5', 'metka']
>>> urlparse.urlunparse(l)
'http://wwwadmin.ru:80/test.php?var=5#metka'
```

Вместо функции `urlparse()` можно воспользоваться функцией `urlsplit(<URL-адрес>[, <Схема>[, <Разбор якоря>]])`. Функция возвращает объект `SplitResult` с результатами разбора URL-адреса. Объект можно преобразовать в кортеж из следующих элементов: `(scheme, netloc, path, query, fragment)`. Обратиться к значениям можно как по индексу, так и названию атрибутов. Пример использования функции `urlsplit()` приведен в листинге 20.3.

Листинг 20.3. Разбор URL-адреса с помощью функции `urlsplit()`

```
>>> from urlparse import urlsplit
>>> url = urlsplit("http://wwwadmin.ru:80/test.php;st?var=5#metka")
>>> url
SplitResult(scheme='http', netloc='wwwadmin.ru:80',
path='/test.php;st', query='var=5', fragment='metka')
>>> url[0], url[1], url[2], url[3], url[4]
('http', 'wwwadmin.ru:80', '/test.php;st', 'var=5', 'metka')
>>> url.scheme, url.netloc, url.hostname, url.port
('http', 'wwwadmin.ru:80', 'wwwadmin.ru', 80)
>>> url.path, url.query, url.fragment
('/test.php;st', 'var=5', 'metka')
>>> ftp = urlsplit("ftp://user:123456@mysite.ru")
>>> ftp.scheme, ftp.hostname, ftp.username, ftp.password
('ftp', 'mysite.ru', 'user', '123456')
```

Выполнить обратную операцию (собрать URL-адрес из отдельных значений) позволяет функция `urlunsplit(<Последовательность>)` (листинг 20.4).

Листинг 20.4. Использование функции `urlunsplit()`

```
>>> from urlparse import urlunsplit
>>> t = ('http', 'wwwadmin.ru:80', '/test.php;st', 'var=5', 'metka')
>>> urlunsplit(t)
'http://wwwadmin.ru:80/test.php;st?var=5#metka'
```

20.2. Кодирование и декодирование строки запроса

В предыдущем разделе мы научились разбирать URL-адрес на составляющие. Обратите внимание на то, что значение параметра `<Запрос>` возвращается в виде строки. Строка запроса является составной конструкцией, содержащей пары `параметр=значение`. Все специальные символы внутри названия параметра и значения кодируются последовательностями `%nn`. Например, для параметра `str`, имеющего значение "Строка" в кодировке Windows-1251, строка запроса будет выглядеть так:

```
str=%D1%F2%F0%EE%EA%E0
```

Если строка запроса содержит несколько пар `параметр=значение`, то они разделяются символом `&`. Добавим параметр `v` со значением 10:

```
str=%D1%F2%F0%EE%EA%E0&v=10
```

В строке запроса может быть несколько параметров с одним названием, но разными значениями. Например, если передаются значения нескольких выбранных переключателей, объединенных в группу:

```
str=%D1%F2%F0%EE%EA%E0&v=10&v=20
```

В версиях Python 2.6 и более поздних разобрать строку запроса на составляющие и декодировать данные позволяют две функции из модуля `urlparse`:

- `parse_qs()` — разбирает строку запроса и возвращает словарь с ключами, содержащими названия параметров, и списком значений. Формат функции:

  ```
  parse_qs(<Строка запроса>[, <Обработка пустых значений>
                           [, <Обработка ошибок>]])
  ```

  Если во втором параметре указано значение `True`, то параметры, не имеющие значений внутри строки запроса, также будут добавлены в результат. По умолчанию пустые параметры игнорируются. Если в третьем параметре указано значение `True`, то при наличии ошибки возбуждается исключение `ValueError`. По умолчанию ошибки игнорируются. Пример разбора строки запроса:

  ```
  >>> import urlparse
  >>> s = "str=%D1%F2%F0%EE%EA%E0&v=10&v=20&t="
  >>> urlparse.parse_qs(s)
  {'str': ['\xd1\xf2\xf0\xee\xea\xe0'], 'v': ['10', '20']}
  >>> urlparse.parse_qs(s, True)
  {'t': [''], 'str': ['\xd1\xf2\xf0\xee\xea\xe0'],
   'v': ['10', '20']}
  ```

- `parse_qsl()` — функция аналогична `parse_qs()`, но возвращает не словарь, а список кортежей из двух элементов. Первый элемент кортежа содержит название параметра, а второй элемент его значение. Если строка запроса содержит несколько параметров с одинаковым значением, то они будут расположены в разных кортежах. Формат функции:

  ```
  parse_qsl(<Строка запроса>[, <Обработка пустых значений>
                            [, <Обработка ошибок>]])
  ```

  Пример разбора строки запроса:

  ```
  >>> params = urlparse.parse_qsl(s)
  >>> params
  [('str', '\xd1\xf2\xf0\xee\xea\xe0'), ('v', '10'),
  ('v', '20')]
  >>> print params[0][1]
  Строка
  >>> urlparse.parse_qsl(s, True)
  [('str', '\xd1\xf2\xf0\xee\xea\xe0'), ('v', '10'),
   ('v', '20'), ('t', '')]
  ```

В версиях Python 2.5 и более ранних разобрать строку запроса на составляющие и декодировать данные позволяют функции `parse_qs()` и `parse_qsl()` из модуля `cgi`. Формат функций полностью аналогичен одноименным функциям из модуля `urlparse`. Пример разбора строки запроса приведен в листинге 20.5.

Листинг 20.5. Пример разбора строки запроса

```
>>> import cgi
>>> s = "str=%D1%F2%F0%EE%EA%E0&v=10&v=20&t="
>>> cgi.parse_qs(s, True)
{'t': [''], 'str': ['\xd1\xf2\xf0\xee\xea\xe0'], 'v': ['10', '20']}
>>> cgi.parse_qsl(s, True)
[('str', '\xd1\xf2\xf0\xee\xea\xe0'), ('v', '10'), ('v', '20'),
 ('t', '')]
```

Выполнить обратную операцию, преобразовать отдельные составляющие в строку запроса позволяет функция `urlencode(<Объект>[, <Флаг>])` из модуля `urllib`. В качестве параметра можно указать словарь с данными (или последовательность), каждый элемент которого содержит кортеж из двух элементов. Первый элемент кортежа становится параметром, а второй элемент его значением. Параметры и значения автоматически обрабатываются с помощью функции `quote_plus()` из модуля `urllib`. В случае указания последовательности параметры внутри строки будут идти в том же порядке, что и внутри последовательности. Пример указания словаря и последовательности приведен в листинге 20.6.

Листинг 20.6. Функция `urlencode()`

```
>>> import urllib
>>> params = {"str": "Строка 2", "var": 20}       # Словарь
>>> urllib.urlencode(params)
'var=20&str=%D1%F2%F0%EE%EA%E0+2'
>>> params = [ ("str", "Строка 2"), ("var", 20) ] # Список
>>> urllib.urlencode(params)
'str=%D1%F2%F0%EE%EA%E0+2&var=20'
```

Если необязательный параметр `<Флаг>` в функции `urlencode()` имеет значение `True`, то можно указать последовательность из нескольких значений во втором параметре кортежа. В этом случае в строку запроса добавляются несколько параметров со значениями из этой последовательности. Значение параметра `<Флаг>` по умолчанию — `False`. В качестве примера укажем список из двух элементов (листинг 20.7).

Листинг 20.7. Составление строки запроса из элементов последовательности

```
>>> params = [ ("var", [10, 20]) ]
```

```
>>> urllib.urlencode(params, False)
'var=%5B10%2C+20%5D'
>>> urllib.urlencode(params, True)
'var=10&var=20'
```

Последовательность можно также указать в качестве значения в словаре:

```
>>> params = { "var": [10, 20] }
>>> urllib.urlencode(params, True)
'var=10&var=20'
```

Выполнить кодирование и декодирование отдельных элементов строки запроса позволяют следующие функции из модуля `urllib`:

- `quote(<Строка>[, <Символы>])` — заменяет все специальные символы последовательностями `%nn`. Цифры, английские буквы и символы подчеркивания (_), точки (.) и дефиса (-) не кодируются. Пробелы преобразуются в последовательность `%20`. Возвращаемое значение для русских букв зависит от кодировки исходной строки. Во втором параметре можно указать символы, которые преобразовывать нельзя. По умолчанию параметр имеет значение `/`. Пример:

```
>>> import urllib
>>> urllib.quote("Строка") # Кодировка windows-1251
'%D1%F2%F0%EE%EA%E0'
>>> urllib.quote(unicode("Строка", "cp1251").encode("utf-8"))
'%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0'
>>> urllib.quote("/~nik/"), urllib.quote("/~nik/", "")
('/%7Enik/', '%2F%7Enik%2F')
>>> urllib.quote("/~nik/", "/~")
'/~nik/'
```

- `quote_plus(<Строка>[, <Символы>])` — функция аналогична `quote()`, но пробелы заменяются на `+`, а не преобразуются в последовательность `%20`. Кроме того, по умолчанию символ `/` преобразуется в последовательность `%2F`. Пример:

```
>>> urllib.quote("Строка 2")
'%D1%F2%F0%EE%EA%E0%202'
>>> urllib.quote_plus("Строка 2")
'%D1%F2%F0%EE%EA%E0+2'
>>> urllib.quote_plus("/~nik/")
'%2F%7Enik%2F'
>>> urllib.quote_plus("/~nik/", "/~")
'/~nik/'
```

- `unquote(<Строка>)` — заменяет последовательности `%nn` на соответствующие символы. Символ `+` пробелом не заменяется. Пример:

```
>>> print urllib.unquote("%D1%F2%F0%EE%EA%E0")
Строка
>>> s = '%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0'
>>> print urllib.unquote(s).decode("utf-8").encode("cp1251")
Строка
>>> print urllib.unquote('%D1%F2%F0%EE%EA%E0+2')
```

```
Строка+2
```

- `unquote_plus(<Строка>)` — функция аналогична `unquote()`, но дополнительно заменяет символ + пробелом. Пример:

```
>>> print urllib.unquote_plus('%D1%F2%F0%EE%EA%E0+2')
Строка 2
>>> print urllib.unquote_plus('%D1%F2%F0%EE%EA%E0%202')
Строка 2
```

20.3. Преобразование относительной ссылки в абсолютную

Очень часто в HTML-документах указываются не абсолютные ссылки, а относительные. При относительном URL-адресе путь определяется с учетом местоположения Web-страницы, на которой находится ссылка, или значения параметра `href` тега `<base>`. Преобразовать относительную ссылку в абсолютный URL-адрес позволяет функция `urljoin()` из модуля `urlparse`. Формат функции:

```
urljoin(<Базовый URL-адрес>, <Относительный или абсолютный URL-адрес>
        [, <Разбор якоря>])
```

В качестве примера рассмотрим преобразование различных относительных ссылок (листинг 20.8).

Листинг 20.8. Варианты преобразования относительных ссылок

```
>>> from urlparse import urljoin
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', 'file.html')
'http://wwwadmin.ru/f1/f2/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', 'f3/file.html')
'http://wwwadmin.ru/f1/f2/f3/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '/file.html')
'http://wwwadmin.ru/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', './file.html')
'http://wwwadmin.ru/f1/f2/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '../file.html')
'http://wwwadmin.ru/f1/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '../../file.html')
'http://wwwadmin.ru/file.html'
>>> urljoin('http://wwwadmin.ru/f1/f2/test.html', '../../../file.html')
'http://wwwadmin.ru/../file.html'
```

В последнем случае мы специально указали уровень относительности больше, чем нужно. Как видно из результата, в данном случае возникает ошибка.

20.4. Разбор HTML-эквивалентов

В HTML-документе некоторые символы являются специальными. Например, знак "меньше" (<) и знак "больше" (>), кавычки и др. Для отображения специальных символов используются HTML-эквиваленты. Например, знак "меньше" заменяется последовательностью `<`, а знак больше — `>`. Манипулировать HTML-эквивалентами позволяют следующие функции из модуля `xml.sax.saxutils`:

- `escape(<Строка>[, <Словарь>])` — заменяет символы <, > и & соответствующими HTML-эквивалентами. Необязательный параметр `<Словарь>` позволяет указать словарь с дополнительными символами в качестве ключей и их HTML-эквивалентами в качестве значений. Пример:

```
>>> from xml.sax.saxutils import escape
>>> s = """&<>" """
>>> escape(s)
'&amp;&lt;&gt;" '
>>> escape(s, { '"': "&quot;", " ": " " } )
'&amp;&lt;&gt;&quot; '
```

- `quoteattr(<Строка>[, <Словарь>])` — функция аналогична `escape()`, но дополнительно заключает строку в кавычки или апострофы. Если внутри строки встречаются только двойные кавычки, то строка заключается в апострофы. Если внутри строки встречаются и кавычки, и апострофы, то двойные кавычки заменяются HTML-эквивалентом, а строка заключается в двойные кавычки. Если кавычки и апострофы не входят в строку, то строка заключается в двойные кавычки. Пример:

```
>>> from xml.sax.saxutils import quoteattr
>>> print quoteattr("""&<>" """)
'&amp;&lt;&gt;" '
>>> print quoteattr("""&<>"'""")
"&amp;&lt;&gt;&quot;'"
>>> print quoteattr("""&<>" """, { '"': "&quot;" } )
"&amp;&lt;&gt;&quot; "
```

- `unescape(<Строка>[, <Словарь>])` — заменяет HTML-эквиваленты `&`, `<` и `>` обычными символами. Необязательный параметр `<Словарь>` позволяет указать словарь с дополнительными HTML-эквивалентами в качестве ключей и обычными символами в качестве значений. Пример:

```
>>> from xml.sax.saxutils import unescape
>>> s = '&amp;&lt;&gt;&quot; '
>>> unescape(s)
'&<>&quot; '
>>> unescape(s, { "&quot;": '"', " ": " " } )
'&<>" '
```

Для замены символов <, > и & HTML-эквивалентами можно также воспользоваться функцией `escape(<Строка>[, <Флаг>])` из модуля `cgi`. Если во втором па-

раметре указано значение `True`, то двойные кавычки также будут заменяться HTML-эквивалентом (листинг 20.9).

Листинг 20.9. Замена спецсимволов HTML-эквивалентами

```
>>> import cgi
>>> cgi.escape("""&<>"' """)
'&amp;&lt;&gt;"\' '
>>> cgi.escape("""&<>"' """, True)
"&amp;&lt;&gt;&quot;' "
```

20.5. Обмен данными по протоколу HTTP

Модуль `httplib` позволяет получить информацию из Интернета по протоколам HTTP и HTTPS. Отправить запрос можно методами `GET`, `POST` и `HEAD`. Для создания объекта соединения, использующего протокол HTTP, предназначен класс `HTTPConnection`. Конструктор класса имеет следующий формат:

```
HTTPConnection(<Домен>[, <Порт>[, strict[,timeout]]])
```

В первом параметре указывается название домена без протокола. Во втором параметре задается номер порта. Если порт не указан, то используется порт 80. Номер порта можно также задать после названия домена через двоеточие. Пример создания объекта соединения:

```
>>> import httplib
>>> con = httplib.HTTPConnection("wwwadmin.ru")
>>> con2 = httplib.HTTPConnection("wwwadmin.ru", 80)
>>> con3 = httplib.HTTPConnection("wwwadmin.ru:80")
```

После создания объекта соединения необходимо отправить параметры запроса с помощью метода `request()`. Формат метода:

```
request(<Метод>, <Путь>[, <Данные>[, headers=<Заголовки>]])
```

В первом параметре указывается метод передачи данных (`GET`, `POST` или `HEAD`). Второй параметр задает путь от корня сайта. Если для передачи данных используется метод `GET`, то после вопросительного знака можно указать передаваемые данные. В необязательном третьем параметре задаются данные, которые передаются методом `POST`. Допустимо указать строку или файловый объект (начиная с Python 2.6). Четвертый параметр задает HTTP-заголовки, отправляемые на сервер. Заголовки указываются в виде словаря.

Получить объект результата запроса позволяет метод `getresponse()`. Прочитать ответ сервера (без заголовков) можно с помощью метода `read([<Количество байт>])`. Если параметр не указан, то метод `read()` возвращает все данные, а при наличии значения — только указанное количество байтов при каждом вызове. Если данных больше нет, метод возвращает пустую строку. Прежде чем выполнять другой запрос, данные должны быть получены полностью. Закрыть объект соединения

позволяет метод `close()`. В качестве примера отправим запрос методом `GET` и прочитаем результат (листинг 20.10).

Листинг 20.10. Отправка данных методом `GET`

```
>>> import urllib, httplib
>>> data = urllib.urlencode({"color": "Красный", "var": 15})
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Referer": "/index.php" }
>>> con = httplib.HTTPConnection("wwwadmin.ru")
>>> con.request("GET", "/testrobots.php?%s" % data, headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print result.read()         # Читаем данные полностью
... Фрагмент опущен ...
>>> con.close()                 # Закрываем объект соединения
```

Теперь отправим данные методом `POST`. В этом случае в первом параметре метода `request()` задается значение "POST", а данные передаются через третий параметр. Размер строки запроса автоматически указывается в заголовке `Content-Length`. Пример отправки данных методом `POST` приведен в листинге 20.11.

Листинг 20.11. Отправка данных методом `POST`

```
>>> import urllib, httplib
>>> data = urllib.urlencode({"color": "Красный", "var": 15})
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Content-Type": "application/x-www-form-urlencoded",
                "Referer": "/index.php" }
>>> con = httplib.HTTPConnection("wwwadmin.ru")
>>> con.request("POST", "/testrobots.php", data, headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print result.read()         # Читаем данные полностью
... Фрагмент опущен ...
>>> con.close()
```

Обратите внимание на заголовок `Content-Type`. Если в этом заголовке указано значение `application/x-www-form-urlencoded`, то это означает, что отправлены данные формы. При наличии этого заголовка некоторые языки программирования автоматически производят разбор строки запроса. Например, в PHP переданные

данные будут доступны через глобальный массив $_POST. Если заголовок не указать, то данные через массив $_POST доступны не будут.

Объект результата предоставляет следующие методы и атрибуты:

- getheader(<Заголовок>[, <Значение по умолчанию>]) — возвращает значение указанного заголовка. Если заголовок не найден, возвращается значение None или значение из второго параметра. Пример:

```
>>> result.getheader("Content-Type")
'text/plain; charset=windows-1251'
>>> print result.getheader("Content-Types")
None
>>> result.getheader("Content-Types", 10)
10
```

- getheaders() — возвращает все заголовки ответа сервера в виде списка кортежей. Каждый кортеж состоит из двух элементов (<Заголовок>, <Значение>). Пример получения заголовков ответа сервера:

```
>>> result.getheaders()
[('transfer-encoding', 'chunked'), ('keep-alive', 'timeout=20'),
('server', 'nginx/0.8.29'), ('connection', 'keep-alive'),
('date', 'Tue, 08 Jun 2010 19:16:40 GMT'),
('content-type', 'text/plain; charset=windows-1251')]
С помощью функции dict() такой список можно преобразовать в словарь:
>>> dict(result.getheaders())
{'transfer-encoding': 'chunked', 'keep-alive': 'timeout=20',
'server': 'nginx/0.8.29', 'connection': 'keep-alive',
'date': 'Tue, 08 Jun 2010 19:16:40 GMT',
'content-type': 'text/plain; charset=windows-1251'}
```

- status — код возврата в виде числа. Успешными считаются коды от 200 до 299 и код 304, означающий, что документ не был изменен со времени последнего посещения. Коды 301 и 302 задают перенаправление. Код 401 означает необходимость авторизации, 403 — доступ закрыт, 404 — документ не найден, а код 500 и коды выше информируют об ошибке сервера. Пример:

```
>>> result.status
200
```

- reason — текстовый статус возврата. Пример:

```
>>> result.reason           # При коде 200
'OK'
>>> result.reason           # При коде 302
'Moved Temporarily'
```

- version — версия протокола в виде числа. Число 10 для протокола HTTP/1.0 и число 11 для протокола HTTP/1.1. Пример:

```
>>> result.version          # Протокол HTTP/1.1
11
```

- `msg` — объект `mimetools.Message`. С его помощью можно получить дополнительную информацию о заголовках ответа сервера.

  Рассмотрим основные методы и атрибуты объекта `mimetools.Message`:
- `headers` — список всех заголовков ответа сервера. Пример:

```
>>> result.msg.headers
['Server: nginx/0.8.29\r\n', 'Date: Tue, 08 Jun 2010 19:16:40
GMT\r\n', 'Content-Type: text/plain; charset=windows-1251\r\n',
'Transfer-Encoding: chunked\r\n', 'Connection: keep-alive\r\n',
'Keep-Alive: timeout=20\r\n\r\n']
```

- `getrawheader(<Заголовок>)` — возвращает значение указанного заголовка в виде необработанной строки или значение `None`. Пример:

```
>>> result.msg.getrawheader("Server")
' nginx/0.8.29\r\n'
```

- `getheaders(<Заголовок>)` — возвращает список значений указанного заголовка или пустой список. Пример:

```
>>> result.msg.getheaders("Server")
['nginx/0.8.29']
```

- `getheader(<Заголовок>[, <Значение по умолчанию>])` — возвращает значение указанного заголовка в виде строки. Если заголовок не найден, возвращается значение `None` или значение из второго параметра. Пример:

```
>>> result.msg.getheader("Server", "")
'nginx/0.8.29'
```

- `gettype()` — возвращает MIME-тип документа из заголовка `Content-Type`:

```
>>> result.msg.gettype()
'text/plain'
```

- `getmaintype()` — возвращает первую составляющую MIME-типа:

```
>>> result.msg.getmaintype()
'text'
```

- `getsubtype()` — возвращает вторую составляющую MIME-типа:

```
>>> result.msg.getsubtype()
'plain'
```

- `getplist()` — позволяет получить параметры из заголовка `Content-Type`:

```
>>> result.msg.getplist()
['charset=windows-1251']
```

- `getparam(<Параметр>)` — позволяет получить значение указанного параметра из заголовка `Content-Type`. Получим кодировку документа:

```
>>> result.msg.getparam("charset")
'windows-1251'
```

- `getdate(<Заголовок>)` — возвращает значение указанного заголовка в виде кортежа из элементов даты и времени. Пример:

```
>>> result.msg.getdate("Date")
(2010, 6, 8, 19, 16, 40, 0, 1, 0)
```

В качестве примера отправим запрос методом HEAD и выведем заголовки ответа сервера (листинг 20.12).

Листинг 20.12. Отправка запроса методом HEAD

```
>>> import httplib
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Referer": "/index.php" }
>>> con = httplib.HTTPConnection("wwwadmin.ru")
>>> con.request("HEAD", "/", headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> for line in result.msg.headers: print line.rstrip()

Server: nginx/0.8.29
Date: Tue, 08 Jun 2010 19:26:44 GMT
Content-Type: text/html
Connection: keep-alive
Keep-Alive: timeout=20
>>> print result.read() # Данные не передаются, только заголовки!

>>> con.close()
```

Перечислим основные HTTP-заголовки и их предназначение:

- GET — заголовок запроса при передаче данных методом GET;
- POST — заголовок запроса при передаче данных методом POST;
- Host — название домена;
- Accept — MIME-типы, поддерживаемые Web-браузером;
- Accept-Language — список поддерживаемых языков в порядке предпочтения;
- Accept-Charset — список поддерживаемых кодировок;
- Accept-Encoding — список поддерживаемых методов сжатия;
- Content-Type — тип передаваемых данных;
- Content-Length — длина передаваемых данных при методе POST;
- Cookie — информация об установленных cookies;
- Last-Modified — дата последней модификации файла;
- Location — перенаправление на другой URL-адрес;
- Pragma — заголовок, запрещающий кэширование документа в протоколе HTTP/1.0;
- Cache-Control — заголовок, управляющий кэшированием документа в протоколе HTTP/1.1;
- Referer — URL-адрес, с которого пользователь перешел на наш сайт;
- Server — название и версия программного обеспечения сервера;
- User-Agent — информация об используемом Web-браузере.

Получить полное описание заголовков можно в спецификации RFC 2616, расположенной по адресу **http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html**. Чтобы "подсмотреть" заголовки, отправляемые Web-браузером и сервером, можно воспользоваться модулем Firebug для Firefox. Для этого на вкладке **Сеть** следует щелкнуть мышью на строке запроса. Кроме того, можно установить панель ieHTTPHeaders в Web-браузере Internet Explorer.

20.6. Обмен данными с помощью модуля *urllib2*

Модуль `urllib2` предоставляет расширенные возможности для получения информации из Интернета. Поддерживаются автоматические перенаправления при получении заголовка `Location`, возможность аутентификации, обработка cookies и др.

Для выполнения запроса предназначена функция `urlopen()`. Формат функции:

```
urlopen(<URL-адрес или объект запроса>[, <Данные>][, <Тайм-аут>])
```

В первом параметре задается полный URL-адрес или объект, возвращаемый конструктором класса `Request`. Запрос выполняется методом `GET`, если данные не указаны во втором параметре, и методом `POST` в противном случае. При передаче данных методом `POST` автоматически добавляется заголовок `Content-Type` со значением `application/x-www-form-urlencoded`. Третий параметр задает максимальное время выполнения запроса в секундах. Параметр доступен, начиная с версии 2.6. Объект, возвращаемый функцией `urlopen()`, содержит следующие методы и атрибуты:

- `read([<Количество байтов>])` — считывает данные. Если параметр не указан, то возвращается содержимое результата от текущей позиции указателя до конца. Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество байтов. Когда достигается конец, метод возвращает пустую строку. Пример:

```
>>> import urllib2
>>> res = urllib2.urlopen("http://wwwadmin.ru/testrobots.php")
>>> print res.read(34)
Название робота: Python-urllib/2.6
>>> print res.read()
... Фрагмент опущен ...
>>> res.read()
''
```

- `readline([<Количество байтов>])` — считывает одну строку при каждом вызове. При достижении конца возвращается пустая строка. Если в необязательном параметре указано число, то считывание будет выполняться до тех пор, пока не встретится символ новой строки (`\n`), символ конца или не будет

прочитано указанное количество байтов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество байтов. Если количество символов в строке больше, то возвращается указанное количество байтов. Пример:

```
>>> res = urllib2.urlopen("http://wwwadmin.ru/testrobots.php")
>>> print res.readline()
Название робота: Python-urllib/2.6
```

- `readlines([<Количество байтов>])` — считывает весь результат в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Если задан параметр, то считывается указанное количество байтов плюс фрагмент до конца строки. При достижении конца возвращается пустой список. Пример:

```
>>> res = urllib2.urlopen("http://wwwadmin.ru/testrobots.php")
>>> res.readlines(3)
['\xcd\xe0\xe7\xe2\xe0\xed\xe8\xe5 \xf0\xee\xe1\xee\xf2\xe0:
Python-urllib/2.6\n']
>>> res.readlines()
... Фрагмент опущен ...
>>> res.readlines()
[]
```

- `next()` — считывает одну строку при каждом вызове. При достижении конца результата возбуждается исключение `StopIteration`. Благодаря методу `next()` можно перебирать результат построчно с помощью цикла `for`. Цикл `for` на каждой итерации будет автоматически вызывать метод `next()`. Пример:

```
>>> res = urllib2.urlopen("http://wwwadmin.ru/testrobots.php")
>>> for line in res: print line,
```

- `close()` — закрывает объект результата;
- `geturl()` — возвращает URL-адрес полученного документа. Так как все перенаправления автоматически обрабатываются, URL-адрес полученного документа может не совпадать с URL-адресом, заданным первоначально;
- `info()` — возвращает объект, с помощью которого можно получить информацию о заголовках ответа сервера. Основные методы и атрибуты этого объекта мы рассматривали при изучении модуля `httplib` (см. значение атрибута `msg` объекта результата). Получить ссылку на тот же объект можно с помощью атрибута `headers`. Пример:

```
>>> res = urllib2.urlopen("http://wwwadmin.ru/")
>>> info = res.info()
>>> info.headers
['Server: nginx/0.8.29\r\n', 'Date: Tue, 08 Jun 2010 19:38:15
GMT\r\n', 'Content-Type: text/html\r\n',
'Transfer-Encoding: chunked\r\n', 'Connection: close\r\n']
>>> info.getrawheader("Server")
' nginx/0.8.29\r\n'
>>> info.getheaders("Server")
```

```
['nginx/0.8.29']
>>> info.getheader("Server", "")
'nginx/0.8.29'
>>> info.gettype(), info.getmaintype(), info.getsubtype()
('text/html', 'text', 'html')
```

- `code` — содержит код возврата в виде числа;
- `msg` — содержит текстовый статус возврата. Пример:

```
>>> res.code, res.msg
(200, 'OK')
```

В качестве примера выполним запросы методами `GET` и `POST` (листинг 20.13).

Листинг 20.13. Отправка данных методами `GET` и `POST`

```
>>> import urllib, urllib2
>>> data = urllib.urlencode({"color": "Красный", "var": 15})
>>> # Отправка данных методом GET
>>> url = "http://wwwadmin.ru/testrobots.php?%s" % data
>>> res = urllib2.urlopen(url)
>>> print res.read()[:34]
Название робота: Python-urllib/2.6
>>> res.close()
>>> # Отправка данных методом POST
>>> url = "http://wwwadmin.ru/testrobots.php"
>>> res = urllib2.urlopen(url, data)
>>> print res.read()
... Фрагмент опущен ...
>>> res.close()
```

Как видно из результата, по умолчанию название робота — "Python-urllib/<Версия Python>". Если необходимо задать свое название робота и передать дополнительные заголовки, то следует создать экземпляр класса `Request` и передать его в функцию `urlopen()` вместо URL-адреса. Конструктор класса `Request` имеет следующий формат:

```
Request(<URL-адрес>[, <Данные>][, headers=<Заголовки>]
                   [, origin_req_host][, unverifiable])
```

В первом параметре указывается URL-адрес. Запрос выполняется методом `GET`, если данные не указаны во втором параметре, и методом `POST` в противном случае. При передаче данных методом `POST` автоматически добавляется заголовок `Content-Type` со значением `application/x-www-form-urlencoded`. Третий параметр задает заголовки запроса в виде словаря. Четвертый и пятый параметр используются для обработки cookies. За дополнительной информацией по этим параметрам обращайтесь к документации. В качестве примера выполним запросы методами `GET` и `POST` (листинг 20.14).

Листинг 20.14. Использование класса `Request`

```
>>> import urllib, urllib2
>>> headers = { "User-Agent": "MySpider/1.0",
                "Accept": "text/html, text/plain, application/xml",
                "Accept-Language": "ru, ru-RU",
                "Accept-Charset": "windows-1251",
                "Referer": "/index.php" }
>>> data = urllib.urlencode({"color": "Красный", "var": 15})
>>> # Отправка данных методом GET
>>> url = "http://wwwadmin.ru/testrobots.php?%s" % data
>>> request = urllib2.Request(url, headers=headers)
>>> res = urllib2.urlopen(request)
>>> print res.read()[:29]
Название робота: MySpider/1.0
>>> res.close()
>>> # Отправка данных методом POST
>>> url = "http://wwwadmin.ru/testrobots.php"
>>> request = urllib2.Request(url, data, headers=headers)
>>> res = urllib2.urlopen(request)
>>> print res.read()
... Фрагмент опущен ...
>>> res.close()
```

Как видно из результата, название нашего робота теперь "MySpider/1.0".

20.7. Определение кодировки

Документы в Интернете могут быть в различных кодировках. Чтобы документ был правильно обработан, необходимо знать его кодировку. Определить кодировку можно по заголовку `Content-Type` в заголовках ответа сервера:

```
Content-Type: text/html; charset=utf-8
```

Кодировку HTML-документа можно также определить по значению параметра `content` тега `<meta>`, расположенного в разделе `HEAD`:

```
<meta http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
```

Очень часто встречается ситуация, когда кодировка в ответе сервера не совпадает с кодировкой, указанной в теге `<meta>`, или кодировка вообще не указана. Определить кодировку документа в этом случае позволяет библиотека `chardet`. Для установки библиотеки со страницы **http://chardet.feedparser.org/download/** скачиваем архив python2-chardet-2.0.1.tgz, а затем распаковываем его в текущую папку, например, с помощью архиватора WinRAR. Запускаем командную строку и пере-

ходим в папку с библиотекой (в моем случае библиотека разархивирована в папку C:\book), выполнив команду:

```
cd C:\book\python2-chardet-2.0.1
```

Затем запускаем программу установки, выполнив команду:

```
C:\Python26\python.exe setup.py install
```

Для проверки установки запускаем редактор IDLE и в окне **Python Shell** выполняем следующий код:

```
>>> import chardet
>>> chardet.__version__
'2.0.1'
```

Определить кодировку строки позволяет функция `detect(<Строка>)`. В качестве значения функция возвращает словарь с двумя элементами. Ключ `encoding` содержит название кодировки, а ключ `confidence` — коэффициент точности определения (вещественное число от 0 до 1). Пример определения кодировки приведен в листинге 20.15.

Листинг 20.15. Пример определения кодировки

```
>>> import chardet
>>> chardet.detect("Строка")
{'confidence': 0.98999999999999999, 'encoding': 'windows-1251'}
>>> chardet.detect(unicode("Строка", "cp1251").encode("koi8-r"))
{'confidence': 0.98999999999999999, 'encoding': 'KOI8-R'}
>>> chardet.detect(unicode("Строка", "cp1251").encode("utf-8"))
{'confidence': 0.98999999999999999, 'encoding': 'utf-8'}
>>> chardet.detect(unicode("Строка", "cp1251").encode("utf-16"))
{'confidence': 1.0, 'encoding': 'UTF-16LE'}
```

Если файл имеет большой размер, то вместо считывания всего файла в строку и использования функции `detect()` можно воспользоваться классом `UniversalDetector`. В этом случае можно читать файл построчно и передавать текущую строку методу `feed()`. Если определение кодировки прошло успешно, атрибут `done` будет иметь значение `True`. Это условие можно использовать для выхода из цикла. После окончания проверки следует вызвать метод `close()`. Получить результат определения кодировки позволяет атрибут `result`. Очистить результат и подготовить объект к дальнейшему определению кодировки можно с помощью метода `reset()`. Пример использования класса `UniversalDetector` приведен в листинге 20.16.

Листинг 20.16. Пример использования класса `UniversalDetector`

```
# -*- coding: cp1251 -*-
from chardet.universaldetector import UniversalDetector
ud = UniversalDetector()       # Создаем объект
```

```
for line in open("file.txt"):
    ud.feed(line)               # Передаем текущую строку
    if ud.done: break           # Прерываем цикл, если done == True
ud.close()                      # Закрываем объект
print ud.result                 # Выводим результат
raw_input()
```

Как показали мои тесты, применение класса `UniversalDetector` оправдано только при использовании кодировок UTF-8 с BOM[1], UTF-16 и других многобайтовых кодировок. В этом случае кодировка определяется по первым байтам. При использовании кодировки Windows-1251 файл все равно просматривается полностью. Причем определение кодировки файла, содержащего 6500 строк, занимает почти секунду. Если сменить кодировку файла на UTF-8 без BOM, то время определения увеличивается до 5 секунд. Использовать класс `UniversalDetector` или нет — решать вам.

[1] Byte order mark — метка порядка байтов.

Заключение

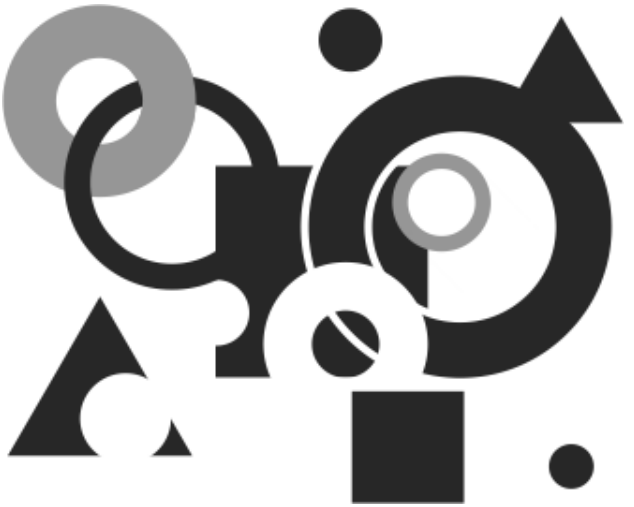
Вот и закончилось наше путешествие в мир Python. Материал книги описывает лишь базовые возможности этого универсального языка программирования. В этом заключении мы рассмотрим, где найти дополнительную информацию и продолжить изучение языка Python.

Первым и самым важным источником информации является сайт **http://www.python.org/**. На нем вы найдете последнюю версию интерпретатора, новости, а также ссылки на все другие ресурсы в Интернете.

На сайте **http://docs.python.org/** расположена документация по Python, которая обновляется в режиме реального времени. Язык постоянно совершенствуется, появляются новые функции, изменяются параметры, добавляются модули и т. д. Регулярно посещайте этот сайт, и вы получите самую последнюю информацию. Документация также устанавливается на компьютер в формате CHM и размещается внутри модулей. Как отобразить эту документацию, мы уже рассматривали в *разд. 1.8*.

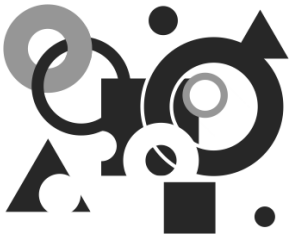
В пакет установки Python входит большое количество модулей, позволяющих решить наиболее часто встречающиеся задачи. Однако на этом возможности Python не заканчиваются. Мир Python включает множество самых разнообразных модулей и целых библиотек, созданных сторонними разработчиками и доступных для свободного скачивания. На странице **http://pypi.python.org/pypi?%3Aaction=index** и сайте **http://sourceforge.net/** вы сможете найти довольно большой список различных модулей. Особенно необходимо отметить библиотеки для создания графического интерфейса: PyQt (**http://www.riverbankcomputing.co.uk/software/pyqt/intro**), wxPython (**http://www.wxpython.org/**), PyGTK (**http://www.pygtk.org/**), PyWin32 (**http://sourceforge.net/projects/pywin32/**) и pyFLTK (**http://pyfltk.sourceforge.net/**). Кроме того, следует обратить внимание на библиотеку pygame (**http://www.pygame.org/**), позволяющую разрабатывать игры, а также на фреймворк Django (**http://www.djangoproject.com/**), с помощью которого можно создавать Web-приложения. При выборе модуля необходимо учитывать версию Python. Обычно версия указывается в составе названия исполняемого файла.

Если в процессе изучения языка возникнут вопросы, то не следует забывать, что сам Интернет предоставляет множество ответов на самые разнообразные вопросы. Достаточно в строке запроса поискового портала (например, **http://yandex.ru/** или **http://www.google.com/**) набрать свой вопрос. Наверняка уже кто-то сталкивался с подобным вопросом и описал решение на каком-либо сайте. Задать свой вопрос можно также на различных форумах (например, советую регулярно посещать **http://python.su/forum/**).



ПРИЛОЖЕНИЯ

Приложение 1



Отличия Python 3 от Python 2

Через некоторое время Python 2 уйдет в прошлое и на смену ему придет Python 3. Поэтому необходимо знать отличия этих двух версий, даже если вы используете вторую версию. В этом приложении мы рассмотрим основные отличия версии 3.1 от 2.6, ограничиваясь рамками материала этой книги. За дополнительной информацией обращайтесь к документации по Python 3.

- Оператор `print` заменяется функцией `print()`. Это изменение полностью нарушает совместимость между Python 2 и Python 3. Функция `print()` имеет следующий формат:

  ```
  print(<Объекты>[, sep=' '][, end='\n'][, file=sys.stdout])
  ```

  В первом параметре через запятую указываются объекты, которые выводятся:

  ```
  >>> print([1, 2, 3], (4, 5), {"x": 6})
  [1, 2, 3] (4, 5) {'x': 6}
  >>> print("Строка1", "Строка2")
  Строка1 Строка2
  ```

  Как видно из примера, между строками вставляется пробел. С помощью параметра `sep` можно указать другой символ. Выведем строки без пробела между ними:

  ```
  >>> print("Строка1", "Строка2", sep="")
  Строка1Строка2
  ```

  После вывода объектов в конце добавляется символ перевода строки. Если необходимо произвести дальнейший вывод на той же строке, то в параметре `end` следует указать другой символ:

  ```
  >>> for i in range(10): print(i, end=" ")
  0 1 2 3 4 5 6 7 8 9
  ```

  Параметр `file` позволяет произвести перенаправление вывода. По умолчанию параметр ссылается на объект `stdout`. Для примера перенаправим вывод в файл:

  ```
  >>> import sys
  >>> f = open("file.txt", "w")
  >>> print("Пишем строку в файл", file=f)
  ```

```
>>> print("Выводим строку")
Выводим строку
>>> f.close()
```

- Функция `raw_input()` заменяется функцией `input()`. Теперь для получения данных от пользователя необходимо применять функцию `input()`. Функция `raw_input()` была удалена. Пример:

```
# -*- coding: cp866 -*-
name = input("Введите свое имя: ") # Получаем данные
print("Привет, {0}".format(name))  # Выводим строку
input() # Ожидаем нажатия клавиши <Enter>
```

  Чтобы вернуться к поведению функции `input()` в Python 2, необходимо передать значение в функцию `eval()` явным образом:

```
result = eval(input("Введите инструкцию: "))
```

  ВНИМАНИЕ!

  Функция `eval()` выполнит любую введенную инструкцию. Никогда не используйте этот код, если не доверяете пользователю.

- Тип `unicode` заменяется типом `str`, а тип `str` типом `bytes`. Все строки по умолчанию теперь являются Unicode-строками. Тип `unicode` и модификатор `u` были удалены. Пример:

```
>>> type("Строка")
<class 'str'>
```

  Роль обычных строк заменяет тип `bytes`. Перед такими строками указывается модификатор `b`. Преобразовать Unicode-строку в обычную строку позволяет функция `bytes()`. Формат функции:

```
bytes([<Строка, имеющая тип str>[, <Кодировка>
      [, <Обработка ошибок>]]])
```

  Пример создания обычной строки:

```
>>> s = bytes("Строка", "cp1251")
>>> s
b'\xd1\xf2\xf0\xee\xea\xe0'
>>> type(s)
<class 'bytes'>
```

  Выполнить обратную операцию, т. е. преобразовать обычную строку в Unicode-строку, позволяет функция `str()`, которая в Python 3 имеет такой же формат, как и функция `unicode()` в Python 2:

```
str([<Строка, имеющая тип bytes>[, <Кодировка>
    [, <Обработка ошибок>]]])
```

  Пример преобразования обычной строки в Unicode-строку:

```
>>> s = bytes("Строка", "cp1251")
>>> u = str(s, "cp1251")
>>> u
'Строка'
```

Выполнить преобразование Unicode-строки в обычную строку позволяет также метод `str.encode()`, а для обратного преобразования можно воспользоваться методом `bytes.decode()`:

```
>>> s = "Строка".encode("cp1251") # Обычная строка
>>> s.decode("cp1251")            # Unicode-строка
'Строка'
```

- Тип `long` больше не существует. В Python 3 для целых чисел существует только тип `int`, который соответствует типу `long` в Python 2. Пример:

```
>>> type(2147483647), type(2147483648)
(<class 'int'>, <class 'int'>)
```

- Целочисленное деление возвращает вещественное число. Как вы уже знаете, в Python 2 при делении целых чисел всегда возвращалось целое число, а не вещественное. В Python 3 оператор `/` возвращает вещественное число, даже если производится деление целых чисел. Пример:

```
>>> 10 / 5, 10.0 / 5, 10 / 5.0    # Оператор /
(2.0, 2.0, 2.0)
>>> 10 // 5, 10.0 // 5, 10 // 5.0 # Оператор //
(2, 2.0, 2.0)
```

- Изменена форма записи двоичных и восьмеричных значений. При указании двоичного значения перед числом необходимо указать 0 и букву "b", а при указании восьмеричного значения — 0 и букву "o". Регистр букв значения не имеет. Пример:

```
>>> 0b100, 0B100, bin(4)
(4, 4, '0b100')
>>> 0o7, 0o12, 0o777, oct(511)
(7, 10, 511, '0o777')
```

- Оператор `<>` удален. Вместо него следует использовать оператор `!=`.
- Функция `xrange()` переименована в `range()`. В Python 3 функция `range()` возвращает объект, поддерживающий итерации. Чтобы получить список чисел, необходимо выполнить явное преобразование с помощью функции `list()`. Пример:

```
>>> range(10), type(range(10))
(range(0, 10), <class 'range'>)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Метод `next()` заменяется методом `__next__()` и добавляется функция `next()`. Объекты, поддерживающие итерации, должны иметь метод `__next__()`. Этот метод будет автоматически вызываться при указании объекта внутри цикла. Кроме того, в Python 3 вводится функция `next()`, которая вызывает метод `__next__()`. Пример:

```
>>> i = enumerate([1, 2])
>>> i.__next__(), next(i)
((0, 1), (1, 2))
```

- Функция `unichr()` переименована в `chr()`.
- Функции `map()`, `zip()` и `filter()` возвращают объекты, а не списки. Чтобы получить список, необходимо использовать функцию `list()`:

```
>>> map(lambda x, y, z: x + y + z, [1, 2], [4, 5], [6, 7])
<map object at 0x01360570>
>>> list(map(lambda x, y, z: x + y + z, [1, 2],[4, 5],[6, 7]))
[11, 14]
>>> zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
<zip object at 0x0135F418>
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>> filter(None, [1, 0, None, [], 2])
<filter object at 0x01360570>
>>> list(filter(None, [1, 0, None, [], 2]))
[1, 2]
```

- Добавлен новый способ создания множества. Создать множество можно не только с помощью функции `set()`, но и указав элементы через запятую внутри фигурных скобок:

```
>>> set([1, 2, 3, 4, 5])
{1, 2, 3, 4, 5}
>>> {1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

- Метод словаря `has_key()` удален. Чтобы проверить существование ключа в словаре, следует использовать оператор `in`:

```
>>> "a" in { "a": 1, "b": 2 }
True
```

- Методы словаря `keys()`, `values()` и `items()` возвращают объекты, поддерживающие итерации. Чтобы получить список, необходимо использовать функцию `list()`:

```
>>> d = { "a": 1, "b": 2 }
>>> d.keys(), d.values()
(dict_keys(['a', 'b']), dict_values([1, 2]))
>>> d.items()
dict_items([('a', 1), ('b', 2)])
>>> list(d.keys()), list(d.values())
(['a', 'b'], [1, 2])
>>> list(d.items())
[('a', 1), ('b', 2)]
```

  Так как метод `keys()` возвращает объект, а не список, то для вывода отсортированного по ключам словаря следует использовать функцию `sorted()`:

```
>>> d = { "a": 1, "b": 2 }
>>> for key in sorted(d.keys()): print(key, d[key])
a 1
b 2
```

- Методы словаря `iterkeys()`, `itervalues()` и `iteritems()` удалены.
- Функции `cmp()`, `reduce()`, `apply()` и `file()` удалены.
- Функция `reload()` перемещена в модуль `imp`. Чтобы повторно загрузить модуль, следует подключить модуль `imp`, а затем передать идентификатор функции `imp.reload()`. Пример повторной загрузки модуля `math`:

```
>>> import math
>>> import imp          # Подключаем модуль imp
>>> imp.reload(math)    # Повторно загружаем модуль math
<module 'math' (built-in)>
```

- Все классы нового стиля. Классы, называемые классическими в Python 2, больше не поддерживаются. Наследовать объект `object` при создании класса необязательно:

```
class Class1:           # Класс нового стиля
    pass
class Class2(object):   # и это класс нового стиля
    pass
print(type(Class1))     # Выведет: <class 'type'>
print(type(Class2))     # Выведет: <class 'type'>
```

- Параметры в инструкции `except` должны разделяться оператором `as`. В Python 2.6 для разделения параметров можно было использовать запятую и оператор `as`. В Python 3 допустимо использовать только оператор `as`:

```
try:
    x = 1 / 0
except NameError as err:
    print(err)
except (IndexError, ZeroDivisionError) as err:
    print(err)
```

- Файловые методы `xreadlines()` и `next()` удалены.
- Класс `StringIO` перемещен в модуль `io`.

```
>>> from io import StringIO
>>> f = StringIO("String1\n")
>>> f.seek(0, 2)          # Перемещаем указатель в конец
8
>>> f.write("String2\n") # Записываем строку
8
>>> f.writelines(["String3\n", "String4\n"])
>>> f.seek(0)             # Перемещаем указатель в начало
0
>>> f.readline()          # Считываем строку
'String1\n'
>>> f.read(8)             # Считываем 8 байт
'String2\n'
>>> f.readlines()         # Считываем строки в список
['String3\n', 'String4\n']
>>> f.close()             # Закрываем "файл"
```

- Функции `urlparse()`, `urlunparse()`, `urlsplit()`, `urlunsplit()`, `parse_qs()`, `parse_qsl()`, `urlencode()`, `quote()`, `quote_plus()`, `unquote()`, `unquote_plus()` и `urljoin()` перемещены в модуль `urllib.parse`.

```
>>> from urllib.parse import urlparse, urlunparse
>>> url = urlparse("http://site.ru:80/test.php;st?v=5#m")
>>> url.netloc, url.hostname, url.port, url.path
('site.ru:80', 'site.ru', 80, '/test.php')
>>> t = ('http', 'site.ru:80', '/test.php', '', 'v=5', 'm')
>>> urlunparse(t)
'http://site.ru:80/test.php?v=5#m'
```

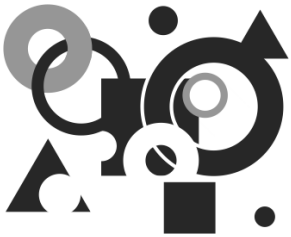
- Класс `HTTPConnection` перемещен в модуль `http.client`.

```
>>> import http.client, urllib.parse
>>> d = urllib.parse.urlencode({"color": "red", "var": 15})
>>> h = { "User-Agent": "MySpider/1.0",
        "Accept": "text/html, text/plain, application/xml",
        "Accept-Language": "ru, ru-RU",
        "Accept-Charset": "windows-1251",
        "Referer": "/index.php" }
>>> con = http.client.HTTPConnection("wwwadmin.ru")
>>> con.request("GET", "/testrobots.php?%s" % d, headers=h)
>>> result = con.getresponse()
>>> print(result.read().decode("cp1251"))
... Фрагмент опущен ...
>>> con.close()
```

- Вместо модуля `urllib2` следует использовать модуль `urllib.request`.

```
>>> import urllib.parse, urllib.request as u
>>> d = urllib.parse.urlencode({"color": "red", "var": 15})
>>> h = { "User-Agent": "MySpider/1.0",
        "Accept": "text/html, text/plain, application/xml",
        "Accept-Language": "ru, ru-RU",
        "Accept-Charset": "windows-1251",
        "Referer": "/index.php" }
>>> url = "http://wwwadmin.ru/testrobots.php?%s" % d
>>> request = u.Request(url, headers=h)
>>> res = u.urlopen(request)
>>> print(res.read().decode("cp1251"))
... Фрагмент опущен ...
>>> res.close()
```

Приложение 2



Описание DVD

Структура DVD, прилагаемого к книге, представлена в табл. П2.1.

Таблица П2.1. *Описание DVD*

Папка или файл	Файл	Описание
\Foto		Папка с цветными иллюстрациями
\Video		Папка с видеороликами
	Start.avi	Способы создания и запуска программы: • открываем и просматриваем документацию в формате CHM; • просматриваем документацию, расположенную внутри модулей; • рассматриваем способ поиска документации к модулю; • запускаем интерактивную оболочку и выполняем несколько инструкций; • создаем ярлык на рабочем столе для запуска IDLE; • запускаем IDLE и выполняем несколько инструкций; • создаем новый файл с программой; • рассматриваем различные способы запуска программы; • изменяем настройки IDLE; • запускаем программу из командной строки
	Versions.avi	Запуск с помощью разных версий Python: • запускаем интерактивную оболочку для разных версий Python; • запускаем редактор PythonWin для Python 2.5 и 3.1;

Таблица П2.1 *(продолжение)*

Папка или файл	Файл	Описание
		• запускаем IDLE для разных версий Python; • запускаем программу проверки установки с помощью разных версий; • запускаем программу с помощью редактора PythonWin; • запускаем программу из командной строки
	Print.avi	Вывод данных и перенаправление вывода. В этом видеоролике рассматривается вывод данных с помощью оператора `print` и метода `write()`, а также перенаправление вывода в файл. Дополнительную информацию *см. в разд. 1.6* и *15.8*
	Input.avi	Ввод данных и перенаправление ввода. В этом видеоролике рассматривается ввод данных с помощью функции `raw_input()`, получение параметров, переданных в командной строке, а также перенаправление ввода. Дополнительную информацию *см. в разд. 1.7 и 15.8*
	Number.avi	Работа с числами. Рассматриваются основные операции при работе с числами. Дополнительную информацию *см. в главе 5*
	String.avi	Работа со строками. Рассматриваются основные операции при работе со строками и регулярными выражениями. Дополнительную информацию *см. в главах 6* и *7*
	List.avi	Работа со списками, кортежами и множествами. Рассматриваются основные операции при работе со списками, кортежами и множествами. Дополнительную информацию *см. в главе 8*
	Dict.avi	Работа со словарями. Рассматриваются основные операции при работе со словарями. Дополнительную информацию *см. в главе 9*
	Internet.avi	Обмен данными по протоколу HTTP. Рассматривается обмен данными по протоколу HTTP с помощью модулей httplib и urllib2. Дополнительную информацию *см. в разд. 20.5* и *20.6*
	Headers.avi	Способы просмотра HTTP-заголовков: • открываем файл index.php с помощью Notepad++. Файл содержит ссылку, при переходе по которой данные будут отправлены методом `GET`, и форму, данные которой будут отправлены методом `POST`;

Таблица П2.1 *(продолжение)*

Папка или файл	Файл	Описание
		• открываем Web-браузер Firefox и запускаем файл index.php; • запускаем Firebug и переходим на вкладку **Сеть**; • просматриваем HTTP-заголовки, отправляя данные методами `GET` и `POST`; • запускаем Web-браузер Internet Explorer и открываем панель ieHTTPHeaders; • просматриваем HTTP-заголовки, отправляя данные методами `GET` и `POST`; • сохраняем HTTP-заголовки в файл и просматриваем результат
	phpMyAdmin.avi	Обзор программы phpMyAdmin: • запускаем серверы Apache и MySQL; • с помощью Web-браузера открываем программу phpMyAdmin; • создаем новую базу данных, а затем таблицу; • вставляем данные в таблицу; • создаем индекс. Чтобы увидеть количество элементов в индексе, производим оптимизацию таблицы; • удаляем индекс, данные, таблицу и базу данных; • далее рассматриваем способы создания дампа всей базы данных и отдельной таблицы; • проверяем таблицу; • производим восстановление таблицы
	HeidiSQL.avi	Обзор программы HeidiSQL: • запускаем серверы Apache и MySQL; • открываем программу HeidiSQL и вводим параметры подключения к MySQL; • для установки соединения с сервером MySQL выбираем сохраненное соединение из списка **Description** и нажимаем кнопку **Connect**; • отображаем содержимое базы данных, а затем таблицы; • выполняем SQL-запрос на вкладке **Query**; • ограничиваем набор данных с помощью создания фильтра, а также указания количества строк; • добавляем новую запись и удаляем существующую;

Таблица П2.1 *(окончание)*

Папка или файл	Файл	Описание
		• создаем новую базу данных и таблицу в ней; • далее рассматриваем способ создания дампа таблицы; • удаляем таблицу и базу данных
\Setup.html		Иллюстрированное описание установки Python
\Listings.doc		Все листинги из книги (если нет программы Word, то прочитать можно с помощью OpenOffice)
\Readme.txt, \Readme.doc		Описание DVD

Предметный указатель

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Б

В

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ч

Я