Н. А. Прохоренок, В. А. Дронов

bhv

Python 3

САМОЕ НЕОБХОДИМОЕ

Основы языка Python 3
Классы и объекты
Итераторы и перечисления
Обработка исключений
Работа с файлами и каталогами
Основы SQLite
Доступ к данным SQLite и MySQL
Использование ODBC
Pillow и Wand: работа с графикой
Получение данных из Интернета
Сжатие и распаковка данных
Примеры и советы из практики

Материалы на www.bhv.ru

Николай Прохоренок
Владимир Дронов

Python 3

САМОЕ НЕОБХОДИМОЕ

Санкт-Петербург
«БХВ-Петербург»
2016

Оглавление

Введение

Добро пожаловать в мир Python!

Python — это интерпретируемый, объектно-ориентированный, тьюринг-полный язык программирования высокого уровня, предназначенный для решения самого широкого́ круга задач. С его помощью можно обрабатывать числовую и текстовую информацию, создавать изображения, работать с базами данных, разрабатывать Web-сайты и приложения с графическим интерфейсом. Python — язык кроссплатформенный, он позволяет создавать программы, которые будут работать во всех операционных системах. В этой книге мы рассмотрим базовые возможности Python версии 3.4 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 «Летающий цирк Монти Пайтона» (Monty Python's Flying Circus). Поэтому правильное произношение названия этого замечательного языка — Пайтон.

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, который 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 или C++, скомпилировать его, а затем подключить к основной программе.

Python относится к категории языков объектно-ориентированных. Это означает, что практически все данные в нем являются объектами, даже значения, относящиеся к элементарным типам, наподобие чисел и строк, а также 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. Такое 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 и объектно-ориентированный стиль программирования (ООП) является при его использовании обязательным. На языке Python можно писать программы как в стиле ООП, так и в процедурном стиле, — как того требует конкретная ситуация или как предпочитает программист.

Python — самый стильный язык программирования в мире, он не допускает двоякого написания кода. Так, языку Perl присущи зависимость от контекста и множественность синтаксиса, и часто два программиста, пишущих на Perl, просто не понимают код друг друга. В Python же код можно написать только одним способом. В нем отсутствуют лишние конструкции. Все программисты должны придерживаться стандарта PEP-8, описанного в документе **https://www.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 программирования. На первый взгляд может показаться, что отсутствие ограничительных символов (фигурных скобок или конструкции `begin...end`) для выделения блоков и обязательная вставка пробелов впереди инструкций могут 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 ни к чему. Бытует мнение, что программа будет по-разному смотреться в разных редакторах. Это неверно. Согласно стандарту, для выделения блоков необходимо использовать *четыре пробела*. А 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 окажется разным, то интерпретатор выведет сообщение о фатальной ошибке, и программа будет остановлена. Таким образом, язык Python приучает программистов писать красивый и понятный код.

Поскольку 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 также выводят различные подсказки и позволяют выполнять отладку программы. Таких редакторов очень много: PyScripter, PythonWin, UliPad, Eclipse с установленным модулем PyDev, Netbeans и др. — полный список редакторов можно найти на странице **http://wiki.python.org/moin/PythonEditors**. Мы же в процессе изложения материала этой книги будем пользоваться интерактивным интерпретатором IDLE, который входит в состав стандартной библиотеки Python в Windows, — он 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, весьма желательно выполнять все имеющиеся в них примеры, а также экспериментировать, что-нибудь в этих примерах изменяя. Все листинги из этой книги вы найдете в файле Listings.doc, электронный архив с которым можно загрузить с FTP-сервера издательства «БХВ-Петербург» по ссылке: **ftp://ftp.bhv.ru/9785977536318.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

Сообщения обо всех замеченных ошибках и опечатках, равно как и возникающие в процессе чтения книги вопросы, авторы просят присылать на адрес издательства «БХВ-Петербург»: **mail@bhv.ru**.

Желаем приятного прочтения и надеемся, что эта книга выведет вас на верный путь в мире профессионального программирования.

ГЛАВА 1

Первые шаги

Прежде чем мы начнем рассматривать синтаксис языка, необходимо сделать два замечания. Во-первых, как уже было отмечено во *введении*, не забывайте, что книги по программированию нужно не только читать — весьма желательно выполнять все имеющиеся в них примеры, а также экспериментировать, что-нибудь в этих примерах изменяя. Поэтому, если вы удобно устроились на диване и настроились просто читать, у вас практически нет шансов изучить язык. Во-вторых, помните, что прочитать такую книгу один раз недостаточно. Начальные главы книги вы должны выучить наизусть! Сколько на это уйдет времени, зависит от ваших способностей и желания. Чем больше вы будете делать самостоятельно, тем большему научитесь. Ну что, приступим к изучению языка? Python достоин того, чтобы его знал каждый программист!

1.1. Установка Python

Вначале необходимо установить на компьютер *интерпретатор* Python (его также называют *исполняющей средой*).

1. Для загрузки дистрибутива переходим на страницу **https://www.python.org/downloads/** и в списке доступных версий щелкаем на гиперссылке **Python 3.4.3** (эта версия является самой актуальной из стабильных версий на момент подготовки книги). На открывшейся странице находим раздел **Files** и щелкаем на гиперссылке **Windows x86 MSI installer** (32-разрядная версия интерпретатора) или **Windows x86-64 MSI installer** (его 64-разрядная версия). В результате на наш компьютер будет загружен файл python-3.4.3.msi или python-3.4.3.amd64.msi соответственно. Затем запускаем загруженный файл двойным щелчком на нем.
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) нам предлагается выбрать каталог для установки. Оставляем каталог по умолчанию (C:\Python34\) и нажимаем кнопку **Next**.
4. В следующем диалоговом окне (рис. 1.3) выбираем компоненты, которые необходимо установить. По умолчанию устанавливаются все компоненты и прописывается ассоциация с файловыми расширениями py, pyw и др. В этом случае запускать Python-программы можно будет с помощью двойного щелчка мышью на значке файла. Оставляем выбранными все компоненты и нажимаем кнопку **Next**.

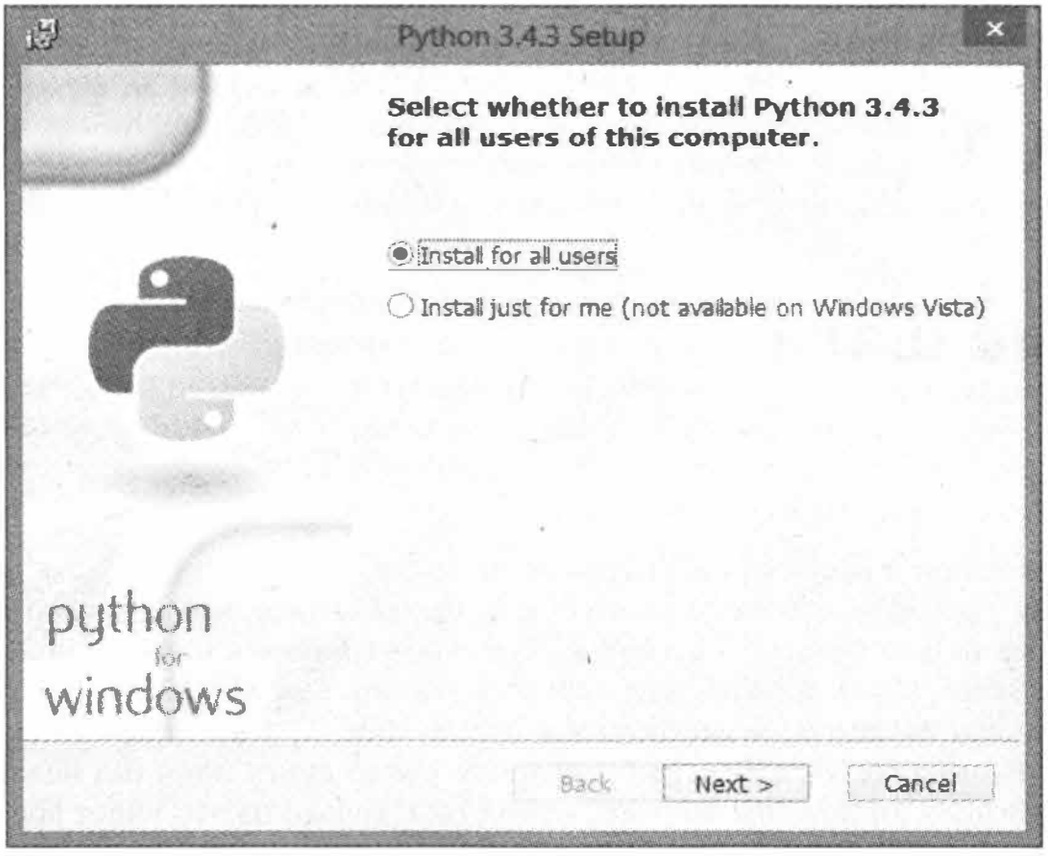


Рис. 1.1. Установка Python. Шаг 1

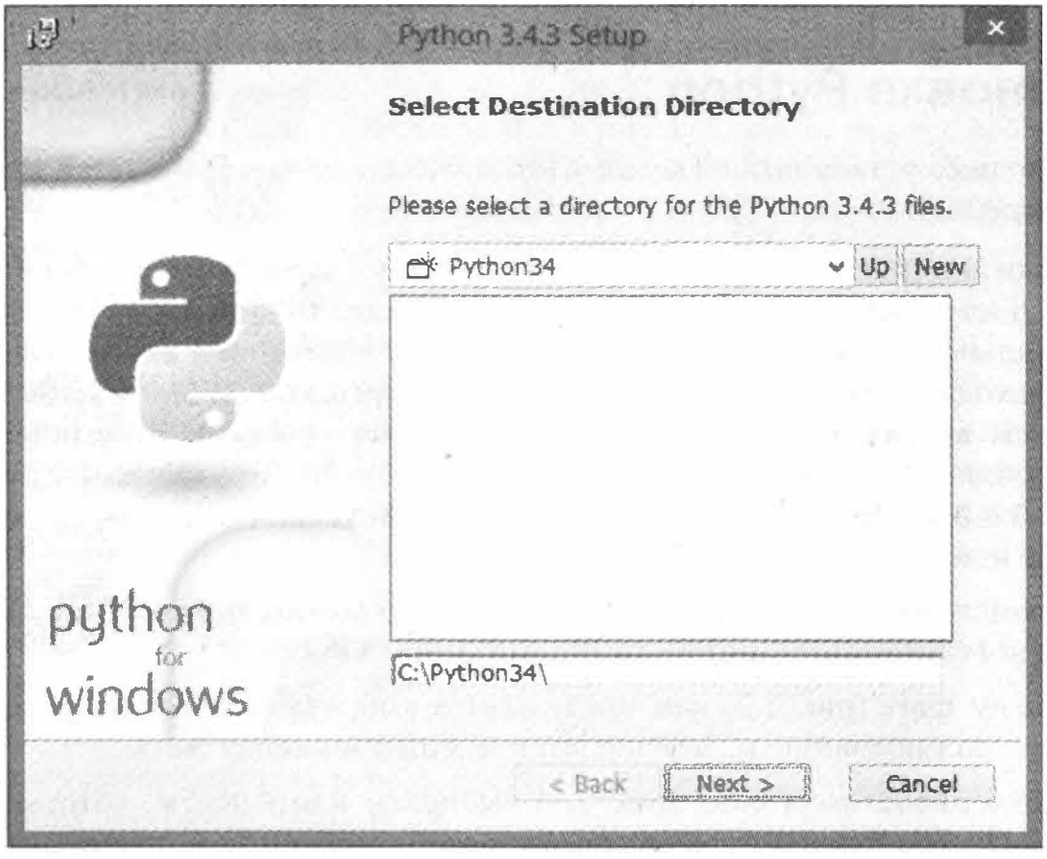


Рис. 1.2. Установка Python. Шаг 2

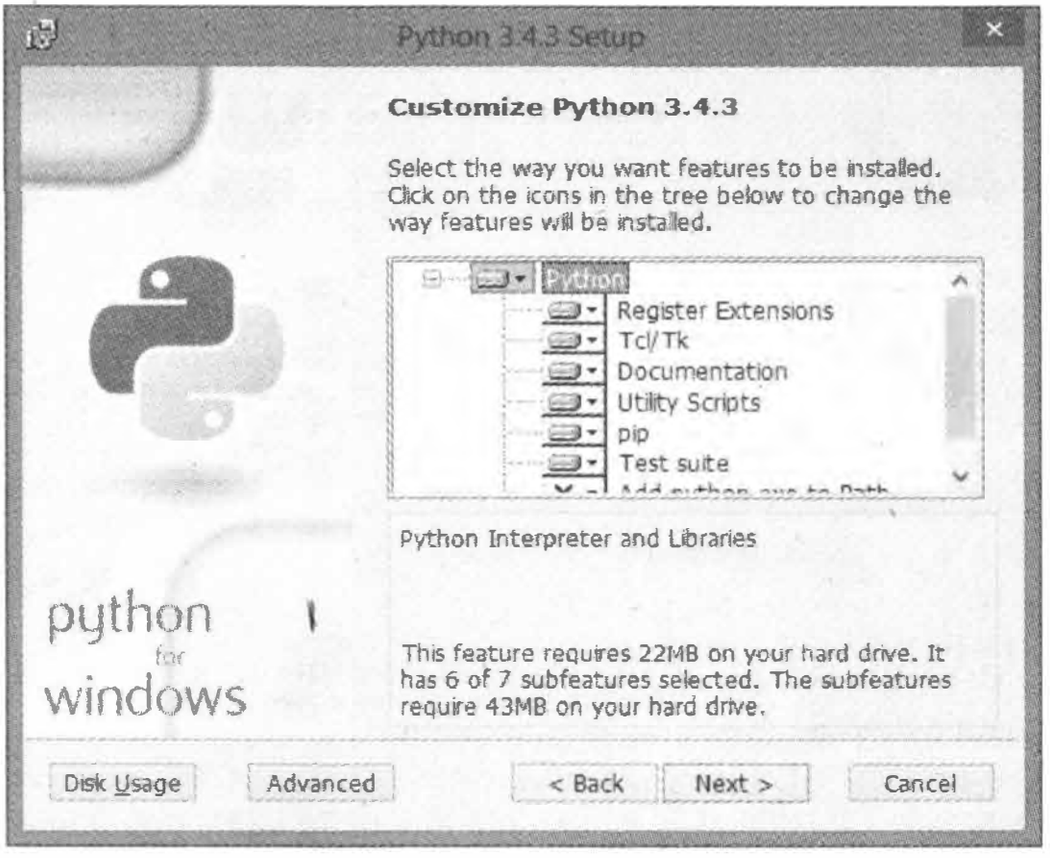


Рис. 1.3. Установка Python. Шаг 3

Примечание

Пользователям Windows Vista и более поздних версий этой системы следует положительно ответить на появившееся на экране предупреждение системы UAC (Контроль учетных записей). Если этого не сделать, Python установлен не будет.

5. После завершения установки откроется окно, изображенное на рис. 1.4. Нажимаем кнопку **Finish** для выхода из программы установки.

В результате установки файлы интерпретатора будут скопированы в папку C:\Python34. В этой папке вы найдете два исполняемых файла: python.exe и pythonw.exe. Файл python.exe предназначен для выполнения консольных приложений. Именно эта программа запускается при двойном щелчке на файле с расширением py. Файл pythonw.exe служит для запуска оконных приложений (при двойном щелчке на файле с расширением pyw) — в этом случае окно консоли выводиться не будет.

Итак, если выполнить двойной щелчок на файле python.exe, то интерактивная оболочка запустится в окне консоли (рис. 1.5). Символы >>> в этом окне означают приглашение для ввода инструкций 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 новой инструкции. Таким образом, это окно можно использовать в качестве калькулятора, а также для изучения языка.

Открыть такое же окно можно с помощью пункта **Python 3.4 (command line - 32 bit)** или **Python 3.4 (command line - 64 bit)** в меню **Пуск | Программы (Все программы) | Python 3.4**.

Вместо такой интерактивной оболочки для изучения языка, а также для создания и редактирования файлов с программой лучше воспользоваться редактором IDLE, который входит.

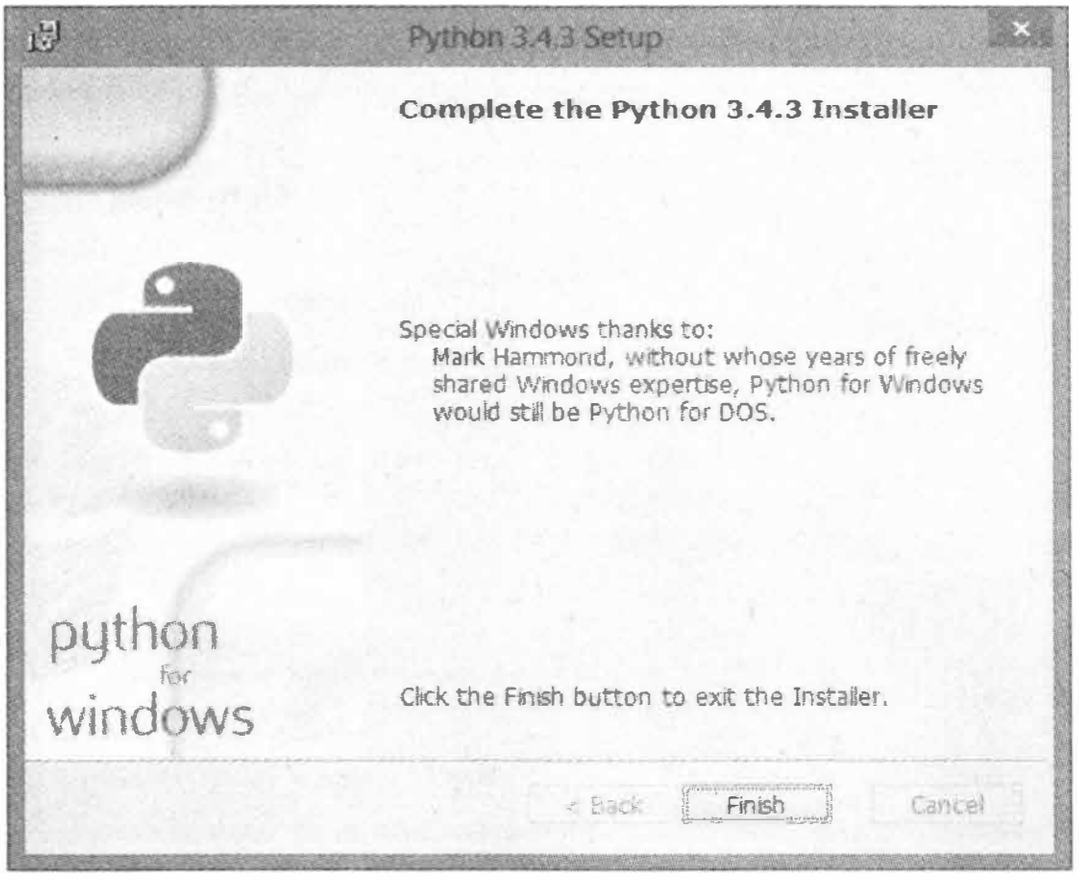


Рис. 1.4. Установка Python. Шаг 4

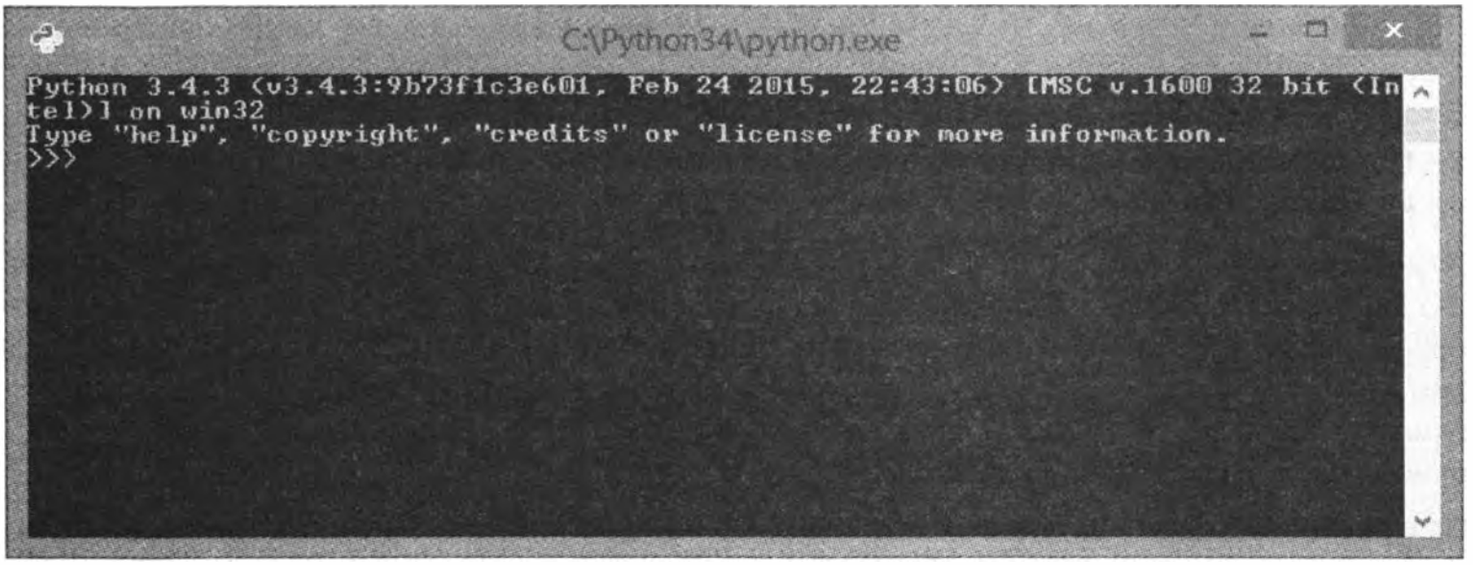


Рис. 1.5. Интерактивная оболочка

в состав установленных компонентов. Для запуска редактора в меню **Пуск | Программы (Все программы) | Python 3.4** выбираем пункт **IDLE (Python 3.4 GUI - 32 bit)** или **IDLE (Python 3.4 GUI - 64 bit)**. 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 в процессе изучения материала книги. Более подробно редактор IDLE мы рассмотрим немного позже.

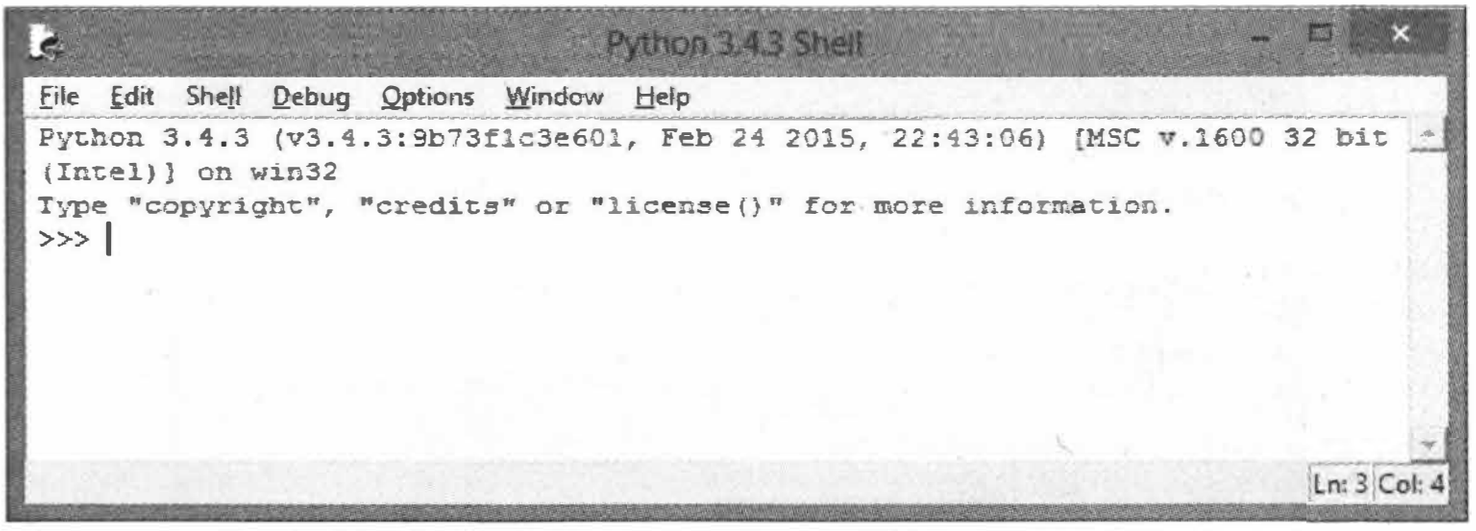


Рис. 1.6. Окно **Python Shell** редактора IDLE

1.1.1. Установка нескольких интерпретаторов Python

Версии языка Python выпускаются с завидной регулярностью, но, к сожалению, сторонние разработчики не успевают за такой скоростью и не столь часто обновляют свои модули. Поэтому приходится при наличии версии Python 3 использовать на практике также и версию Python 2. Как же быть, если установлена версия 3.4, а необходимо запустить модуль для версии 2.7? В этом случае удалять версию 3.4 с компьютера не нужно. Все программы установки позволяют выбрать устанавливаемые компоненты. Существует также возможность задать ассоциацию запускаемой версии с файловым расширением — так вот эту возможность необходимо при установке просто отключить.

В качестве примера мы дополнительно установим на компьютер версию 2.7.8.10, но вместо программы установки с сайта **https://www.python.org/** выберем альтернативный дистрибутив от компании ActiveState.

Итак, переходим на страницу **http://www.activestate.com/activepython/downloads/** и скачиваем дистрибутив. Последовательность запуска нескольких программ установки от компании ActiveState имеет значение, поскольку в контекстное меню добавляется пункт **Edit with Pythonwin**. С помощью этого пункта запускается редактор PythonWin, который можно использовать вместо IDLE. Соответственно, из контекстного меню будет открываться версия PythonWin, которая была установлена последней. Установку программы производим в каталог по умолчанию (C:\Python27\).

> ***Внимание!***
>
> При установке в окне **Custom Setup** (рис. 1.7) необходимо отключить компонент **Register as Default Python** (рис. 1.8). Не забудьте это сделать, иначе Python 3.4.3 перестанет быть текущей версией.

В состав ActivePython, кроме редактора PythonWin, входит также редактор IDLE. Однако ни в одном меню нет пункта, с помощью которого можно его запустить. Чтобы это исправить, создадим файл IDLE27.cmd со следующим содержимым:

```
@echo off
start C:\Python27\pythonw.exe C:\Python27\Lib\idlelib\idle.pyw
```

С помощью двойного щелчка на этом файле можно будет запускать редактор IDLE для версии Python 2.7.

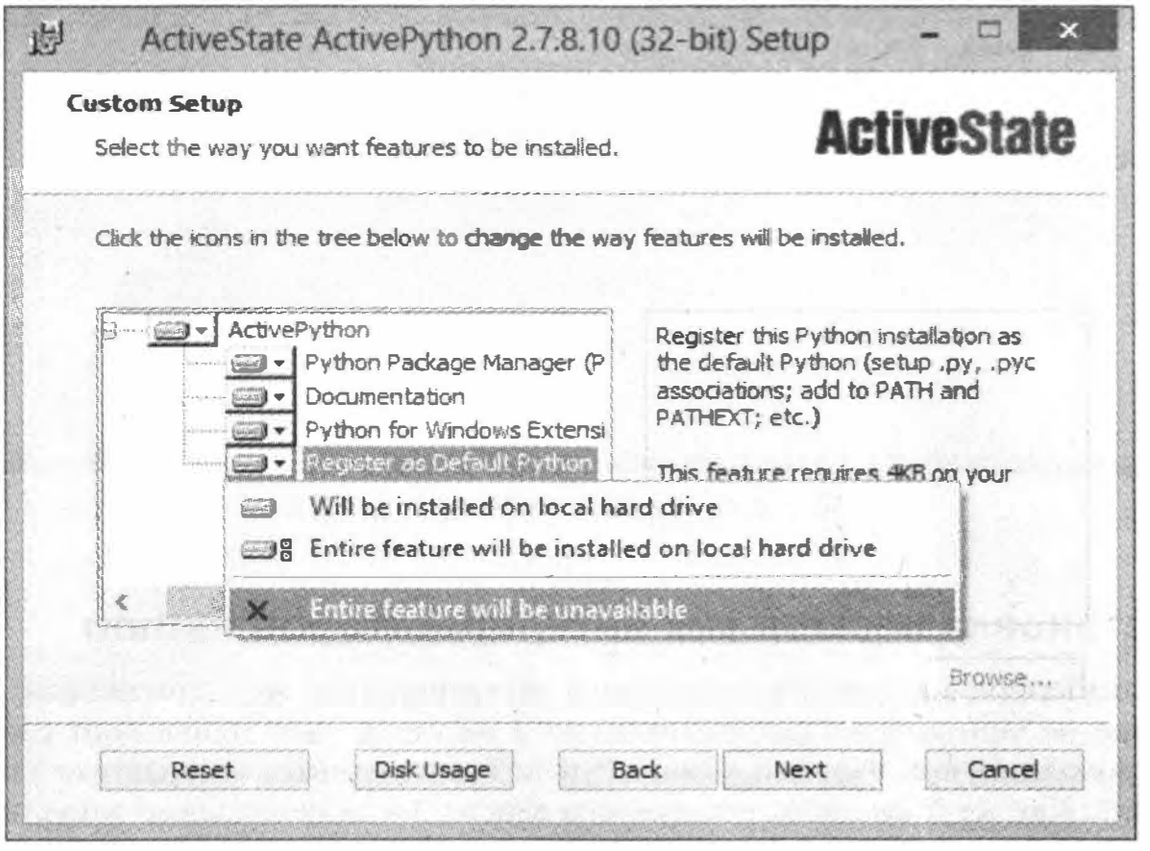


Рис. 1.7. Окно **Custom Setup**

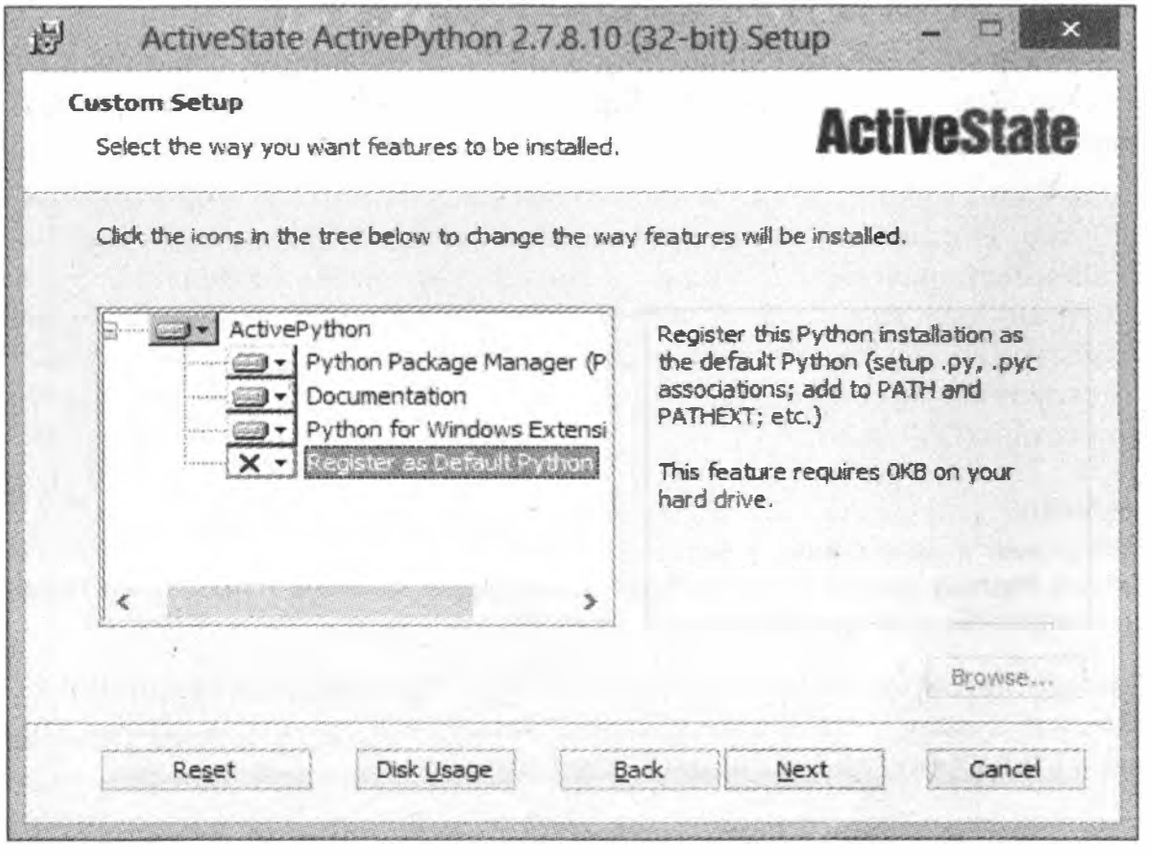


Рис. 1.8. Компонент **Register as Default Python** отключен

Ну, а запуск IDLE для версии Python 3.4 будет по-прежнему осуществляться так же, как и предлагалось ранее, — выбором в меню **Пуск | Программы (Все программы) | Python 3.4** пункта **IDLE (Python 3.4 GUI - 32 bit)** или **IDLE (Python 3.4 GUI - 64 bit)**.

1.1.2. Запуск программы с помощью разных версий Python

Теперь рассмотрим запуск программы с помощью разных версий Python. По умолчанию при двойном щелчке на значке файла запускается Python 3.4. Чтобы запустить Python-программу с помощью другой версии этого языка, щелкаем правой кнопкой мыши на значке файла с программой и в контекстном меню находим пункт **Открыть с помощью**.

В Windows XP при выборе этого пункта появится подменю, в котором изначально будет присутствовать только программа python.exe. Чтобы добавить другую версию, щелкаем на пункте **Выбрать программу**, в открывшемся окне нажимаем кнопку **Обзор** и выбираем программу python2.7.exe из папки C:\Python27. Выбранная нами программа будет добавлена в подменю, открывающееся при выборе пункта **Открыть с помощью**. Впоследствии для выполнения файла под управлением Python 2.7 мы просто выберем этот пункт.

В Windows Vista и более поздних версиях этой системы при выборе упомянутого пункта изначально не будет открываться никакого подменю. Вместо этого на экране появится небольшое окно выбора альтернативной программы для запуска файла (рис. 1.9). Сразу же сбросим флажок **Использовать это приложение для всех файлов .py** и нажмем ссылку **Дополнительно**. В окне появится список установленных на нашем компьютере программ, но нужного нам приложения python2.7.exe в нем не будет. Поэтому щелкнем на ссылке **Найти другое приложение на этом компьютере**, находящейся под списком. На экране появится стандартное диалоговое окно открытия файла, в котором мы выберем программу python2.7.exe из папки C:\Python27. Теперь эта программа появится в подменю, открывающемся при выборе пункта **Открыть с помощью** (рис. 1.10), — здесь Python 2.7 представлен как **Python Launcher for Windows (Console)**.

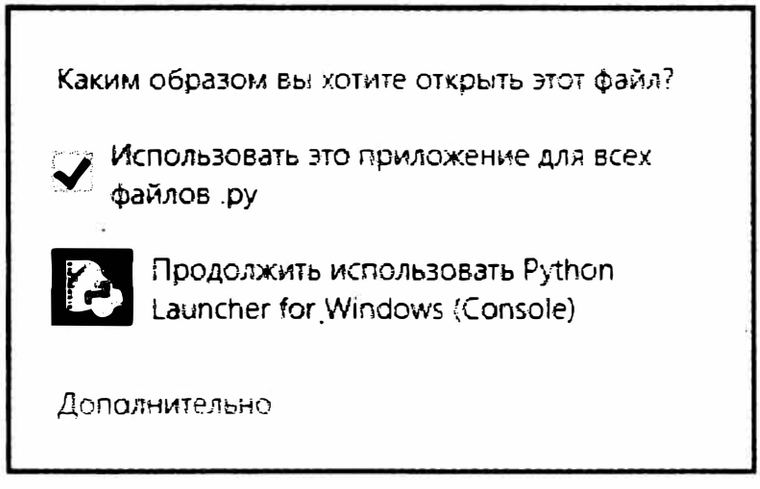


Рис. 1.9. Диалоговое окно выбора альтернативной программы для запуска файла

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
```

Затем запустите программу с помощью двойного щелчка на значке файла. Если результат выполнения: `(3, 4, 3, 'final', 0)`, то установка прошла нормально, а если `(2, 7, 8, 'final', 0)`, то вы не отключили компонент **Register as Default Python**.

Для изучения материала этой книги по умолчанию должна запускаться версия Python 3.4.

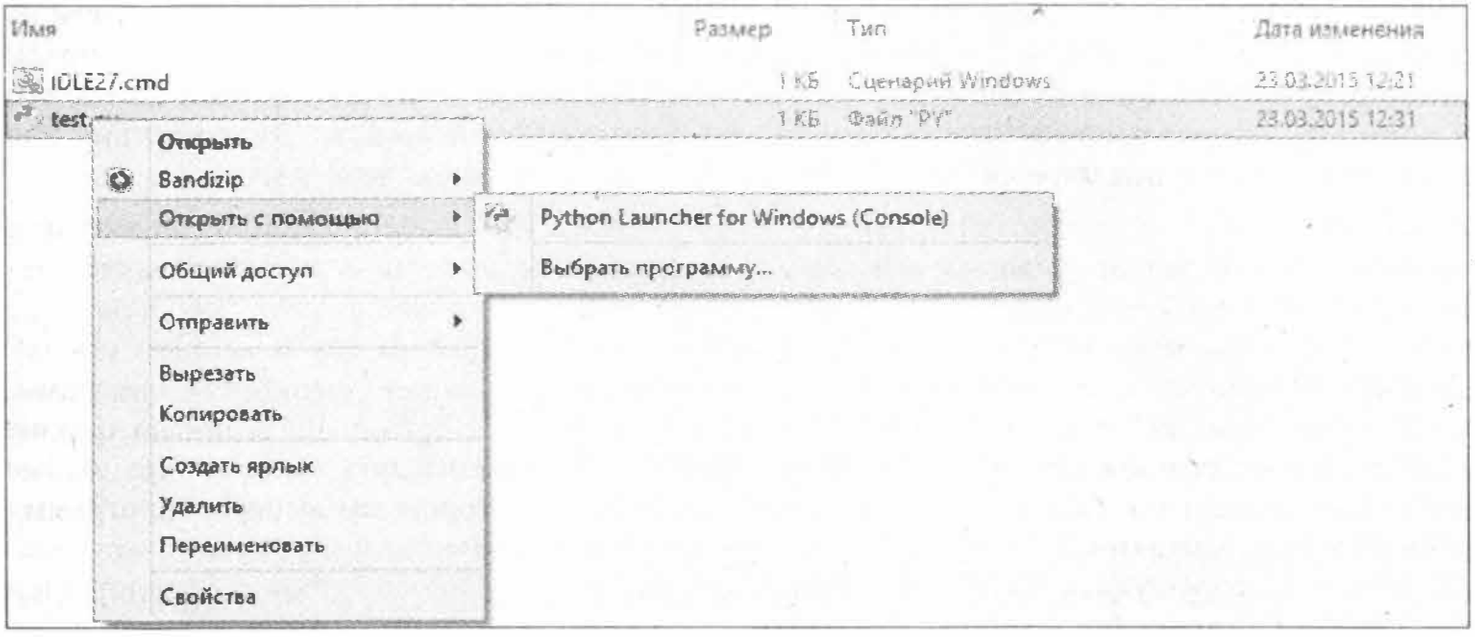


Рис. 1.10. Варианты запуска программы разными версиями Python

1.2. Первая программа на Python

Изучение 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 функции print()
print("Привет, мир!")
```

Для запуска программы в меню **Пуск | Программы (Все программы) | Python 3.4** выбираем пункт **IDLE (Python 3.4 GUI - 32 bit)** или **IDLE (Python 3.4 GUI - 64 bit)**. В результате откроется окно **Python Shell**, в котором символы >>> означают приглашение ввести команду. Вводим сначала первую строку из листинга 1.2, а затем вторую. После ввода каждой строки 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 функции 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 File**. В открывшемся окне набираем код из листинга 1.2, а затем сохраняем его под именем hello_world.py, выбрав пункт меню **File | Save As**. При этом редактор сохранит файл в кодировке UTF-8 без BOM (Byte Order Mark, метка порядка байтов). Именно кодировка UTF-8 является кодировкой по умолчанию в Python 3. Если файл содержит инструкции в другой кодировке, то необходимо в первой или второй строке указать кодировку с помощью инструкции:

```
# -*- coding: <Кодировка> -*-
```

Например, для кодировки Windows-1251 инструкция будет выглядеть так:

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

Редактор IDLE учитывает указанную кодировку и автоматически производит перекодирование при сохранении файла. При использовании других редакторов следует проконтролировать соответствие указанной кодировки и реальной кодировки файла. Если кодировки не совпадают, то данные будут преобразованы некорректно, или во время преобразования произойдет ошибка.

Запустить программу на выполнение можно, выбрав пункт меню **Run | Run Module** или нажав клавишу <F5>. Результат выполнения программы будет отображен в окне **Python Shell**.

Запустить программу можно также с помощью двойного щелчка мыши на значке файла. В этом случае результат выполнения будет отображен в консоли Windows. Следует учитывать, что после вывода результата окно консоли сразу закрывается. Чтобы предотвратить закрытие окна, необходимо добавить вызов функции `input()`, которая станет ожидать нажатия клавиши <Enter> и не позволит окну сразу закрыться. С учетом сказанного наша программа будет выглядеть так, как показано в листинге 1.4.

Листинг 1.4. Программа для запуска с помощью двойного щелчка мыши

```
# -*- coding: utf-8 -*-
print("Привет, мир!")          # Выводим строку
input()                        # Ожидаем нажатия клавиши <Enter>
```

Примечание

Если до функции `input()` возникнет ошибка, то сообщение о ней будет выведено в консоль, но сама консоль после этого сразу закроется, и вы не сможете прочитать сообщение об ошибке. Попав в подобную ситуацию, запустите программу из командной строки или с помощью редактора IDLE и вы сможете прочитать сообщение об ошибке.

В языке Python 3 строки по умолчанию хранятся в кодировке Unicode. При выводе кодировка Unicode автоматически преобразуется в кодировку терминала. Поэтому русские буквы

отображаются корректно, хотя в окне консоли в Windows по умолчанию используется кодировка cp866, а файл с программой у нас в кодировке UTF-8.

Чтобы отредактировать уже созданный файл, запустим IDLE, выполним команду меню **File | Open** и укажем нужный файл, который будет открыт в другом окне.

> ***Напоминание***
>
> Поскольку программа на языке Python представляет собой обычный текстовый файл, сохраненный с расширением py или pyw, его можно редактировать с помощью других программ — например, Notepad++. Можно также воспользоваться специализированными редакторами — скажем, PyScripter.

Когда интерпретатор Python начинает выполнение программы, хранящейся в файле, он сначала компилирует ее в особое внутреннее представление, — это делается с целью увеличить производительность кода. Файл с откомпилированным кодом хранится в папке __pycache__, вложенной в папку, где хранится сам файл программы, а его имя имеет следующий вид:

<имя файла с исходным, неоткомпилированным кодом>.cpython-<первые две цифры номера версии Python>.pyc

Так, при запуске на исполнение файла test4.py будет создан файл откомпилированного кода с именем test4.cpython-34.pyc.

При последующем запуске того же файла на выполнение будет исполняться именно откомпилированный код. Если же мы исправим исходный код, программа его автоматически перекомпилирует. При необходимости мы можем удалить файлы с откомпилированным кодом или даже саму папку __pycache__ — впоследствии интерпретатор сформирует их заново.

1.3. Структура программы

Как вы уже знаете, программа на языке Python представляет собой обычный текстовый файл с инструкциями. Каждая инструкция располагается на отдельной строке. Если инструкция не является вложенной, то она должна начинаться с начала строки, иначе будет выведено сообщение об ошибке (листинг 1.5).

Листинг 1.5. Ошибка SyntaxError

```
>>>  import sys

SyntaxError: unexpected indent
>>>
```

В этом случае перед инструкцией `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
```

В некоторых операционных системах 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, необходимо, чтобы в правах доступа к файлу был установлен бит на выполнение. 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`).

Во второй строке (для ОС Windows в первой строке) следует указать кодировку. Если кодировка не указана, то предполагается, что файл сохранен в кодировке UTF-8. 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 1.6.

Листинг 1.6. Вывод списка поддерживаемых кодировок

```
# -*- coding: utf-8 -*-
import encodings.aliases
arr = encodings.aliases.aliases
keys = list( arr.keys() )
keys.sort()
for key in keys:
    print("%s => %s" % (key, arr[key]))
```

Во многих языках программирования (например, в PHP, Perl и др.) каждая инструкция должна завершаться точкой с запятой. В языке Python в конце инструкции также можно поставить точку с запятой, но это не обязательно. Более того, в отличие от языка JavaScript, где рекомендуется завершать инструкции точкой с запятой, в языке Python точку с запятой ставить *не рекомендуется*. Концом инструкции является конец строки. Тем не менее, если необходимо разместить несколько инструкций на одной строке, точку с запятой *следует указать* (листинг 1.7).

Листинг 1.7. Несколько инструкций на одной строке

```
>>> x = 5; y = 10; z = x + y # Три инструкции на одной строке
>>> print(z)
15
```

Еще одной отличительной особенностью языка Python является отсутствие ограничительных символов для выделения инструкций внутри блока. Например, в языке PHP инструкции 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 1.8).

Листинг 1.8. Выделение инструкций внутри блока

```
i = 1
while i < 11:
    print(i)
    i += 1
print("Конец программы")
```

Обратите внимание, что перед всеми инструкциями внутри блока расположено одинаковое количество пробелов. Именно так в языке Python выделяются *блоки*. Инструкции, перед которыми расположено одинаковое количество пробелов, являются *телом блока*. В нашем примере две инструкции выполняются десять раз. Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В нашем случае это функция `print()`, которая выводит строку "Конец программы". Если количество пробелов внутри блока окажется разным, то интерпретатор выведет сообщение о фатальной ошибке, и программа будет остановлена. Так язык Python приучает программистов писать красивый и понятный код.

> ***Примечание***
>
> В языке Python принято использовать четыре пробела для выделения инструкций внутри блока.

Если блок состоит из одной инструкции, то допустимо разместить ее 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 инструкция является слишком длинной, то ее можно перенести на следующую строку, например так:

- 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. Этот способ лучше, т. к. внутри круглых скобок можно разместить любое выражение. Пример:

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

- определение списка и словаря можно разместить на нескольких строках, т. к. при этом 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 программы, интерпретатор полностью их игнорирует. Внутри комментария может располагаться любой текст, включая инструкции, которые выполнять не следует.

> ***Совет***
>
> 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 инструкции. Например, в следующем примере комментарий расположен после инструкции, предписывающей вывести надпись "Привет, мир!":

```
print("Привет, мир!") # Выводим надпись с помощью функции print()
```

Если же символ комментария разместить перед инструкцией, то она не будет выполнена:

```
# print("Привет, мир!") Эта инструкция выполнена 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):

```
"""
Эта инструкция выполнена не будет
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 инструкции внутри утроенных кавычек выполнены не будут, поскольку интерпретатор сочтет их простым текстом. Такие строки являются строками документирования, а не комментариями.

1.5. Скрытые возможности IDLE

Поскольку в процессе изучения материала этой книги в качестве редактора мы будем использовать IDLE, 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 (при условии, что инструкция возвращает значение), 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 функции `print()`. В окне **Python Shell** это делать не обязательно. Например, мы можем просто ввести строку и нажать клавишу <Enter> для получения результата:

```
>>> "Привет, мир!"
'Привет, мир!'
>>>
```

Обратите внимание на то, что строки выводятся в апострофах. Этого не произойдет, если выводить строку с помощью функции `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 последней инструкции 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
85.0
>>>
```

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 инструкции. Просто вводите инструкцию дальше, а список закроется. Такой же список будет автоматически появляться (с некоторой задержкой) при обращении к атрибутам объекта или модуля 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 введенную инструкцию ее приходится набирать заново. Можно, конечно, скопировать инструкцию, 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 инструкцию, ее предварительно необходимо выделить. Редактор IDLE избавляет нас от лишних действий и предоставляет комбинацию клавиш <Alt>+<N> для вставки первой введенной инструкции, а также комбинацию <Alt>+<P> для вставки последней инструкции. Каждое последующее нажатие этих клавиш будет вставлять следующую (или предыдущую) инструкцию. Для еще более быстрого повторного ввода инструкции следует предварительно ввести ее первые буквы. В этом случае перебираться будут только инструкции, начинающиеся с этих букв.

1.6. Вывод результатов работы программы

Вывести результаты работы программы можно с помощью функции `print()`. Функция имеет следующий формат:

Функция `print()` преобразует объект в строку и посылает ее в стандартный вывод `stdout`. С помощью параметра `file` можно перенаправить вывод в другое место — например, в файл. При этом, если параметр `flush` имеет значение `False`, выводимые значения будут принудительно записаны в файл. Перенаправление вывода мы подробно рассмотрим при изучении файлов.

После вывода строки автоматически добавляется символ перевода строки:

```
print("Строка 1")
print("Строка 2")
```

Результат:

```
Строка 1
Строка 2
```

Если необходимо вывести результат на той же строке, то в функции `print()` данные указываются через запятую в первом параметре:

```
print("Строка 1", "Строка 2")
```

Результат:

```
Строка 1 Строка 2
```

Как видно из примера, между выводимыми строками автоматически вставляется пробел. С помощью параметра `sep` можно указать другой символ. Например, выведем строки без пробела между ними:

```
print("Строка1", "Строка2", sep="")
```

Результат:

```
Строка 1Строка 2
```

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:

```
print("Строка 1", "Строка 2", end=" ")
print("Строка 3")
# Выведет: Строка 1 Строка 2 Строка 3
```

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

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

Результат выполнения:

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

Здесь мы использовали цикл `for`, который позволяет последовательно перебирать элементы. На каждой итерации цикла переменной `n` присваивается новое число, которое мы выводим с помощью функции `print()`, расположенной на следующей строке.

Обратите внимание, что перед функцией мы добавили четыре пробела. Как уже отмечалось ранее, таким образом в языке Python выделяются блоки. При этом инструкции, перед которыми расположено одинаковое количество пробелов, представляют собой тело цикла. Все эти инструкции выполняются определенное количество раз. Концом блока является инструкция, перед которой расположено меньшее количество пробелов. В нашем случае это функция `print()` без параметров, которая вставляет символ перевода 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
```

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

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

В первой строке с помощью оператора `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 в Python 3 предназначена функция `input()`, которая получает данные со стандартного ввода `stdin`. Функция имеет следующий формат:

```
[<Значение> = ] input([<Сообщение>])
```

Для примера переделаем нашу первую программу так, чтобы она здоровалась не со всем миром, а только с нами (листинг 1.9).

Листинг 1.9. Пример использования функции `input()`

```
# -*- coding: utf-8 -*-
name = input("Введите ваше имя: ")
print("Привет,", name)
input("Нажмите <Enter> для закрытия окна")
```

Вводим код и сохраняем файл, например, под именем test2.py, а затем запускаем программу на выполнение с помощью двойного щелчка на значке файла. Откроется черное окно, в котором вы увидите 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 `input()`. В этом случае окно не закроется, пока не будет нажата клавиша <Enter>.

При использовании функции `input()` следует учитывать, что при достижении конца файла или при нажатии комбинации клавиш <Ctrl>+<Z>, а затем <Enter> генерируется исключение `EOFError`. Если не предусмотреть обработку исключения, то программа аварийно завершится. Обработать исключение можно следующим образом:

```
try:
    s = input("Введите данные: ")
    print(s)
except EOFError:
    print("Обработали исключение EOFError")
```

Если внутри блока `try` возникнет исключение `EOFError`, то управление будет передано в блок `except`. После исполнения инструкций в блоке `except` программа нормально продолжит работу.

В Python 2 для ввода данных применялись две функции: `raw_input()` и `input()`. Функция `raw_input()` просто возвращала введенные данные, а функция `input()` предварительно обрабатывала данные с помощью функции `eval()` и затем возвращала результат ее выполнения. В Python 3 функция `raw_input()` была переименована в `input()`, а прежняя функция `input()` — удалена. Чтобы вернуться к поведению функции `input()` в Python 2, необходимо передать значение в функцию `eval()` явным образом:

```
# -*- coding: utf-8 -*-
result = eval(input("Введите инструкцию: ")) # Вводим:  2 + 2
print("Результат:", result)                  # Выведет: 4
input()
```

> ***Внимание!***
>
> **Функция `eval()` выполнит любую введенную инструкцию. Никогда не используйте этот код, 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 test3.py в папке C:\book. Содержимое файла приведено в листинге 1.10.

Листинг 1.10. Получение данных из командной строки

```
# -*- coding: utf-8 -*-
import sys
arr = sys.argv[:]
for n in arr:
    print(n)
```

Теперь запустим программу на выполнение из командной строки и передадим ей данные. Откроем командную строку, для чего в меню **Пуск** выберем пункт **Выполнить**. В появив-

шемся окне наберем команду `cmd` и нажмем кнопку **OK**. Откроется черное окно с приглашением для ввода команд. Перейдем в папку C:\book, набрав команду:

```
cd C:\book
```

В командной строке должно появиться приглашение:

```
C:\book>
```

Для запуска нашей программы вводим команду:

```
C:\Python34\python.exe test3.py -uNik -p123
```

В этой команде мы передаем имя файла (`test3.py`) и некоторые данные (`-uNik` и `-p123`). Результат выполнения программы будет выглядеть так:

```
test3.py
-uNik
-p123
```

1.8. Доступ к документации

При установке Python на компьютер помимо собственно интерпретатора копируется документация по этому языку в формате CHM. Чтобы отобразить документацию, в меню **Пуск** выбираем пункт **Программы (Все программы) | Python 3.4 | Python 3.4 Manuals**.

Если в меню **Пуск | Программы (Все программы) | Python 3.4** выбрать пункт **Python 3.4 Docs Server (pydoc - 32 bit)** или **Python 3.4 Docs Server (pydoc - 64 bit)**, запустится сервер документов pydoc (рис. 1.11). Он представляет собой программу, написанную на самом Python, выступающую в роли Web-сервера и выводящую результаты работы в Web-браузере.

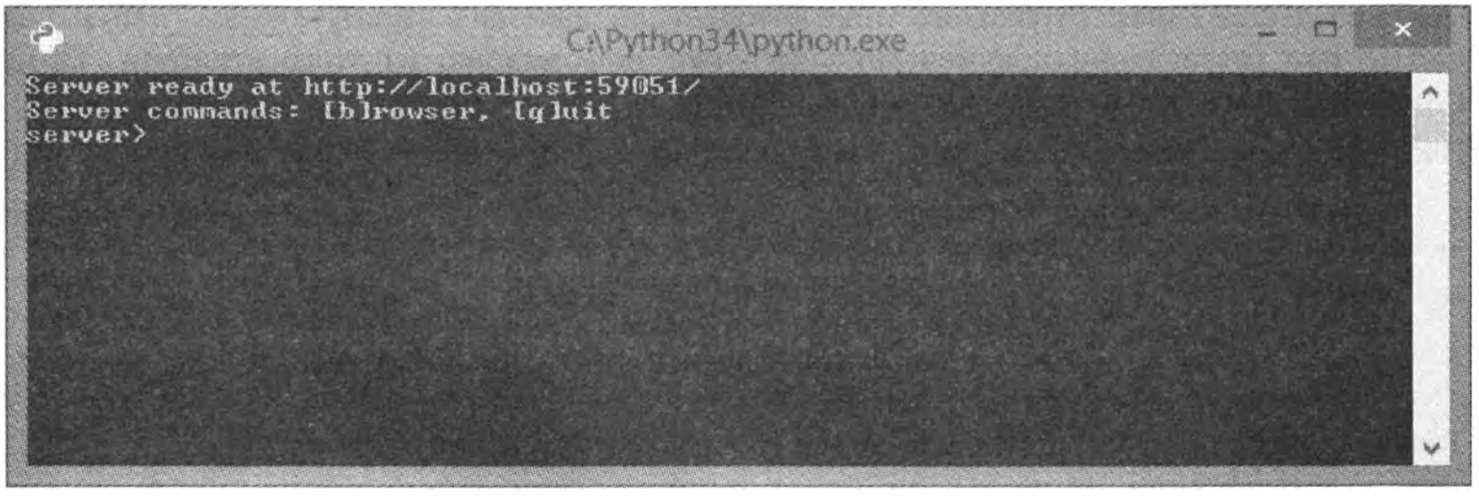


Рис. 1.11. Окно pydoc

Сразу после запуска pydoc откроется Web-браузер, в котором будет выведен список всех стандартных модулей, поставляющихся в составе Python. Нажав на название модуля, представляющее собой гиперссылку, мы откроем страницу с описанием всех классов, функций и констант, объявленных в этом модуле.

Чтобы завершить работу pydoc, следует переключиться в его окно (см. рис. 1.11), ввести в нем команду `q` (от quit — выйти) и нажать клавишу <Enter> — окно при этом автомати-

чески закроется. А введенная там команда b (от browser — браузер) повторно выведет страницу со списком модулей.

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

```
>>> help(input)
```

Результат выполнения:

```
Help on built-in function input in module builtins:

input(...)
    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 `builtins`, содержащий определения всех встроенных функций и классов, а затем выведем документацию по этому модулю:

```
>>> import builtins
>>> help(builtins)
```

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. Такие строки называются *строками документирования*.

В качестве примера создадим два файла в одной папке. Содержимое файла **test4.py**:

```
# -*- coding: utf-8 -*-
""" Это описание нашего модуля """
def func():
    """ Это описание функции"""
    pass
```

Теперь подключим этот модуль и выведем содержимое строк документирования:

```
# -*- coding: utf-8 -*-
import test4                        # Подключаем файл test4.py
help(test4)
```

Результат выполнения:

```
Help on module test4:

NAME
    test4 - Это описание нашего модуля
```

```
FUNCTIONS
    func()
        Это описание функции

FILE
    c:\users\dronov_va\documents\работа\python самое необходимое\материалы\test4.py
```

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

```
# -*- coding: utf-8 -*-
import test4                        # Подключаем файл test4.py
print(test4.__doc__)
print(test4.func.__doc__)
```

Результат выполнения:

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

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

```
>>> print(input.__doc__)
```

Результат выполнения:

```
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: utf-8 -*-
import test4                        # Подключаем файл test4.py
print( dir(test4) )
```

Результат выполнения:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'func']
```

Теперь получим список всех встроенных идентификаторов:

```
>>> import builtins
>>> print( dir(builtins) )
```

Функция `dir()` **может не принимать параметров вообще. В этом случае возвращается список идентификаторов текущего модуля:**

```
# -*- coding: utf-8 -*-
import test4                        # Подключаем файл test4.py
print( dir() )
```

Результат выполнения:

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* на объект (адрес объекта в памяти компьютера). 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, поскольку идентификаторам с таким символом определено специальное назначение. Например, имена, начинающиеся с символа подчеркивания, не импортируются из модуля с помощью инструкции `from module import *`, а имена, включающие по два символа подчеркивания — в начале и в 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
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

Помимо ключевых слов, следует избегать совпадений со встроенными идентификаторами. Дело в том, что, 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(abs)
Help on built-in function abs in module builtins:
```

```
abs(...)
    abs(number) -> number

    Return the absolute value of the argument.

>>> help = 10
>>> help
10
>>> help(abs)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    help(abs)
TypeError: 'int' object is not callable
```

В этом примере мы с помощью встроенной функции `help()` получаем справку по функции `abs()`. 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. Очень часто подобная ошибка возникает при попытке назвать переменную, в которой предполагается хранение строки, именем `str`. Вроде бы логично, но `str` является часто используемым встроенным идентификатором и после такого переопределения поведение программы становится непредсказуемым. В редакторе IDLE встроенные идентификаторы подсвечиваются фиолетовым цветом. Обращайте внимание на цвет переменной — он должен быть черным. Если вы заметили, что переменная подсвечена, то название переменной следует обязательно изменить. Получить полный список встроенных идентификаторов позволяет код, приведенный в листинге 2.3.

Листинг 2.3. Получение списка встроенных идентификаторов

```
>>> import builtins
>>> dir(builtins)
```

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

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 Python 3 объекты могут иметь следующие типы данных:

- `bool` — **логический тип данных. Может содержать значения** `True` **или** `False`**, которые ведут себя как числа** `1` **и** `0` **соответственно:**

```
>>> type(True), type(False)
(<class 'bool'>, <class 'bool'>)
>>> int(True), int(False)
(1, 0)
```

- `NoneType` — **объект со значением** `None` **(обозначает отсутствие значения):**

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

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

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

- `int` — **целые числа. Размер числа ограничен лишь объемом оперативной памяти:**

```
>>> type(2147483647), type(999999999999999999999999)
(<class 'int'>, <class 'int'>)
```

- `float` — **вещественные числа:**

```
>>> type(5.1), type(8.5e-3)
(<class 'float'>, <class 'float'>)
```

- `complex` — **комплексные числа:**

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

- `str` — **Unicode-строки:**

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

- `bytes` — **неизменяемая последовательность байтов:**

```
>>> type(bytes("Строка", "utf-8"))
<class 'bytes'>
```

- `bytearray` — **изменяемая последовательность байтов:**

```
>>> type(bytearray("Строка", "utf-8"))
<class 'bytearray'>
```

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

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

- `tuple` — **кортежи:**

```
>>> type( (1, 2, 3) )
<class 'tuple'>
```

- `range` — **диапазоны:**

```
>>> type( range(1, 10) )
<class 'range'>
```

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

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

- `set` — **множества (коллекции уникальных объектов):**

```
>>> type( {"a", "b", "c"} )
<class 'set'>
```

- `frozenset` — **неизменяемые множества:**

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

- `ellipsis` — **обозначается в виде трех точек или слова** `Ellipsis`. **Тип** `ellipsis` **используется в расширенном синтаксисе получения среза:**

```
>>> type(...), ..., ... is Ellipsis
(<class 'ellipsis'>, Ellipsis, True)
>>> class C():
        def __getitem__(self, obj): return obj

>>> c = C()
>>> c[..., 1:5, 0:9:1, 0]
(Ellipsis, slice(1, 5, None), slice(0, 9, 1), 0)
```

- `function` — **функции:**

```
>>> def func(): pass

>>> type(func)
<class 'function'>
```

- `module` — **модули:**

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

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

```
>>> class C: pass

>>> type(C)
<class 'type'>
>>> type(type(""))
<class 'type'>
```

Основные типы данных делятся на *изменяемые* **и** *неизменяемые*. **К изменяемым типам относятся списки, словари и тип** `bytearray`. **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, кортежи, диапазоны и тип `bytes`. 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, кортежи, диапазоны, типы `bytes` и `bytearray`, а к отображениям — словари.

Последовательности и отображения поддерживают механизм итераторов, позволяющий произвести обход всех элементов с помощью метода `__next__()` или функции `next()`. Например, вывести элементы списка можно так:

```
>>> arr = [1, 2]
>>> i = iter(arr)
>>> i.__next__()                  # Метод __next__()
1
>>> next(i)                       # Функция next()
2
```

Если используется словарь, то на каждой итерации возвращается ключ:

```
>>> d = {"x": 1, "y": 2}
>>> i = iter(d)
>>> i.__next__()                  # Возвращается ключ
'y'
>>> d[i.__next__()]               # Получаем значение по ключу
1
```

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 так же. Для примера вставим тире после каждой буквы:

```
>>> for i in "Строка":
        print(i + " -", end=" ")
```

Результат:

```
С - т - р - о - к - а -
```

Пример перебора элементов словаря:

```
>>> d = {"x": 1, "y": 2}
>>> for key in d:
        print( d[key] )
```

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

2.3. Присваивание значения переменным

В языке Python используется *динамическая типизация*. Это означает, что при присваивании переменной значения 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
>>> x, y
(10, 10)
```

После присваивания значения в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при *групповом присваивании*. Групповое присваивание можно использовать для чисел, строк и кортежей, но для изменяемых 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 повышения 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)
304
```

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, кортежи, диапазоны, типы `bytes` и `bytearray`. 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)
>>> [x, y, z] = (1, 2, 3)        # Список слева, кортеж справа
>>> x, y, z
(1, 2, 3)
```

Обратите внимание на то, что 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#130>", line 1, in <module>
    x, y, z = (1, 2, 3, 4)
ValueError: too many values to unpack (expected 3)
```

Python 3 при несоответствии количества элементов справа и слева от оператора = позволяет сохранить в переменной список, состоящий из лишних элементов. Для этого перед именем переменной указывается звездочка (*). Пример:

```
>>> x, y, *z = (1, 2, 3, 4)
>>> x, y, z
(1, 2, [3, 4])
>>> x, *y, z = (1, 2, 3, 4)
>>> x, y, z
(1, [2, 3], 4)
>>> *x, y, z = (1, 2, 3, 4)
>>> x, y, z
([1, 2], 3, 4)
>>> x, y, *z = (1, 2, 3)
>>> x, y, z
(1, 2, [3])
>>> x, y, *z = (1, 2)
>>> x, y, z
(1, 2, [])
```

Как видно из примера, переменная, перед которой указана звездочка, всегда содержит список. Если для этой переменной не хватило значений, то ей присваивается пустой список. Следует помнить, что звездочку можно указать только перед одной переменной. В противном случае возникнет неоднозначность и интерпретатор выведет сообщение об ошибке:

```
>>> *x, y, *z = (1, 2, 3, 4)
SyntaxError: two starred expressions in assignment
```

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
```

Определить, на какой тип данных ссылается переменная, позволяет функция `type(<Имя переменной>)`:

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

Проверить тип данных, хранящихся в 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 последним можно производить операции, предназначенные лишь 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#0>", line 1, in <module>
    2 + "25"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Для преобразования типов данных предназначены следующие функции:

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

```
>>> bool(0), bool(1), bool(""), bool("Строка"), bool([1, 2]), bool([])
(False, True, False, True, True, False)
```

- `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("0o71", 8), int("A", 16)
(71, 57, 57, 10)
```

Если преобразование невозможно, то генерируется исключение:

```
>>> int("71s")
Traceback (most recent call last):
  File "<pyshell#9>", 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, 7.1)
>>> float("Infinity"), float("-inf")
(inf, -inf)
>>> float("Infinity") + float("-inf")
nan
```

- `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}")
>>> str( bytes("строка", "utf-8") )
"b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0\\xba\\xd0
\\xb0'"
>>> str( bytearray("строка", "utf-8") )
"bytearray(b'\\xd1\\x81\\xd1\\x82\\xd1\\x80\\xd0\\xbe\\xd0
\\xba\\xd0\\xb0')"
```

- `str(<Объект>[, <Кодировка>[, <Обработка ошибок>]])` — **преобразует объект типа** `bytes` **или** `bytearray` **в строку. В третьем параметре можно задать значение** `"strict"` **(при ошибке возбуждается исключение** `UnicodeDecodeError` **— значение по умолчанию),** `"replace"` **(неизвестный символ заменяется символом, имеющим код** `\uFFFD`**) или** `"ignore"` **(неизвестные символы игнорируются). Примеры:**

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    str(obj1, "ascii", "strict")
UnicodeDecodeError: 'ascii' codec can't decode byte
0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
'1'
```

- `bytes(<Строка>, <Кодировка>[, <Обработка ошибок>])` — **преобразует строку в объект типа** `bytes`**. В третьем параметре могут быть указаны значения** `"strict"` **(значение по умолчанию),** `"replace"` **или** `"ignore"`**.**

Примеры:

```
>>> bytes("строка", "cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка123", "ascii", "ignore")
b'123'
```

- `bytes(<Последовательность>)` — **преобразует последовательность целых чисел от 0 до 255 в объект типа** `bytes`. **Если число не попадает в диапазон, то возбуждается исключение** `ValueError`. **Пример:**

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

- `bytearray(<Строка>, <Кодировка>[, <Обработка ошибок>])` — **преобразует строку в объект типа** `bytearray`. **В третьем параметре могут быть указаны значения** `"strict"` **(значение по умолчанию),** `"replace"` **или** `"ignore"`. **Пример:**

```
>>> bytearray("строка", "cp1251")
bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
```

- `bytearray(<Последовательность>)` — **преобразует последовательность целых чисел от 0 до 255 в объект типа** `bytearray`. **Если число не попадает в диапазон, то возбуждается исключение** `ValueError`. **Пример:**

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

- `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 `input()`. **Воспользуемся этой функцией для получения чисел от пользователя (листинг 2.4).**

Листинг 2.4. Получение данных от пользователя

```
# -*- coding: utf-8 -*-
x = input("x = ")              # Вводим 5
y = input("y = ")              # Вводим 12
print(x + y)
input()
```

Результатом выполнения этого скрипта будет не число, а строка `512`. **Таким образом, следует запомнить, что функция** `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: utf-8 -*-
x = int(input("x = "))         # Вводим 5
y = int(input("y = "))         # Вводим 12
print(x + y)
input()
```

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 разберемся 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#1>", line 1, in <module>
    del x; x
NameError: name 'x' is not defined
```

Пример удаления нескольких переменных:

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

ГЛАВА 3

Операторы

Операторы позволяют произвести с данными определенные действия. Например, операторы присваивания служат для сохранения данных в переменной, математические операторы позволяют выполнить арифметические вычисления, а оператор конкатенации строк служит для соединения двух строк в одну. Рассмотрим операторы, доступные в Python 3, подробно.

3.1. Математические операторы

Производить операции над числами позволяют следующие операторы:

- `+` — сложение:

```
>>> 10 + 5                  # Целые числа
15
>>> 12.4 + 5.2              # Вещественные числа
17.6
>>> 10 + 12.4               # Целые и вещественные числа
22.4
```

- `-` — вычитание:

```
>>> 10 - 5                  # Целые числа
5
>>> 12.4 - 5.2              # Вещественные числа
7.2
>>> 12 - 5.2                # Целые и вещественные числа
6.8
```

- `*` — умножение:

```
>>> 10 * 5                  # Целые числа
50
>>> 12.4 * 5.2              # Вещественные числа
64.48
>>> 10 * 5.2                # Целые и вещественные числа
52.0
```

- `/` — деление. Результатом деления всегда является вещественное число, даже если производится деление целых чисел. Обратите внимание на эту особенность, если вы раньше

программировали на Python 2. В Python 2 при делении целых чисел остаток отбрасывался и возвращалось целое число, в Python 3 поведение оператора изменилось. Примеры:

```
>>> 10 / 5                # Деление целых чисел без остатка
2.0
>>> 10 / 3                # Деление целых чисел с остатком
3.3333333333333335
>>> 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 вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое число и вещественное, то целое число будет автоматически преобразовано в вещественное число, а затем произведена операция над вещественными числами. Результатом этой операции станет 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

Побитовые операторы предназначены для манипуляции отдельными битами. Язык Python поддерживает следующие побитовые операторы:

- ~ — двоичная инверсия. Значение каждого бита заменяется на противоположное:

```
>>> x = 100                  # 01100100
>>> x = ~x                   # 10011011
```

- & — двоичное И:

```
>>> x = 100                  # 01100100
>>> y = 75                   # 01001011
>>> z = x & y                # 01000000
>>> "{0:b} & {1:b} = {2:b}".format(x, y, z)
'1100100 & 1001011 = 1000000'
```

- | — двоичное ИЛИ:

```
>>> x = 100                    # 01100100
>>> y = 75                     # 01001011
>>> z = x | y                  # 01101111
>>> "{0:b} | {1:b} = {2:b}".format(x, y, z)
'1100100 | 1001011 = 1101111'
```

- ^ — двоичное исключающее ИЛИ:

```
>>> x = 100                    # 01100100
>>> y = 250                    # 11111010
>>> z = x ^ y                  # 10011110
>>> "{0:b} ^ {1:b} = {2:b}".format(x, y, z)
'1100100 ^ 11111010 = 10011110'
```

- << — сдвиг влево — сдвигает двоичное представление числа влево на один или более разрядов и заполняет разряды справа нулями:

```
>>> x = 100                    # 01100100
>>> y = x << 1                 # 11001000
>>> z = y << 1                 # 10010000
>>> k = z << 2                 # 01000000
```

- >> — сдвиг вправо — сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, если число положительное:

```
>>> x = 100                    # 01100100
>>> y = x >> 1                 # 00110010
>>> z = y >> 1                 # 00011001
>>> k = z >> 2                 # 00000110
```

Если число отрицательное, то разряды слева заполняются единицами:

```
>>> x = -127                   # 10000001
>>> y = x >> 1                 # 11000000
>>> z = y >> 2                 # 11110000
>>> k = z << 1                 # 11100000
>>> m = k >> 1                 # 11110000
```

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)
```

- `not in` — **проверка на невхождение. Если элемент не входит в последовательность, возвращается** `True`:

```
>>> "Строка" not in "Строка для поиска"   # Строки
False
>>> "Строка2" not in "Строка для поиска"  # Строки
True
>>> 2 not in [1, 2, 3], 4 not in [1, 2, 3]    # Списки
(False, True)
>>> 2 not in (1, 2, 3), 6 not in (1, 2, 3)    # Кортежи
(False, True)
```

3.4. Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной. Перечислим операторы присваивания, доступные в языке Python:

- `=` — **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
3.3333333333333335
>>> 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 далее 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
20.0
```

С помощью скобок можно изменить последовательность вычисления выражения:

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

Теперь порядок вычислений станет иным:

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
22.5
```

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 выполнить его. Логические выражения возвращают только два значения: `True` (истина) или `False` (ложь), которые ведут себя как целые числа `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` возвращает 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)
```

- `!=` — **не равно:**

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

- `<` — **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)
```

Оператор `in` **можно также использовать для проверки существования ключа словаря:**

```
>>> "x" in {"x": 1, "y": 2}, "z" in {"x": 1, "y": 2}
(True, False)
```

- `not in` — **проверка на невхождение в последовательность:**

```
>>> "Строка" not in "Строка для поиска" # Строки
False
>>> 2 not in [1, 2, 3], 4 not in [1, 2, 3]  # Списки
(False, True)
```

```
>>> 2 not in (1, 2, 3), 4 not in (1, 2, 3)  # Кортежи
(False, True)
```

- `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 повышения 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)
```

- `is not` — проверяет, ссылаются ли две переменные на разные объекты. Если это так, возвращается значение `True`:

```
>>> x = y = [1, 2]
>>> x is not y
False
>>> x = [1, 2]; y = [1, 2]
>>> x is not y
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 так как перед выражением стоит оператор `not`, выражение вернет `False`. Круглые скобки можно не указывать, поскольку оператор `not` имеет более низкий приоритет выполнения, чем операторы сравнения.

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

```
>>> 1 < 5 and 2 < 5                  # True and True == True
True
```

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

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

Следующее 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 выполнить 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 четырьмя пробелами). Концом блока является инструкция, перед которой 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. Тем не менее, круглые скобки следует использовать только при необходимости разместить условие на нескольких строках.

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

Листинг 4.1. Проверка числа на четность

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0:
    print(x, " - четное число")
else:
    print(x, " - нечетное число")
input()
```

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

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0: print(x, " - четное число")
else: print(x, " - нечетное число")
input()
```

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: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0: print(x, end=" "); print("- четное число")
else: print(x, end=" "); print("- нечетное число")
input()
```

> ***Совет***
>
> Знайте, что так сделать можно, но никогда на практике не пользуйтесь этим способом, поскольку 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 следующий код читается намного проще, чем предыдущий:

```
# -*- coding: utf-8 -*-
x = int(input("Введите число: "))
if x % 2 == 0:
    print(x, end=" ")
    print("- четное число")
```

```
else:
    print(x, end=" ")
    print("- нечетное число")
input()
```

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

Листинг 4.2. Проверка нескольких условий

```
# -*- coding: utf-8 -*-
print("""Какой операционной системой вы пользуетесь?
1 — Windows 8
2 — Windows 7
3 — Windows Vista
4 — Windows XP
5 — Другая""")
os = input("Введите число, соответствующее ответу: ")
if   os == "1":
    print("Вы выбрали: Windows 8")
elif os == "2":
    print("Вы выбрали: Windows 7")
elif os == "3":
    print("Вы выбрали: Windows Vista")
elif os == "4":
    print("Вы выбрали: Windows XP")
elif os == "5":
    print("Вы выбрали: другая")
elif not os:
    print("Вы не ввели число")
else:
    print("Мы не смогли определить вашу операционную систему")
input()
```

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. Поскольку 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: utf-8 -*-
print("""Какой операционной системой вы пользуетесь?
1 — Windows 8
2 — Windows 7
3 — Windows Vista
4 — Windows XP
5 — Другая""")
os = input("Введите число, соответствующее ответу: ")
if os != "":
    if   os == "1":
        print("Вы выбрали: Windows 8")
    elif os == "2":
        print("Вы выбрали: Windows 7")
    elif os == "3":
        print("Вы выбрали: Windows Vista")
    elif os == "4":
        print("Вы выбрали: Windows XP")
    elif os == "5":
        print("Вы выбрали: другая")
    else:
        print("Мы не смогли определить вашу операционную систему")
else:
    print("Вы не ввели число")
input()
```

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'
>>> s = "Yes" if 11 % 2 == 0 else "No"
>>> s
'No'
```

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 range(1, 101): print(x)
```

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

Цикл `for` применяется для перебора элементов последовательности и имеет такой формат:

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

Здесь присутствуют следующие конструкции:

- `<Последовательность>` — объект, поддерживающий механизм итерации. Например: строка, список, кортеж, диапазон, словарь и др.;
- `<Текущий элемент>` — на каждой итерации через этот параметр доступен текущий элемент последовательности или ключ словаря;
- `<Инструкции 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`. Этот 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, end=" ")
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 объект `dict_keys`, который содержит все ключи словаря. Во втором способе 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}
>>> arr.keys()
dict_keys(['y', 'x', 'z'])
>>> for key in arr.keys():      # Использование метода keys()
        print(key, arr[key])

y 2
x 1
z 3
>>> for key in arr:             # Словари также поддерживают итерации
        print(key, arr[key])

y 2
x 1
z 3
```

Обратите внимание на то, что элементы словаря выводятся в произвольном порядке, а не в порядке, в котором они были указаны при создании объекта. Чтобы вывести элементы в алфавитном порядке, следует отсортировать ключи с помощью функции `sorted()`:

```
>>> arr = {"x": 1, "y": 2, "z": 3}
>>> for key in sorted(arr):
        print(key, arr[key])

x 1
y 2
z 3
```

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)

1 2
3 4
```

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

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

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 индексов. 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 возвращаемые значения. Если параметр <Шаг> не указан, то используется значение 1. Функция возвращает *диапазон* — особый объект, поддерживающий итерационный протокол. С помощью диапазона внутри цикла for можно получить значение текущего элемента. 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 диапазон значений от 0 до len(arr) – 1. На каждой итерации цикла через переменную i доступен текущий элемент из диапазона индексов. Чтобы получить доступ к элементу списка, указываем индекс внутри квадратных скобок. Умножаем каждый элемент списка на 2, а затем выводим результат с помощью функции print().

Рассмотрим несколько примеров использования функции range(). Выведем числа от 1 до 100:

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

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

```
for i in range(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)
```

В Python 2 функция range() возвращала список чисел. В Python 3 поведение функции изменилось — теперь она возвращает диапазон. Чтобы получить список чисел, следует передать диапазон, возвращенный функцией range(), в функцию list() (листинг 4.9).

Листинг 4.9. Создание списка чисел на основе диапазона

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

```
>>> obj[0], obj[1], obj[2]      # Доступ по индексу
(0, 1, 2)
>>> obj[0:2]                    # Получение среза
range(0, 2)
>>> i = iter(obj)
>>> next(i), next(i), next(i)   # Доступ с помощью итераторов
(0, 1, 2)
>>> list(obj)                   # Преобразование диапазона в список
[0, 1, 2]
>>> 1 in obj, 7 in obj          # Проверка вхождения значения
(True, False)
```

Диапазон поддерживает два полезных метода:

- `index(<Значение>)` — **возвращает индекс элемента, имеющего указанное значение. Если значение не входит в диапазон, возбуждается исключение** `ValueError`. **Пример:**

```
>>> obj = range(1, 5)
>>> obj.index(1), obj.index(4)
(0, 3)
>>> obj.index(5)
... Фрагмент опущен ...
ValueError: 5 is not in range
```

- `count(<Значение>)` — **возвращает количество элементов с указанным значением. Если элемент не входит в диапазон, возвращается значение** `0`. **Пример:**

```
>>> obj = range(1, 5)
>>> obj.count(1), obj.count(10)
(1, 0)
```

Функция `enumerate(<Объект>[, start=0])` **на каждой итерации цикла** `for` **возвращает кортеж из индекса и значения текущего элемента. С помощью необязательного параметра** `start` **можно задать начальное значение индекса. 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()` **не создает список, а возвращает итератор. С помощью функции** `next()` **можно обойти всю последовательность. Когда перебор будет закончен, возбуждается исключение** `StopIteration`:

```
>>> arr = [1, 2]
>>> obj = enumerate(arr, start=2)
>>> next(obj)
(2, 1)
```

```
>>> next(obj)
(3, 2)
>>> next(obj)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(obj)
StopIteration
```

Кстати, цикл `for` при работе активно использует итераторы, но делает это незаметно для нас.

4.5. Цикл *while*

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

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

Последовательность работы цикла `while`:

1. **Переменной-счетчику присваивается начальное значение.**
2. **Проверяется условие и, если оно истинно, выполняются инструкции 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`. Этот 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)             # <Инструкции>
    i += 1               # <Приращение>
```

> ***Внимание!***
>
> **Если `<Приращение>` не указано, цикл будет бесконечным. Чтобы прервать бесконечный цикл, следует нажать комбинацию клавиш <Ctrl>+<C>. В результате генерируется исключение `KeyboardInterrupt`, и выполнение программы останавливается. 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 инструкций 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 range(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 станут выполняться бесконечно. Однако использование оператора `break` прерывает выполнение цикла, как только будет напечатано 100 строк.

> ***Внимание!***
>
> Оператор `break` прерывает выполнение цикла, а не программы, т. е. далее будет выполнена инструкция, следующая 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: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = input("Введите число: ")
    if x == "stop":
        break                       # Выход из цикла
    x = int(x)                      # Преобразуем строку в число
    summa += x
print("Сумма чисел равна:", summa)
input()
```

Процесс ввода трех чисел и получения суммы выглядит так (значения, введенные пользователем, здесь выделены полужирным шрифтом):

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

ГЛАВА 5

Числа

Язык Python 3 поддерживает следующие числовые типы:

- `int` — целые числа. Размер числа ограничен лишь объемом оперативной памяти;
- `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 вещественные числа и самый сложный тип — комплексные числа. Таким образом, если в операции участвуют целое число и вещественное, 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 двоичной, восьмеричной или шестнадцатеричной форме. Такие числа будут автоматически преобразованы в десятичные целые числа.

- Двоичные числа начинаются с комбинации символов `0b` (или `0B`) и содержат цифры от `0` или `1`:

```
>>> 0b11111111, 0b101101
(255, 45)
```

- Восьмеричные числа начинаются с нуля и следующей за ним латинской буквы `o` (регистр не имеет значения) и содержат цифры от `0` до `7`:

```
>>> 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)
```

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

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 другое значение. 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')
```

Кроме того, можно использовать дроби, поддержка которых содержится в модуле `fractions`. При создании дроби можно как указать два числа: числитель и знаменатель, так и одно число или строку, содержащую число, которое будет преобразовано в дробь.

Для примера создадим несколько дробей. Вот так формируется дробь $^4/_5$:

```
>>> from fractions import Fraction
>>> Fraction(4, 5)
Fraction(4, 5)
```

А вот так — дробь $^1/_2$, причем можно сделать это тремя способами:

```
>>> Fraction(1, 2)
Fraction(1, 2)
>>> Fraction("0.5")
Fraction(1, 2)
>>> Fraction(0.5)
Fraction(1, 2)
```

Над дробями можно производить арифметические операции, как и над обычными числами:

```
>>> Fraction(9, 5) - Fraction(2, 3)
Fraction(17, 15)
>>> Fraction("0.3") - Fraction("0.1") - Fraction("0.1") - Fraction("0.1")
Fraction(0, 1)
>>> float(Fraction(0, 1))
0.0
```

Комплексные числа записываются в формате:

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

Здесь буква `J` может стоять в любом регистре. Примеры комплексных чисел:

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

Подробное рассмотрение модулей `decimal` и `fractions`, а также комплексных чисел выходит за рамки нашей книги. За подробной информацией обращайтесь к соответствующей документации.

5.1. Встроенные функции и методы 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 преобразуемого числа (значение по умолчанию `10`). Пример:

```
>>> int(7.5), int("71", 10), int("0o71", 8), int("0xA", 16)
(7, 71, 57, 10)
>>> int(), int("0b11111111", 2)
(0, 255)
```

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

```
>>> float(7), float("7.1"), float("12.")
(7.0, 7.1, 12.0)
>>> float("inf"), float("-Infinity"), float("nan")
(inf, -inf, nan)
>>> float()
0.0
```

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

```
>>> bin(255), bin(1), bin(-45)
('0b11111111', '0b1', '-0b101101')
```

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

```
>>> oct(7), oct(8), oct(64)
('0o7', '0o10', '0o100')
```

- `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>])` — для чисел с дробной частью меньше `0.5` возвращает число, округленное до ближайшего меньшего целого, а для чисел с дробной частью больше `0.5` возвращает число, округленное до ближайшего большего целого. Если дробная часть равна `0.5`, то округление производится до ближайшего четного числа. Пример:

```
>>> round(0.49), round(0.50), round(0.51)
(0, 0, 1)
>>> round(1.49), round(1.50), round(1.51)
(1, 2, 2)
>>> round(2.49), round(2.50), round(2.51)
(2, 2, 3)
>>> round(3.49), round(3.50), round(3.51)
(3, 4, 4)
```

Во втором параметре можно указать желаемое количество знаков после запятой. Если оно не указано, используется значение `0` (т. е. число будет округлено до целого):

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

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

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

- `pow(<Число>, <Степень>[, <Делитель>])` — 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 полученного результата на значение этого параметра:

```
>>> 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)
```

- `sum(<Последовательность>[, <Начальное значение>])` — возвращает сумму значений элементов последовательности (например: списка, кортежа) плюс `<Начальное значение>`. Если второй параметр не указан, начальное значение принимается равным `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)
```

Следует понимать, что все типы данных, поддерживаемые Python, представляют собой *классы*. Класс `float`, представляющий вещественные числа, поддерживает следующие полезные *методы*:

- `is_integer()` — возвращает `True`, если заданное вещественное число не содержит дробной части, т. е. фактически представляет собой целое число:

```
>>> (2.0).is_integer()
True
>>> (2.3).is_integer()
False
```

- as_integer_ratio() — возвращает кортеж из двух целых чисел, представляющих собой числитель и знаменатель дроби, которая соответствует заданному числу:

```
>>> (0.5).as_integer_ratio()
(1, 2)
>>> (2.3).as_integer_ratio()
(2589569785738035, 1125899906842624)
```

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
3.141592653589793
```

- e — возвращает значение константы *e*:

```
>>> math.e
2.718281828459045
```

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)
3.141592653589793
```

- exp() — экспонента;
- log(<Число>[, <База>]) — логарифм по заданной базе. Если база не указана, вычисляется натуральный логарифм (по базе *e*);

- `log10()` — **десятичный логарифм;**
- `log2()` — **логарифм по базе 2;**
- `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, 6, 6)
```

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

```
>>> math.floor(5.49), math.floor(5.50), math.floor(5.51)
(5, 5, 5)
```

- `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:**

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

- `fsum(<Список чисел>)` — **возвращает точную сумму чисел из заданного списка:**

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

> ***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 его функции:

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

```
>>> import random
>>> random.random()
0.9753144027290991
>>> random.random()
0.5468390487484339
>>> random.random()
0.13015058054767736
```

- `seed([<Параметр>][, version=2])` — **настраивает генератор случайных чисел на новую последовательность. Если первый параметр не указан, в качестве базы для случайных чисел будет использовано системное время. Если значение первого 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)
9.965569925394552
>>> random.uniform(0, 10)
0.4455638245043303
```

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

```
>>> random.randint(0, 10)
4
>>> random.randint(0, 10)
10
```

- `randrange([<Начало>, ]<Конец>[, <Шаг>])` — **возвращает случайный элемент из числовой последовательности. Параметры аналогичны параметрам функции** `range()`. **Можно сказать, что функция** `randrange` **«за кадром» создает диапазон, из которого и будут выбираться возвращаемые случайные числа:**

```
>>> random.randrange(10)
5
>>> random.randrange(0, 10)
2
>>> random.randrange(0, 10, 2)
6
```

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

```
>>> random.choice("string")          # Строка
'i'
```

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

- `shuffle(<Список>[, <Число от 0.0 до 1.0>])` — **перемешивает элементы списка случайным образом. Если второй параметр не указан, то используется значение, возвращаемое функцией** `random()`. **Никакого результата при этом не возвращается. Пример:**

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

- `sample(<Последовательность>, <Количество элементов>)` — **возвращает список из указанного количества элементов, которые будут выбраны случайным образом из заданной последовательности. В качестве таковой 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(range(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: utf-8 -*-
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 range(count_char):
        passw.append(random.choice(arr))
    return "".join(passw)

# Вызываем функцию
print( passw_generator(10) )  # Выведет что-то вроде ngODHE8J8x
print( passw_generator() )    # Выведет что-то вроде ZxcpkF5O
input()
```

ГЛАВА 6

Строки и двоичные данные

Строки представляют собой упорядоченные последовательности 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 (операторы `in` и `not 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 будет 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 3 поддерживает следующие строковые типы:

- `str` — Unicode-строка. Обратите внимание, конкретная кодировка: UTF-8, UTF-16 или UTF-32 — здесь не указана. 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 последовательность байтов в какой-либо кодировке:

  ```
  >>> type("строка")
  <class 'str'>
  ```

```
>>> "строка".encode(encoding="cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> "строка".encode(encoding="utf-8")
b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
```

- `bytes` — **неизменяемая последовательность байтов. Каждый элемент последовательности может хранить целое число от** `0` **до** `255`**, которое обозначает код символа. Объект типа** `bytes` **поддерживает большинство строковых методов и, если это возможно, выводится как последовательность символов. Однако доступ по индексу возвращает целое число, а не символ. Пример:**

```
>>> s = bytes("стр str", "cp1251")
>>> s[0], s[5], s[0:3], s[4:7]
(241, 116, b'\xf1\xf2\xf0', b'str')
>>> s
b'\xf1\xf2\xf0 str'
```

Объект типа `bytes` **может содержать как однобайтовые, так и многобайтовые символы. Обратите внимание на то, что функции и методы строк некорректно работают с многобайтовыми кодировками, — например, функция** `len()` **вернет количество байтов, а не символов:**

```
>>> len("строка")
6
>>> len(bytes("строка", "cp1251"))
6
>>> len(bytes("строка", "utf-8"))
12
```

- `bytearray` — **изменяемая последовательность байтов. Тип** `bytearray` **аналогичен типу** `bytes`**, но позволяет изменять элементы по индексу и содержит дополнительные методы, дающие возможность добавлять и удалять элементы. Пример:**

```
>>> s = bytearray("str", "cp1251")
>>> s[0] = 49; s                # Можно изменить символ
bytearray(b'1tr')
>>> s.append(55); s             # Можно добавить символ
bytearray(b'1tr7')
```

Во всех случаях, когда речь идет о текстовых данных, следует использовать тип `str`**. Именно этот тип мы будем называть словом «строка». Типы** `bytes` **и** `bytearray` **следует задействовать для записи бинарных данных — например, изображений, а также для промежуточного хранения текстовых данных. Более подробно типы** `bytes` **и** `bytearray` **мы рассмотрим в конце этой главы.**

6.1. Создание строки

Создать строку можно следующими способами:

- **с помощью функции** `str([<Объект>[, <Кодировка>[, <Обработка ошибок>]]])`**. Если указан только первый параметр, то функция возвращает строковое представление любого объекта. Если параметры не указаны вообще, то возвращается пустая строка.**

Пример:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0")
"b'\\xf1\\xf2\\xf0\\xee\\xea\\xe0'"
```

Обратите внимание на преобразование объекта типа `bytes`. **Мы получили строковое представление объекта, а не нормальную строку. Чтобы получить из объектов типа** `bytes` **и** `bytearray` **именно строку, следует указать кодировку во втором параметре:**

```
>>> str(b"\xf1\xf2\xf0\xee\xea\xe0", "cp1251")
'строка'
```

В третьем параметре могут быть указаны значения `"strict"` **(при ошибке возбуждается исключение** `UnicodeDecodeError` **— значение по умолчанию),** `"replace"` **(неизвестный символ заменяется символом, имеющим код** `\uFFFD`**) или** `"ignore"` **(неизвестные символы игнорируются):**

```
>>> obj1 = bytes("строка1", "utf-8")
>>> obj2 = bytearray("строка2", "utf-8")
>>> str(obj1, "utf-8"), str(obj2, "utf-8")
('строка1', 'строка2')
>>> str(obj1, "ascii", "strict")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    str(obj1, "ascii", "strict")
UnicodeDecodeError: 'ascii' codec can't decode byte
0xd1 in position 0: ordinal not in range(128)
>>> str(obj1, "ascii", "ignore")
'1'
```

- **указав строку между апострофами или двойными кавычками:**

```
>>> 'строка', "строка", '"x": 5', "'x': 5"
('строка', 'строка', '"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 они 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[1]:**

---

[1] Вообще-то, наклоненный таким образом слеш «\» называется *обратным*, в отличие от *прямого* слеша, наклоненного так «/». Однако для упрощения изложения здесь и далее мы будем называть обратный слеш просто слешем.

```
>>> 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 string literal
```

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 \ 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 string literal
>>> print("string\\")
string\
```

указав строку между утроенными апострофами или утроенными кавычками. Такие объекты можно разместить на нескольких строках, допускается также одновременно использовать и 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 атрибуте `__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
```

Поскольку 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:\Python34\lib\site-packages")
C:\Python34\lib\site-packages
```

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

```
>>> print("C:\\Python34\\lib\\site-packages")
C:\Python34\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:\Python34\lib\site-packages\")

SyntaxError: EOL while scanning string literal
>>> print(r"C:\Python34\lib\site-packages\\")
C:\Python34\lib\site-packages\\
```

Чтобы избавиться от лишнего слеша, можно использовать операцию конкатенации строк, обычные строки или удалить слеш явным образом:

```
>>> print(r"C:\Python34\lib\site-packages" + "\\") # Конкатенация
C:\Python34\lib\site-packages\
```

```
>>> print("C:\\Python34\\lib\\site-packages\\")     # Обычная строка
C:\Python34\lib\site-packages\
>>> print(r"C:\Python34\lib\site-packages\\"[:-1]) # Удаление слеша
C:\Python34\lib\site-packages\
```

6.2. Специальные символы

Специальные символы — это комбинации знаков, обозначающих служебные или непечатаемые символы, которые невозможно вставить обычным способом. Приведем перечень специальных символов, допустимых 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;
- `\N` — восьмеричное значение `N`. Например, `\74` соответствует символу `<`;
- `\xN` — шестнадцатеричное значение `N`. Например, `\x6a` соответствует символу `j`;
- `\\` — обратный слеш;
- `\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 последовательностям. 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#90>", 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 — чтобы получить положительный индекс, значение вычитается из длины строки:

```
>>> 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#94>", 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 здесь не являются обязательными. 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)
```

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

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

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, end=" ")
```

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
<class '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, проверку на вхождение и невхождение. Повторить строку указанное количество раз можно с помощью оператора `*`, выполнить проверку на вхождение фрагмента в строку позволяет оператор `in`, а проверить на невхождение — оператор `not in`:

```
>>> "-" * 20
'--------------------'
>>> "yt" in "Python"                # Найдено
True
>>> "yt" in "Perl"                  # Не найдено
False
>>> "PHP" not in "Python"           # Не найдено
True
```

6.4. Форматирование строк

Вместо соединения строк с помощью оператора `+` лучше использовать форматирование. Эта операция позволяет соединять строку с любым другим типом данных и выполняется быстрее конкатенации.

> ***Примечание***
>
> В последующих версиях Python оператор форматирования `%` может быть удален. Вместо этого оператора в новом коде следует использовать метод `format()`, который рассматривается в следующем разделе.

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 комбинацию символов** 0o, **для шестнадцатеричных значений добавляет комбинацию символов** 0x **(если используется тип** x**) или** 0X **(если используется тип** X**), для вещественных чисел предписывает всегда выводить дробную точку, даже если задано значение** 0 **в параметре** <Точность>:

```
>>> print("%#o %#o %#o" % (0o77, 10, 10.5))
0o77 0o12 0o12
>>> 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)
'3.141592653589793 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"))
'Обычная строка'
```

- a — **преобразует объект в строку с помощью функции** `ascii()`:

```
>>> print("%a" % ("строка"))
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

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

```
>>> for i in range(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" % (0o77, 10, 10.5))
77 12 12
>>> print("%#o %#o %#o" % (0o77, 10, 10.5))
0o77 0o12 0o12
```

- 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** f **или** e **(выбирается более короткая запись числа):**

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

- G — **эквивалентно** f **или** E **(выбирается более короткая запись 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#55>", 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 Web-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: utf-8 -*-
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
input()
```

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>
```

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 указанной в вызове метода 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')
```

- `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), "123456".zfill(5)
('00000000000000000005', '123456')
```

6.5. Метод *format()*

Помимо операции форматирования, мы можем использовать для этой же цели метод `format()`. Он имеет следующий синтаксис:

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

В параметре `<Строка специального формата>` внутри символов фигурных скобок: `{` 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'
>>> "{model} - {color}".format(color="red", model="BMW") # Ключи
'BMW - red'
>>> d = {"color": "red", "model": "BMW"}
>>> "{model} - {color}".format(**d)                    # Ключи
'BMW - red'
>>> "{color} - {0}".format(2015, color="red")          # Комбинация
'red - 2015'
```

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.model} — {0.color}".format(car)                # Атрибуты
'BMW — red'
```

Существует также краткая форма записи, при которой параметр <Поле> не указывается. В этом случае скобки без указанного индекса нумеруются слева направо, начиная с нуля:

```
>>> "{} — {} — {} — {n}".format(1, 2, 3, n=4) # "{0} — {1} — {2} — {n}"
'1 — 2 — 3 — 4'
>>> "{} — {} — {n} — {}".format(1, 2, 3, n=4) # "{0} — {1} — {n} — {2}"
'1 — 2 — 4 — 3'
```

- Параметр <Функция> задает функцию, с помощью которой обрабатываются данные перед вставкой в строку. Если указано значение `s`, то данные обрабатываются функцией `str()`, если значение `r`, то функцией `repr()`, а если значение `a`, то функцией `ascii()`. 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()
'строка'
>>> print("{0!a}".format("строка"))                    # ascii()
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

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

```
[[<Заполнитель>]<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 приведенного примера, пространство между знаком и числом по умолчанию заполняется пробелами, а знак положительного числа не указывается. Чтобы вместо пробелов пространство заполнялось нулями, необходимо указать нуль перед шириной поля:

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

- Такого же эффекта можно достичь, указав нуль в параметре <Заполнитель>. В этом параметре допускаются и другие символы, которые будут выводиться вместо пробелов:

```
>>> "'{0:0=10}' '{1:0=10}'".format(-3, 3)
"'-000000003' '0000000003'"
>>> "'{0:*<10}' '{1:+>10}' '{2:.^10}'".format(3, 3, 3)
"'3*********' '+++++++++3' '....3.....'"
```

- 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. Например, выведем большое число с разделением тысячных разрядов пробелом:

```
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, 'Russian_Russia.1251')
'Russian_Russia.1251'
>>> print("{0:n}".format(100000000).replace("\uffa0", " "))
100 000 000
```

В Python 3 между разрядами вставляется символ с кодом \uffa0, который отображается квадратиком. Чтобы вывести символ пробела, мы производим замену в строке с помощью метода replace(). В Python версии 2 поведение было другим. Там вставлялся символ с кодом \xa0 и не нужно было производить замену. Чтобы в Python 3 вставлялся символ с кодом \xa0, следует воспользоваться функцией format() из модуля locale:

```
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, "Russian_Russia.1251")
'Russian_Russia.1251'
>>> print(locale.format("%d", 100000000, grouping = True))
100 000 000
>>> locale.localeconv()["thousands_sep"]
'\xa0'
```

Можно также разделить тысячные разряды запятой, указав ее в строке формата:

```
>>> print("{0:,d}".format(100000000))
100,000,000
```

- 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'"
```

- По умолчанию выводимое число имеет шесть знаков после запятой. Задать другое количество знаков после запятой мы можем в параметре <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'"
```

Здесь по умолчанию количество знаков после запятой также равно шести, но мы можем указать другую величину этого параметра:

```
>>> "'{0:.2e}' '{1:.2E}'".format(3000, 18657.81452)
"'3.00e+03' '1.87E+04'"
```

- `g` — эквивалентно `f` или `e` (выбирается более короткая запись 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 `f` или `E` (выбирается более короткая запись 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 функцией `print()` для вывода объектов. Пример:

```
>>> str(), str([1, 2]), str((3, 4)), str({"x": 1})
('', '[1, 2]', '(3, 4)', "{'x': 1}")
>>> print("строка1\nстрока2")
строка1
строка2
```

- `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})
("'Строка'", '[1, 2, 3]', "{'x': 5}")
>>> repr("строка1\nстрока2")
"'строка1\\nстрока2'"
```

- `ascii(<Объект>)` — возвращает строковое представление объекта. В строке могут быть символы только из кодировки ASCII.

Пример:

```
>>> ascii([1, 2, 3]), ascii({"x": 5})
('[1, 2, 3]', "{'x': 5}")
>>> ascii("строка")
"'\\u0441\\u0442\\u0440\\u043e\\u043a\\u0430'"
```

- `len(<Строка>)` — **возвращает количество символов в строке:**

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

Приведем перечень основных методов для работы со строками:

- `strip([<Символы>])` — **удаляет указанные в параметре символы в начале и в конце строки. Если параметр не задан, удаляются пробельные символы: 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'"
```

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

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.lstrip(), s2.lstrip("tsr"))
"'str     ' — 'okstrstrstr'"
```

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

```
>>> s1, s2 = "     str     ", "strstrstrokstrstrstr"
>>> "'%s' — '%s'" % (s1.rstrip(), s2.rstrip("tsr"))
"'     str' — 'strstrstrok'"
```

- `split([<Разделитель>[, <Лимит>]])` — **разделяет строку на подстроки по указанному разделителю и добавляет эти подстроки в список, который возвращается в качестве результата. Если первый параметр не указан или имеет значение `None`, то в качестве разделителя используется символ пробела. Во втором параметре можно задать количество подстрок в результирующем списке — если он не указан или равен `-1`, в список попадут все подстроки. Если подстрок больше указанного количества, то список будет содержать еще один элемент — с остатком 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 возникнуть 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>]])` — 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 перевода 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 — сам разделитель, а третий элемент — фрагмент, расположенный после разделителя. Поиск производится слева направо. Если символ-разделитель не найден, то первый элемент кортежа будет содержать всю строку, а остальные элементы останутся 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 окажутся 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#48>", line 1, in <module>
    " ".join(("word1", "word2", 5))
TypeError: sequence item 2: expected str instance, 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'
```

В Python 3 можно также преобразовать строку в тип `bytearray`, **а затем изменить символ по индексу:**

```
>>> s = "Python"
>>> b = bytearray(s, "cp1251"); b
bytearray(b'Python')
>>> b[0] = ord("J"); b
bytearray(b'Jython')
>>> s = b.decode("cp1251"); s
'Jython'
```

6.7. Настройка локали

Для установки локали (совокупности локальных настроек системы) служит 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 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')
```

Получить настройки локали позволяет функция `localeconv()`. Функция возвращает словарь с настройками. Результат выполнения функции для локали `Russian_Russia.1251` выглядит следующим образом:

```
>>> locale.localeconv()
{'mon_decimal_point': ',', 'int_frac_digits': 2, 'p_sep_by_space': 0,
'frac_digits': 2, 'thousands_sep': ' ', 'n_sign_posn': 1,
'decimal_point': ',', 'int_curr_symbol': 'RUR', 'n_cs_precedes': 0,
```

```
'p_sign_posn': 1, 'mon_thousands_sep': '\xa0', 'negative_sign': '-',
'currency_symbol': 'p.', 'n_sep_by_space': 0, 'mon_grouping': [3, 0],
'p_cs_precedes': 0, 'positive_sign': '', 'grouping': [3, 0]}
```

6.8. Изменение регистра символов

Для изменения регистра символов предназначены следующие методы:

- `upper()` — **заменяет все символы строки соответствующими прописными буквами:**

```
>>> print("строка".upper())
СТРОКА
```

- `lower()` — **заменяет все символы строки соответствующими строчными буквами:**

```
>>> print("СТРОКА".lower())
строка
```

- `swapcase()` — **заменяет все строчные символы соответствующими прописными буквами, а все прописные символы — строчными:**

```
>>> print("СТРОКА строка".swapcase())
строка СТРОКА
```

- `capitalize()` — **делает первую букву строки прописной:**

```
>>> print("строка строка".capitalize())
Строка строка
```

- `title()` — **делает первую букву каждого слова прописной:**

```
>>> s = "первая буква каждого слова станет прописной"
>>> print(s.title())
Первая Буква Каждого Слова Станет Прописной
```

- `casefold()` — **то же самое, что и** `lower()`**, но дополнительно преобразует все символы с диакритическими знаками и лигатуры в буквы стандартной латиницы. Обычно применяется для сравнения строк:**

```
>>> "Python".casefold() == "python".casefold()
True
>>> "grosse".casefold() == "große".casefold()
True
```

6.9. Функции для работы с символами

Для работы с отдельными символами предназначены следующие функции:

- `chr(<Код символа>)` — **возвращает символ по указанному коду:**

```
>>> print(chr(1055))
П
```

- `ord(<Символ>)` — **возвращает код указанного символа:**

```
>>> print(ord("П"))
1055
```

6.10. 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. Формат метода:

  ```
  <Строка>.find(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Если начальная позиция не указана, то поиск будет осуществляться с начала строки. Если параметры `<Начало>` и `<Конец>` указаны, то производится операция извлечения среза:

  ```
  <Строка>[<Начало>:<Конец>]
  ```

  и поиск подстроки будет выполняться в этом фрагменте. Пример:

  ```
  >>> 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)
  ```

- `index()` — метод аналогичен методу `find()`, но если подстрока в строку не входит, то возбуждается исключение `ValueError`. Формат метода:

  ```
  <Строка>.index(<Подстрока>[, <Начало>[, <Конец>]])
  ```

  Пример:

  ```
  >>> 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#24>", line 1, in <module>
      s.index("тест")
  ValueError: substring not found
  ```

- `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. Формат метода:

  ```
  <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. Пример:

  ```
  >>> 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`. Формат метода:

  ```
  <Строка>.rindex(<Подстрока>[, <Начало>[, <Конец>]])
  ```

Пример:

```
>>> 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#30>", 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. Формат метода:**

```
<Строка>.count(<Подстрока>[, <Начало>[, <Конец>]])
```

Пример:

```
>>> 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. Формат метода:**

```
<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. Пример:**

```
>>> s = "пример пример Пример Пример"
>>> s.startswith("при"), s.startswith("При")
(True, False)
>>> s.startswith("при", 6), s.startswith("При", 14)
(False, True)
```

Начиная с Python 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. Формат метода:**

```
<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.**

Пример:

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith("ока"), s.endswith("ОКА")
(False, True)
>>> s.endswith("ока", 0, 9)
True
```

Начиная с Python версии 2.5, параметр `<Подстрока>` **может быть кортежем:**

```
>>> s = "подстрока ПОДСТРОКА"
>>> s.endswith(("ока", "ОКА"))
True
```

- `replace()` — **производит замену всех вхождений заданной подстроки в строке на другую подстроку и возвращает результат в виде новой строки. Метод зависит от регистра символов. Формат метода:**

```
<Строка>.replace(<Подстрока для замены>, <Новая подстрока>[,
                 <Максимальное количество замен>])
```

Если количество замен не указано, будет выполнена замена всех найденных подстрок. 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')
```

- `translate(<Таблица символов>)` — **заменяет символы в соответствии с параметром** `<Таблица символов>`. **Параметр** `<Таблица символов>` **должен быть словарем, ключами которого являются Unicode-коды заменяемых символов, а значениями — Unicode-коды заменяющих символов. Если в качестве значения указать** `None`**, то символ будет удален. Для примера удалим букву** `П`**, а также изменим регистр всех букв** `р`**:**

```
>>> s = "Пример"
>>> d = {ord("П"): None, ord("р"): ord("Р")}
>>> d
{1088: 1056, 1055: None}
>>> s.translate(d)
'РимеР'
```

Упростить создание параметра `<Таблица символов>` **позволяет статический метод** `maketrans()`. **Формат метода:**

```
str.maketrans(<X>[, <Y>[, <Z>]])
```

Если указан только первый параметр, то он должен быть словарем:

```
>>> t = str.maketrans({"a": "A", "o": "O", "c": None})
>>> t
{1072: 'A', 1089: None, 1086: 'O'}
```

```
>>> "строка".translate(t)
'трОкА'
```

Если указаны два первых параметра, то они должны быть строками одинаковой длины. В результате будет создан словарь с ключами из строки `<X>` и значениями из строки `<Y>`, расположенными в той же позиции. Изменим регистр некоторых символов:

```
>>> t = str.maketrans("абвгдежзи", "АБВГДЕЖЗИ")
>>> t
{1072: 1040, 1073: 1041, 1074: 1042, 1075: 1043, 1076: 1044,
1077: 1045, 1078: 1046, 1079: 1047, 1080: 1048}
>>> "абвгдежзи".translate(t)
'АБВГДЕЖЗИ'
```

В третьем параметре можно дополнительно указать строку из символов, которым будет сопоставлено значение `None`. После выполнения метода `translate()` эти символы будут удалены из строки. Заменим все цифры на `0`, а некоторые буквы удалим из строки:

```
>>> t = str.maketrans("123456789", "0" * 9, "str")
>>> t
{116: None, 115: None, 114: None, 49: 48, 50: 48, 51: 48,
52: 48, 53: 48, 54: 48, 55: 48, 56: 48, 57: 48}
>>> "str123456789str".translate(t)
'000000000'
```

6.11. Проверка типа содержимого строки

Для проверки типа содержимого строки предназначены следующие методы:

- `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)
```

- `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)
```

- `isdigit()` — возвращает `True`, если строка содержит только цифры, в противном случае — `False`:

```
>>> "0123".isdigit(), "123abc".isdigit(), "abc123".isdigit()
(True, False, False)
```

- `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:

```
>>> "123".isdecimal(), "123стр".isdecimal()
(True, False)
```

- `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:

```
>>> "\u2155".isnumeric(), "\u2155".isdigit()
(True, False)
>>> print("\u2155") # Выведет символ "1/5"
```

- `isupper()` — возвращает `True`, если строка содержит буквы только верхнего регистра, в противном случае — `False`:

```
>>> "STRING".isupper(), "СТРОКА".isupper(), "".isupper()
(True, True, False)
>>> "STRING1".isupper(), "СТРОКА, 123".isupper(), "123".isupper()
(True, True, False)
>>> "string".isupper(), "STRing".isupper()
(False, False)
```

- `islower()` — возвращает `True`, если строка содержит буквы только нижнего регистра, в противном случае — `False`:

```
>>> "srting".islower(), "строка".islower(), "".islower()
(True, True, False)
>>> "string1".islower(), "str, 123".islower(), "123".islower()
(True, True, False)
>>> "STRING".islower(), "Строка".islower()
(False, False)
```

- `istitle()` — возвращает `True`, если все слова в строке начинаются с заглавной буквы, в противном случае — `False`. Если строка пуста, также возвращается `False`. 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)
```

- `isprintable()` — возвращает `True`, если строка содержит только печатаемые символы, в противном случае — `False`. Отметим, что пробел относится к печатаемым символам. Примеры:

```
>>> "123".isprintable
True
```

```
>>> "PHP Python".isprintable()
True
>>> "\n".isprintable()
False
```

- `isspace()` — **возвращает** `True`, **если строка содержит только пробельные символы, в противном случае** — `False`:

```
>>> "".isspace(), " \n\r\t".isspace(), "str str".isspace()
(False, True, False)
```

- `isidentifier()` — **возвращает** `True`, **если строка представляет собой допустимое с точки зрения Python имя переменной, функции или класса, в противном случае** — `False`:

```
>>> "s".isidentifier()
True
>>> "func".isidentifier()
True
>>> "123func".isidentifier()
False
```

Следует иметь в виду, что метод `isidentifier()` **лишь проверяет, удовлетворяет ли заданное имя правилам языка. Он не проверяет, совпадает ли это имя с ключевым словом Python. Для этого надлежит применять функцию** `iskeyword()`**, объявленную в модуле** `keyword`**, которая возвращает** `True`**, если переданная ей строка совпадает с одним из ключевых слов:**

```
>>> import keyword
>>> keyword.iskeyword("else")
True
>>> keyword.iskeyword("elsewhere")
False
```

Переделаем нашу программу (см. листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой. Кроме того, предусмотрим возможность ввода отрицательных целых чисел (листинг 6.4).

Листинг 6.4. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = 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)
input()
```

Процесс ввода значений и получения результата выглядит так (значения, введенные пользователем, выделены здесь полужирным шрифтом):

```
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
```

6.12. Тип данных *bytes*

Тип данных `str` отлично подходит для хранения текстовой информации, но что делать, если необходимо обрабатывать изображения? Ведь изображение не имеет кодировки, а значит, оно не может быть преобразовано в Unicode-строку. Для решения этой проблемы в Python 3 были введены типы `bytes` и `bytearray`, которые позволяют хранить последовательность целых чисел в диапазоне от `0` до `255`. Каждое такое число обозначает код символа. Тип данных `bytes` относится к неизменяемым типам, как и строки, а тип данных `bytearray` — к изменяемым, как и списки.

Создать объект типа `bytes` можно следующими способами:

- **с помощью функции `bytes([<Строка>, <Кодировка>[, <Обработка ошибок>]])`. Если параметры не указаны, то возвращается пустой объект. Чтобы преобразовать строку в объект типа `bytes`, необходимо передать минимум два первых параметра. Если строка указана только в первом параметре, то возбуждается исключение `TypeError`. Пример:**

```
>>> bytes()
b''
>>> bytes("строка", "cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> bytes("строка")
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    bytes("строка")
TypeError: string argument without an encoding
```

В третьем параметре могут быть указаны значения `"strict"` **(при ошибке возбуждается исключение** `UnicodeEncodeError` **— значение по умолчанию),** `"replace"` **(неизвестный символ заменяется символом вопроса) или** `"ignore"` **(неизвестные символы игнорируются). Пример:**

```
>>> bytes("string\uFFFD", "cp1251", "strict")
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    bytes("string\uFFFD", "cp1251", "strict")
  File "C:\Python34\lib\encodings\cp1251.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character
'\ufffd' in position 6: character maps to <undefined>
>>> bytes("string\uFFFD", "cp1251", "replace")
b'string?'
>>> bytes("string\uFFFD", "cp1251", "ignore")
b'string'
```

- **с помощью метода строк** `encode([encoding="utf-8"][, errors="strict"])`. **Если кодировка не указана, то строка преобразуется в последовательность байтов в кодировке UTF-8. В параметре** `errors` **могут быть указаны значения** `"strict"` **(значение по умолчанию),** `"replace"`, `"ignore"`, `"xmlcharrefreplace"` **или** `"backslashreplace"`. **Пример:**

```
>>> "строка".encode()
b'\xd1\x81\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
>>> "строка".encode(encoding="cp1251")
b'\xf1\xf2\xf0\xee\xea\xe0'
>>> "строка\uFFFD".encode(encoding="cp1251",
                          errors="xmlcharrefreplace")
b'\xf1\xf2\xf0\xee\xea\xe0&#65533;'
>>> "строка\uFFFD".encode(encoding="cp1251",
                          errors="backslashreplace")
b'\xf1\xf2\xf0\xee\xea\xe0\\ufffd'
```

- **указав букву** `b` **(регистр не имеет значения) перед строкой в апострофах, кавычках, тройных апострофах или тройных кавычках. Обратите внимание на то, что в строке могут быть только символы с кодами, входящими в кодировку ASCII. Все остальные символы должны быть представлены специальными последовательностями:**

```
>>> b"string", b'string', b"""string""", b'''string'''
(b'string', b'string', b'string', b'string')
>>> b"строка"
SyntaxError: bytes can only contain ASCII literal characters.
>>> b"\xf1\xf2\xf0\xee\xea\xe0"
b'\xf1\xf2\xf0\xee\xea\xe0'
```

- **с помощью функции** `bytes(<Последовательность>)`, **которая преобразует последовательность целых чисел от** `0` **до** `255` **в объект типа** `bytes`. **Если число не попадает в диапазон, то возбуждается исключение** `ValueError`. **Пример:**

```
>>> b = bytes([225, 226, 224, 174, 170, 160])
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
```

```
>>> str(b, "cp866")
'строка'
```

- **с помощью функции** `bytes(<Число>)`**, которая задает количество элементов в последовательности. Каждый элемент будет содержать нулевой символ:**

```
>>> bytes(10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

- **с помощью метода** `bytes.fromhex(<Строка>)`**. Строка в этом случае должна содержать шестнадцатеричные значения символов:**

```
>>> b = bytes.fromhex(" e1 e2e0ae aaa0 ")
>>> b
b'\xe1\xe2\xe0\xae\xaa\xa0'
>>> str(b, "cp866")
'строка'
```

Объекты типа `bytes` **относятся к последовательностям. Каждый элемент такой последовательности может хранить целое число от** `0` **до** `255`**, которое обозначает код символа. Как и все последовательности, объекты поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение:**

```
>>> b = bytes("string", "cp1251")
>>> b
b'string'
>>> b[0]                          # Обращение по индексу
115
>>> b[1:3]                        # Получение среза
b'tr'
>>> b + b"123"                    # Конкатенация
b'string123'
>>> b * 3                         # Повторение
b'stringstringstring'
>>> 115 in b, b"tr" in b, b"as" in b
(True, True, False)
```

Как видно из примера, при выводе объекта целиком, а также при извлечении среза, производится попытка отображения символов. Однако доступ по индексу возвращает целое число, а не символ. Если преобразовать объект в список, то мы получим последовательность целых чисел:

```
>>> list(bytes("string", "cp1251"))
[115, 116, 114, 105, 110, 103]
```

Тип `bytes` **относится к неизменяемым типам. Это означает, что можно получить значение по индексу, но изменить его нельзя:**

```
>>> b = bytes("string", "cp1251")
>>> b[0] = 168
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    b[0] = 168
TypeError: 'bytes' object does not support item assignment
```

Объекты типа bytes поддерживают большинство строковых методов, которые мы рассматривали в предыдущих разделах. Однако некоторые из этих методов могут некорректно работать с русскими буквами — в этих случаях следует использовать тип str, а не тип bytes. Не поддерживаются объектами типа bytes строковые методы encode(), isidentifier(), isprintable(), isnumeric(), isdecimal(), format_map() и format(), а также операция форматирования.

При использовании методов следует учитывать, что в параметрах нужно указывать объекты типа bytes, а не строки:

```
>>> b = bytes("string", "cp1251")
>>> b.replace(b"s", b"S")
b'String'
```

Необходимо также помнить, что смешивать строки и объекты типа bytes в выражениях нельзя. Предварительно необходимо явно преобразовать объекты к одному типу, а лишь затем производить операцию:

```
>>> b"string" + "string"
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    b"string" + "string"
TypeError: can't concat bytes to str
>>> b"string" + "string".encode("ascii")
b'stringstring'
```

Объект типа bytes может содержать как однобайтовые символы, так и многобайтовые. При использовании многобайтовых символов некоторые функции могут работать не так, как вы думаете, — например, функция len() вернет количество байтов, а не символов:

```
>>> len("строка")
6
>>> len(bytes("строка", "cp1251"))
6
>>> len(bytes("строка", "utf-8"))
12
```

Преобразовать объект типа bytes в строку позволяет метод decode(). Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр encoding задает кодировку символов (по умолчанию UTF-8) в объекте bytes, а параметр errors — способ обработки ошибок при преобразовании. В параметре errors можно указать значения "strict" (значение по умолчанию), "replace" или "ignore". Пример преобразования:

```
>>> b = bytes("строка", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
('строка', 'строка')
```

Для преобразования можно также воспользоваться функцией str():

```
>>> b = bytes("строка", "cp1251")
>>> str(b, "cp1251")
'строка'
```

Чтобы изменить кодировку данных, следует сначала преобразовать тип `bytes` **в строку, а затем произвести обратное преобразование, указав нужную кодировку. Преобразуем данные из кодировки Windows-1251 в кодировку KOI8-R, а затем обратно (листинг 6.5).**

Листинг 6.5. Преобразование кодировок

```
>>> w = bytes("Строка", "cp1251") # Данные в кодировке windows-1251
>>> k = w.decode("cp1251").encode("koi8-r")
>>> k, str(k, "koi8-r")           # Данные в кодировке KOI8-R
(b'\xf3\xd4\xd2\xcf\xcb\xc1', 'Строка')
>>> w = k.decode("koi8-r").encode("cp1251")
>>> w, str(w, "cp1251")           # Данные в кодировке windows-1251
(b'\xd1\xf2\xf0\xee\xea\xe0', 'Строка')
```

6.13. Тип данных *bytearray*

Тип данных `bytearray` **является разновидностью типа** `bytes` **и поддерживает те же самые методы и операции. В отличие от типа** `bytes`**, тип** `bytearray` **допускает возможность непосредственного изменения объекта и содержит дополнительные методы, позволяющие выполнять эти изменения.**

Создать объект типа `bytearray` **можно следующими способами:**

- **с помощью функции** `bytearray([<Строка>, <Кодировка>[, <Обработка ошибок>]])`**. Если параметры не указаны, то возвращается пустой объект. Чтобы преобразовать строку в объект типа** `bytearray`**, необходимо передать минимум два первых параметра. Если строка указана только в первом параметре, то возбуждается исключение** `TypeError`**. Примеры:**

```
>>> bytearray()
bytearray(b'')
>>> bytearray("строка", "cp1251")
bytearray(b'\xf1\xf2\xf0\xee\xea\xe0')
>>> bytearray("строка")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    bytearray("строка")
TypeError: string argument without an encoding
```

В третьем параметре могут быть указаны значения `"strict"` **(при ошибке возбуждается исключение** `UnicodeEncodeError` **— значение по умолчанию),** `"replace"` **(неизвестный символ заменяется символом вопроса) или** `"ignore"` **(неизвестные символы игнорируются). Примеры:**

```
>>> bytearray("string\uFFFD", "cp1251", "strict")
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    bytearray("string\uFFFD", "cp1251", "strict")
  File "C:\Python34\lib\encodings\cp1251.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
```

```
UnicodeEncodeError: 'charmap' codec can't encode character
'\ufffd' in position 6: character maps to <undefined>
>>> bytearray("string\uFFFD", "cp1251", "replace")
bytearray(b'string?')
>>> bytearray("string\uFFFD", "cp1251", "ignore")
bytearray(b'string')
```

- с помощью функции `bytearray(<Последовательность>)`, которая преобразует последовательность целых чисел от `0` до `255` в объект типа `bytearray`. Если число не попадает в диапазон, то возбуждается исключение `ValueError`. Примеры:

```
>>> b = bytearray([225, 226, 224, 174, 170, 160])
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

- с помощью функции `bytearray(<Число>)`, которая задает количество элементов в последовательности. Каждый элемент будет содержать нулевой символ:

```
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
```

- с помощью метода `bytearray.fromhex(<Строка>)`. Строка в этом случае должна содержать шестнадцатеричные значения символов:

```
>>> b = bytearray.fromhex(" e1 e2e0ae aaa0 ")
>>> b
bytearray(b'\xe1\xe2\xe0\xae\xaa\xa0')
>>> str(b, "cp866")
'строка'
```

Тип `bytearray` относится к изменяемым типам. Поэтому можно не только получить значение по индексу, но и изменить его (что не свойственно строкам):

```
>>> b = bytearray("Python", "ascii")
>>> b[0]                              # Можем получить значение
80
>>> b[0] = b"J"[0]                    # Можем изменить значение
>>> b
bytearray(b'Jython')
```

При изменении значения важно помнить, что присваиваемое значение должно быть целым числом в диапазоне от `0` до `255`. Чтобы получить число в предыдущем примере, мы создали объект типа `bytes`, а затем присвоили значение, расположенное по индексу `0` (`b[0] = b"J"[0]`). Можно, конечно, сразу указать код символа, но ведь держать все коды символов в памяти свойственно компьютеру, а не человеку.

Для изменения объекта можно также использовать следующие методы:

- `append(<Число>)` — добавляет один элемент в конец объекта. Метод изменяет текущий объект и ничего не возвращает.

Пример:

```
>>> b = bytearray("string", "ascii")
>>> b.append(b"1"[0]); b
bytearray(b'string1')
```

- `extend(<Последовательность>)` — **добавляет элементы последовательности в конец объекта. Метод изменяет текущий объект и ничего не возвращает. Пример:**

```
>>> b = bytearray("string", "ascii")
>>> b.extend(b"123"); b
bytearray(b'string123')
```

Добавить несколько элементов можно с помощью операторов + и +=:

```
>>> b = bytearray("string", "ascii")
>>> b + b"123"                    # Возвращает новый объект
bytearray(b'string123')
>>> b += b"456"; b                # Изменяет текущий объект
bytearray(b'string456')
```

Кроме того, можно воспользоваться операцией присваивания значения срезу:

```
>>> b = bytearray("string", "ascii")
>>> b[len(b):] = b"123"           # Добавляем элементы в последовательность
>>> b
bytearray(b'string123')
```

- `insert(<Индекс>, <Число>)` — **добавляет один элемент в указанную позицию. Остальные элементы смещаются. Метод изменяет текущий объект и ничего не возвращает. Добавим элемент в начало объекта:**

```
>>> b = bytearray("string", "ascii")
>>> b.insert(0, b"1"[0]); b
bytearray(b'1string')
```

Метод `insert()` позволяет добавить только один элемент. Чтобы добавить несколько элементов, можно воспользоваться операцией присваивания значения срезу. Добавим несколько элементов в начало объекта:

```
>>> b = bytearray("string", "ascii")
>>> b[:0] = b"123"; b
bytearray(b'123string')
```

- `pop([<Индекс>])` — **удаляет элемент, расположенный по указанному индексу, и возвращает его. Если индекс не указан, то удаляет и возвращает последний элемент. Примеры:**

```
>>> b = bytearray("string", "ascii")
>>> b.pop()                  # Удаляем последний элемент
103
>>> b
bytearray(b'strin')
>>> b.pop(0)                 # Удаляем первый элемент
115
>>> b
bytearray(b'trin')
```

Удалить элемент списка позволяет также оператор `del`:

```
>>> b = bytearray("string", "ascii")
>>> del b[5]              # Удаляем последний элемент
>>> b
bytearray(b'strin')
>>> del b[:2]             # Удаляем первый и второй элементы
>>> b
bytearray(b'rin')
```

- `remove(<Число>)` — удаляет первый элемент, содержащий указанное значение. Если элемент не найден, возбуждается исключение `ValueError`. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> b = bytearray("strstr", "ascii")
>>> b.remove(b"s"[0])     # Удаляет только первый элемент
>>> b
bytearray(b'trstr')
```

- `reverse()` — изменяет порядок следования элементов на противоположный. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> b = bytearray("string", "ascii")
>>> b.reverse(); b
bytearray(b'gnirts')
```

Преобразовать объект типа `bytearray` в строку позволяет метод `decode()`. Метод имеет следующий формат:

```
decode([encoding="utf-8"][, errors="strict"])
```

Параметр `encoding` задает кодировку символов (по умолчанию UTF-8) в объекте `bytearray`, а параметр `errors` — способ обработки ошибок при преобразовании. В параметре `errors` можно указать значения `"strict"` (значение по умолчанию), `"replace"` или `"ignore"`. Пример преобразования:

```
>>> b = bytearray("строка", "cp1251")
>>> b.decode(encoding="cp1251"), b.decode("cp1251")
('строка', 'строка')
```

Для преобразования можно также воспользоваться функцией `str()`:

```
>>> b = bytearray("строка", "cp1251")
>>> str(b, "cp1251")
'строка'
```

6.14. Преобразование объекта в последовательность байтов

Преобразовать объект в последовательность байтов (выполнить его *сериализацию*), а затем восстановить (*десериализовать*) объект позволяет модуль `pickle`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import pickle
```

Для преобразования предназначены две функции:

- `dumps(<Объект>[, <Протокол>][, fix_imports=True])` — **производит сериализацию объекта и возвращает последовательность байтов специального формата. Формат зависит от указанного протокола: число от `0` до `4` (поддержка протокола 4 появилась в Python 3.4). Если второй параметр не указан, будет использован протокол 4 для Python 3.4 или 3 — для предыдущих версий Python 3. Пример преобразования списка и кортежа:**

```
>>> import pickle
>>> obj1 = [1, 2, 3, 4, 5]      # Список
>>> obj2 = (6, 7, 8, 9, 10)     # Кортеж
>>> pickle.dumps(obj1)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
>>> pickle.dumps(obj2)
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

- `loads(<Последовательность байтов>[, fix_imports=True][, encoding="ASCII"][, errors="strict"])` — **преобразует последовательность байтов специального формата обратно в объект, выполняя его десериализацию. Пример восстановления списка и кортежа:**

```
>>> pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
[1, 2, 3, 4, 5]
>>> pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
(6, 7, 8, 9, 10)
```

6.15. Шифрование строк

Для шифрования строк предназначен модуль `hashlib`. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
import hashlib
```

Модуль предоставляет следующие функции: `md5()`, `sha1()`, `sha224()`, `sha256()`, `sha384()` и `sha512()`. В качестве необязательного параметра функциям можно передать шифруемую последовательность байтов. Пример:

```
>>> import hashlib
>>> h = hashlib.sha1(b"password")
```

Передать последовательность байтов можно также с помощью метода `update()`. В этом случае объект присоединяется к предыдущему значению:

```
>>> h = hashlib.sha1()
>>> h.update(b"password")
```

Получить зашифрованную последовательность байтов и строку позволяют два метода: `digest()` и `hexdigest()`. Первый метод возвращает значение, относящееся к типу `bytes`, а второй — строку, содержащую шестнадцатеричные цифры. Примеры:

```
>>> h = hashlib.sha1(b"password")
>>> h.digest()
b'[\xaaa\xe4\xc9\xb9??\x06\x82*\x0bi\xf83\x1b~\xe6\x8f\xd8'
>>> h.hexdigest()
'5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8'
```

Наиболее часто применяемой является функция `md5()`, которая шифрует строку с помощью алгоритма MD5. Эта функция используется для шифрования паролей, т. к. не существует алгоритма для дешифровки зашифрованных ей значений. Для сравнения введенного пользователем пароля с сохраненным в базе необходимо зашифровать введенный пароль, а затем произвести сравнение (листинг 6.6).

Листинг 6.6. Проверка правильности ввода пароля

```
>>> import hashlib
>>> h = hashlib.md5(b"password")
>>> p = h.hexdigest()
>>> p                                  # Пароль, сохраненный в базе
'5f4dcc3b5aa765d61d8327deb882cf99'
>>> h2 = hashlib.md5(b"password")     # Пароль, введенный пользователем
>>> if p == h2.hexdigest(): print("Пароль правильный")

Пароль правильный
```

Свойство `digest_size` хранит размер значения, генерируемого описанными ранее функциями шифрования, в байтах:

```
>>> h = hashlib.sha512(b"password")
>>> h.digest_size
64
```

Python 3.4 поддерживает новый способ устойчивого к взлому шифрования паролей с помощью функции `pbkdf2_hmac()`:

```
pbkdf2_hmac(<Основной алгоритм шифрования>, <Шифруемый пароль>, <"Соль">,
<Количество проходов шифрования>, dklen=None)
```

В качестве основного алгоритма шифрования следует указать строку с наименованием этого алгоритма: `"md5"`, `"sha1"`, `"sha224"`, `"sha256"`, `"sha384"` и `"sha512"`. Шифруемый пароль указывается в виде значения типа `bytes`. `"Соль"` — это особая величина типа `bytes`, выступающая в качестве ключа шифрования, — ее длина не должна быть менее 16 символов. Количество проходов шифрования следует указать достаточно большим (так, при использовании алгоритма SHA512 оно должно составлять 100000).

> ***Примечание***
>
> Кодирование данных с применением функции `pbkdf2_hmac()` отнимает очень много системных ресурсов и может занять значительное время, особенно на маломощных компьютерах.

Последним параметром функции `pbkdf2_hmac()` можно указать длину результирующего закодированного значения в байтах — если она не задана или равна `None`, будет создано значение стандартной для выбранного алгоритма длины (64 байта для алгоритма SHA512). Закодированный пароль возвращается в виде величины типа `bytes`. Пример:

```
>>> import hashlib
>>> dk = hashlib.pbkdf2_hmac('sha512', b'1234567', b'saltsaltsaltsalt', 100000)
>>> dk
b"Sb\x85tc-\xcb@\xc5\x97\x19\x90\x94@\x9f\xde\x07\xa4p-\x83\x94\xf4\x94\x99\x07\xec\
xfa\xf3\xcd\xc3\x88jv\xd1\xe5\x9a\x119\x15/\xa4\xc2\xd3N\xaba\x02\xc0s\xc1\xd1\x0b\
x86xj(\x8c>Mr'@\xbb"
```

ГЛАВА 7

Регулярные выражения

Регулярные выражения предназначены для выполнения в строке сложного поиска или замены. В языке Python использовать регулярные выражения позволяет модуль `re`. Прежде чем задействовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
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 оператор `|`):

- `L` или `LOCALE` — учитываются настройки текущей локали;
- `I` или `IGNORECASE` — поиск без учета регистра. Пример:

```
>>> import re
>>> p = re.compile(r"^[а-яе]+$", re.I | re.U)
>>> print("Найдено" if p.search("АБВГДЕЕ") else "Нет")
Найдено
>>> p = re.compile(r"^[а-яе]+$", re.U)
>>> print("Найдено" if p.search("АБВГДЕЕ") else "Нет")
Нет
```

- `M` или `MULTILINE` — поиск в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`"\n"`). Символ `^` соответствует привязке к началу каждой подстроки, а символ `$` — позиции перед символом перевода строки;
- `S` или `DOTALL` — метасимвол «точка» по умолчанию соответствует любому символу, кроме символа перевода строки (`\n`). Символу перевода строки метасимвол «точка» будет соответствовать в присутствии дополнительного модификатора. Символ `^` соответствует привязке к началу всей строки, а символ `$` — привязке к концу всей строки. Пример:

```
>>> p = re.compile(r".$",
>>> print("Найдено" if p.search("\n") else "Нет")
Нет
```

```
>>> p = re.compile(r"^.$", re.M)
>>> print("Найдено" if p.search("\n") else "Нет")
Нет
>>> p = re.compile(r"^.$", re.S)
>>> print("Найдено" if p.search("\n") else "Нет")
Найдено
```

- X или VERBOSE — если флаг указан, то пробелы и символы перевода строки будут проигнорированы. Внутри регулярного выражения можно использовать и 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 | re.S)
>>> print("Найдено" if p.search("1234567890") else "Нет")
Найдено
>>> print("Найдено" if p.search("abcd123") else "Нет")
Нет
```

- A или ASCII — классы \w, \W, \b, \B, \d, \D, \s и \S будут соответствовать символам в кодировке ASCII (по умолчанию перечисленные классы соответствуют Unicode-символам).

> ***Примечание***
>
> Флаги U и UNICODE, включающие режим соответствия Unicode-символам классов \w, \W, \b, \B, \d, \D, \s и \S, сохранены в Python 3 лишь для совместимости с ранними версиями этого языка и никакого влияния на обработку регулярных выражений не оказывают.

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 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, их следует экранировать с помощью слеша. Некоторые специальные символы теряют свое особое значение, если их разместить внутри квадратных скобок, — в этом случае экранировать их не нужно. Например, как уже было отмечено ранее, метасимвол «точка» по умолчанию соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед точкой нужно указать символ \ или разместить точку внутри квадратных скобок: [.]. Продемонстрируем это на примере проверки правильности введенной даты (листинг 7.1).

Листинг 7.1. Проверка правильности ввода даты

```
# -*- coding: utf-8 -*-
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
input()
```

В этом примере мы осуществляли привязку к началу и концу строки с помощью следующих метасимволов:

- **^ — привязка к началу строки или подстроки. Она зависит от флагов** M **(или** MULTILINE**) и** S **(или** DOTALL**);**
- **$ — привязка к концу строки или подстроки. Она зависит от флагов** M **(или** MULTILINE**) и** S **(или** DOTALL**);**
- **\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** $ **— 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, для проверки, содержит ли строка число (листинг 7.3).

Листинг 7.3. Проверка наличия целого числа в строке

```
# -*- coding: utf-8 -*-
import re                       # Подключаем модуль
p = re.compile(r"^[0-9]+$", re.S)
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
input()
```

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: utf-8 -*-
import re                       # Подключаем модуль
p = re.compile(r"[0-9]+", re.S)
if p.search("Строка245"):
    print("Число")              # Выведет: Число
else:
    print("Не число")
input()
```

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: utf-8 -*-
import re                       # Подключаем модуль
p = re.compile(r"[0-9]+$", re.S)
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]+", re.S)
if p.search("Строка245"):
    print("Есть число в начале строки")
else:
    print("Нет числа в начале строки")
# Выведет: Нет числа в начале строки
input()
```

Поддерживаются также два метасимвола, позволяющие указать привязку к началу или концу слова:

- `\b` — привязка к началу слова (началом слова считается пробел или любой символ, не являющийся буквой, цифрой или знаком подчеркивания);
- `\B` — привязка к позиции, не являющейся началом слова.

Рассмотрим несколько примеров:

```
>>> p = re.compile(r"\bpython\b")
>>> print("Найдено" if p.search("python") else "Нет")
Найдено
>>> print("Найдено" if p.search("pythonware") else "Нет")
Нет
>>> p = re.compile(r"\Bth\B")
>>> print("Найдено" if p.search("python") else "Нет")
Найдено
>>> print("Найдено" if p.search("this") else "Нет")
Нет
```

В квадратных скобках `[]` можно указать символы, которые могут встречаться на этом месте в строке. Можно перечислить символы подряд или указать диапазон через дефис:

- `[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]`, а буква «Ё» — в диапазон `[А-Я]`.

Значение в скобках инвертируется, если после первой скобки вставить 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 необходимо указать после перечисления всех символов, перед закрывающей квадратной скобкой или сразу после открывающей квадратной скобки. Все специальные символы можно сделать обычными, если перед ними указать символ \.

Метасимвол | позволяет сделать выбор между альтернативными значениями. Выражение n|m соответствует одному из символов: n или m. Пример:

```
>>> p = re.compile(r"красн((ая)|(ое))")
>>> print("Найдено" if p.search("красная") else "Нет")
Найдено
>>> print("Найдено" if p.search("красное") else "Нет")
Найдено
>>> print("Найдено" if p.search("красный") else "Нет")
Нет
```

Вместо перечисления символов можно использовать стандартные классы:

- \d — соответствует любой цифре. При указании флага A (ASCII) эквивалентно [0-9];
- \w — соответствует любой букве, цифре или символу подчеркивания. При указании флага A (ASCII) эквивалентно [a-zA-Z0-9_];
- \s — любой пробельный символ. При указании флага A (ASCII) эквивалентно [\t\n\r\f\v];
- \D — не цифра. При указании флага A (ASCII) эквивалентно [^0-9];
- \W — не буква, не цифра и не символ подчеркивания. При указании флага A (ASCII) эквивалентно [^a-zA-Z0-9_];
- \S — не пробельный символ. При указании флага A (ASCII) эквивалентно [^\t\n\r\f\v].

> ***Примечание***
>
> В Python 3 поддержка Unicode в регулярных выражениях установлена по умолчанию. При этом все классы трактуются гораздо шире. Так, класс \d соответствует не только десятичным цифрам, но и другим цифрам из кодировки Unicode, — например, дробям, класс \w включает не только латинские буквы, но и любые другие, а класс \s охватывает также неразрывные пробелы. Поэтому на практике лучше явно указывать символы внутри квадратных скобок, а не использовать классы.

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 вхождений 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 и 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, следует после открывающей круглой скобки 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 фрагмен-

там, 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 скобки. В результате список состоит только из фрагментов, полностью соответствующих регулярному выражению.

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, так: `\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><b>Text4</b>"
>>> p = re.compile(r"<([a-z]+)>(.*?)</\1>", re.S | re.I)
>>> p.findall(s)
[('b', 'Text1'), ('I', 'Text3'), ('b', 'Text4')]
```

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 = "test@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
'test'
>>> 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:

- `(?aiLmsux)` — позволяет установить опции регулярного выражения. Буквы `a`, `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 дефиса, причем перед дефисом и после слов должны следовать пробельные символы:

```
>>> 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 слова не попали в результат, т. к. расположены в начале и в 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 слова успешно попали в результат. Почему же слова word2 и word4 не попали в список совпадений — ведь перед дефисом есть пробел и после слова есть пробел? Чтобы понять причину, рассмотрим поиск по шагам. Первое слово успешно попадает в результат, т. к. перед дефисом 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 дефис не расположен 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 дефис не расположен в начале строки. Поэтому следующим совпадением будет слово word5, и поиск будет завершен. Таким образом, слова word2 и word4 не попадают в результат, поскольку 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 Match, 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 параметре <Шаблон> указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре <Модификатор> можно указать флаги, ис-

пользуемые в функции `compile()`. **Если соответствие найдено, то возвращается объект** `Match`, **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 `Match`, **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 параметре `<Шаблон>` **указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре** `<Модификатор>` **можно указать флаги, используемые в функции** `compile()`. **Если соответствие найдено, то возвращается объект** `Match`, **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
```

- `fullmatch()` — **выполняет проверку, соответствует ли переданная строка регулярному выражению целиком. Поддержка этого метода появилась в Python 3.4. Формат:**

  `fullmatch(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])`

  Если соответствие найдено, то возвращается объект `Match`, **в противном случае возвращается** `None`. **Примеры:**

```
>>> p = re.compile "[Pp]ython"
>>> print("Найдено" if p.fullmatch("Python") else "Нет")
Найдено
>>> print("Найдено" if p.fullmatch("py") else "Нет")
Нет
```

```
>>> print("Найдено" if p.fullmatch("PythonWare") else "Нет")
Нет
>>> print("Найдено" if p.fullmatch("PythonWare", 0, 6) else "Нет")
Найдено
```

Вместо метода `fullmatch()` **можно воспользоваться функцией** `fullmatch()`. **Формат функции:**

```
re.fullmatch(<Шаблон>, <Строка>[, <Модификатор>])
```

В параметре `<Шаблон>` **указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре** `<Модификатор>` **можно указать флаги, используемые в функции** `compile()`. **Если строка полностью совпадает с шаблоном, возвращается объект** `Match`, **в противном случае возвращается значение** `None`. **Примеры:**

```
>>> p = "[Pp]ython"
>>> print("Найдено" if re.fullmatch(p, "Python") else "Нет")
Найдено
>>> print("Найдено" if re.fullmatch(p, "py") else "Нет")
Нет
```

В качестве примера переделаем нашу программу (см. листинг 4.16) суммирования произвольного количества целых чисел, введенных пользователем, таким образом, чтобы при вводе строки вместо числа программа не завершалась с фатальной ошибкой. Предусмотрим также возможность ввода отрицательных целых чисел (листинг 7.12).

Листинг 7.12. Суммирование неопределенного количества чисел

```
# -*- coding: utf-8 -*-
import re
print("Введите слово 'stop' для получения результата")
summa = 0
p = re.compile(r"^[-]?[0-9]+$", re.S)
while True:
    x = 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)
input()
```

Объект `Match`, **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;**

- `pattern` — **исходная строка с регулярным выражением;**
- `flags` — **комбинация флагов, заданных при создании регулярного выражения в функции** `compile()`**, и флагов, указанных в самом регулярном выражении, в конструкции** `(?aiLmsux)`**;**

- `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`**, если поиск завершился неудачей;**
- `lastgroup` — **возвращает название последней группы или значение** `None`**, если эта группа не имеет имени, или поиск завершился неудачей. Пример:**

```
>>> p = re.compile(r"(?P<num>[0-9]+)(?P<str>[a-z]+)")
>>> m = p.search("123456string 67890text")
>>> m
<_sre.SRE_Match object at 0x00FC9DE8>
>>> 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`**, возвращается фрагмент, полностью соответствующий шаблону. Если указан номер или название группы, возвращается фрагмент, совпадающий с этой группой. Через запятую можно указать несколько номеров или названий групп — в этом случае возвращается кортеж, содержащий фрагменты, что соответствует 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 или название группы>]) — **возвращает индекс начала фрагмента, соответствующего заданной группе. Если параметр не указан, то фрагментом является полное соответствие с шаблоном. Если соответствия нет, то возвращается значение** -1;

- end([<Номер или название группы>]) — **возвращает индекс конца фрагмента, соответствующего заданной группе. Если параметр не указан, то фрагментом является полное соответствие с шаблоном. Если соответствия нет, то возвращается значение** -1;

- span([<Номер или название группы>]) — **возвращает кортеж, содержащий начальный и конечный индексы фрагмента, соответствующего заданной группе. Если параметр не указан, то фрагментом является полное соответствие с шаблоном. 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(<Шаблон>) — **производит замену в строке. Внутри указанного шаблона можно использовать обратные ссылки:** \номер группы, \g<номер группы> **и** \g<название группы>. **Для 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 на соответствие шаблону введенный пользователем адрес электронной почты (листинг 7.13).

Листинг 7.13. Проверка e-mail на соответствие шаблону

```
# -*- coding: utf-8 -*-
import re
email = input("Введите e-mail: ")
pe = r"^([a-z0-9_.-]+)@(([a-z0-9-]+\.)+[a-z]{2,6})$"
p = re.compile(pe, re.I | re.S)
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))
input()
```

Результат выполнения:

```
Введите e-mail: user@mail.ru
E-mail user@mail.ru соответствует шаблону
ящик: user домен: mail.ru
```

7.3. Поиск всех совпадений с шаблоном

Для поиска всех совпадений с шаблоном предназначено несколько функций и методов.

- **Метод `findall()` ищет все совпадения с шаблоном. Формат метода:**

  ```
  findall(<Строка>[, <Начальная позиция>[, <Конечная позиция>]])
  ```

  Если соответствия найдены, 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")
[]
>>> t = r"([0-9]{3} - [0-9]{3} - ([0-9]{2})"
```

```
>>> p = re.compile(t)
>>> 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 параметре `<Шаблон>` **указывается строка с регулярным выражением или скомпилированное регулярное выражение. В параметре** `<Модификатор>` **можно указать флаги, используемые в функции** `compile()`. **Пример:**

```
>>> re.findall(r"[0-9]+", "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']
```

- **Метод** `finditer()` **аналогичен методу** `findall()`, **но возвращает итератор, а не список. На каждой итерации цикла возвращается объект** `Match`. **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 параметре `<Шаблон>` **указывается строка 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

Метод `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 группы`, `\g<номер группы>` **и** `\g<название группы>`. **Для 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 `Match`, **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: utf-8 -*-
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    x = int(m.group(0))
    x += 10
    return "{0}".format(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))
input()
```

Результат выполнения:

```
2018, 2019, 2020, 2021
2018, 2019, 2010, 2011
```

> ***Внимание!***
>
> **Название функции указывается без круглых 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>
       [, flags=0]]
```

В качестве параметра <Шаблон> **можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Поменяем два тега местами, а также изменим регистр букв:**

```
# -*- coding: utf-8 -*-
import re
def repl(m):
    """ Функция для замены. m — объект Match """
    tag1 = m.group("tag1").upper()
    tag2 = m.group("tag2").upper()
    return "<{0}><{1}>".format(tag2, tag1)

p = r"<(?P<tag1>[a-z]+)><(?P<tag2>[a-z]+)>"
print(re.sub(p, repl, "<br><hr>"))
input()
```

Результат выполнения:

```
<HR><BR>
```

Метод 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>
       [, flags=0]])
```

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)
```

Для выполнения замен также можно использовать метод expand(), **поддерживаемый объектом** Match. **Формат метода:**

```
expand(<Шаблон>)
```

Внутри указанного шаблона можно использовать обратные ссылки: \номер группы, \g<номер группы> **и** \g<название группы>. **Пример:**

```
>>> 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>'
```

7.5. Прочие функции и методы

Метод `split()` разбивает строку по шаблону и возвращает список подстрок. Его формат:

```
split(<Исходная строка>[, <Лимит>])
```

Если во втором параметре задано число, то в списке окажется указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент — с остатком строки. Примеры:

```
>>> 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 в строке не найден, то список будет состоять только из одного элемента, содержащего 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>[, flags=0]])
```

В качестве параметра `<Шаблон>` можно указать строку с регулярным выражением или скомпилированное регулярное выражение. Пример:

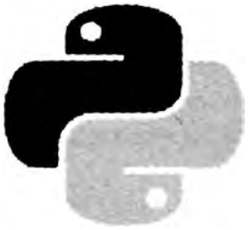
```
>>> p = re.compile(r"[\s,.]+")
>>> re.split(p, "word1, word2\nword3")
['word1', 'word2', 'word3']
>>> re.split(r"[\s,.]+", "word1, word2\nword3")
['word1', 'word2', 'word3']
```

Функция `escape(<Строка>)` позволяет экранировать все специальные символы в строке, полученной от пользователя. Эту строку в дальнейшем можно безопасно использовать внутри регулярного выражения. Пример:

```
>>> print(re.escape(r"[]().*"))
\[\]\(\)\.\*
```

Функция `purge()` выполняет очистку кэша, в котором хранятся промежуточные данные, используемые в процессе выполнения регулярных выражений. Ее рекомендуется вызывать после обработки большого количества регулярных выражений. Результата эта функция не возвращает. Пример:

```
>>> re.purge()
```



ГЛАВА 8

Списки, кортежи, множества и диапазоны

Списки, *кортежи*, *множества* и *диапазоны* — это нумерованные наборы объектов. Каждый элемент набора содержит лишь ссылку на объект — по этой причине они 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 просто 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`) и невхождение (оператор `not 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#7>", line 1, in <module>
    t[0] = 50               # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

- Множества могут быть как изменяемыми, так и неизменяемыми. Их основное отличие от только что рассмотренных типов данных — хранение лишь уникальных значений (неуникальные значения автоматически отбрасываются). Пример:

```
>>> set([0, 1, 1, 2, 3, 3, 4])
{0, 1, 2, 3, 4}
```

- **Что касается диапазонов, то они представляют собой наборы чисел, сформированные на основе заданных начального, конечного значений и величины шага между числами. Их важнейшее преимущество перед всеми остальными наборами объектов — небольшой объем занимаемой оперативной памяти. Пример:**

```
>>> r = range(0, 101, 10)
>>> for i in r: print(i, end = " ")

0 10 20 30 40 50 60 70 80 90 100
```

Рассмотрим все упомянутые типы данных более 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]
```

- **перечислив 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#20>", 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])
```

Точно такая же ситуация возникает при использовании оператора повторения `*`**. Например, в следующей инструкции производится попытка создания двух вложенных списков с помощью оператора** `*`**:**

```
>>> arr = [ [] ] * 2        # Якобы создали два вложенных списка
>>> arr
[[], []]
>>> arr[0].append(5)        # Добавляем элемент
>>> arr                     # Изменились два элемента
[[5], [5]]
```

Создавать вложенные списки следует с помощью метода `append()` **внутри цикла:**

```
>>> arr = []
>>> for i in range(2): arr.append([])

>>> arr
[[], []]
>>> arr[0].append(5); arr
[[5], []]
```

Можно также воспользоваться генераторами списков:

```
>>> arr = [ [] for i in range(2) ]
>>> arr
[[], []]
>>> arr[0].append(5); arr
[[5], []]
```

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
```

Но что же делать, если необходимо создать копию списка? Первый способ заключается в применении операции извлечения среза, второй — в использовании функции `list()`**, а третий — в применении появившегося в Python 3.3 метода** `copy()` **(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[:]
>>>             # или вызовом метода copy(): y = x.copy()
>>> 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', 3.2, '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#86>", line 1, in <module>
    x, y = [1, 2, 3]    # Количество элементов должно совпадать
ValueError: too many values to unpack (expected 2)
```

В Python 3 при позиционном присваивании перед одной из переменных слева от оператора = можно указать звездочку. В этой переменной будет сохраняться список, состоящий из «лишних» элементов. Если таких элементов нет, то список будет пустым:

```
>>> x, y, *z = [1, 2, 3]; x, y, z
(1, 2, [3])
>>> x, y, *z = [1, 2, 3, 4, 5]; x, y, z
(1, 2, [3, 4, 5])
>>> x, y, *z = [1, 2]; x, y, z
(1, 2, [])
>>> *x, y, z = [1, 2]; x, y, z
([], 1, 2)
>>> x, *y, z = [1, 2, 3, 4, 5]; x, y, z
(1, [2, 3, 4], 5)
>>> *z, = [1, 2, 3, 4, 5]; z
[1, 2, 3, 4, 5]
```

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#99>", 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 — чтобы получить положительный индекс, значение вычитается из общего количества элементов списка:

```
>>> 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 не являются обязательными. 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, указанном 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. То есть, 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'
```

```
>>> arr = [ [1, { "a": 10, "b": ["s", 5] } ] ]
>>> arr[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, end=" ")

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, для генерации индексов воспользоваться функцией `range()`. Функция возвращает объект-диапазон, поддерживающий итерации, а с помощью диапазона внутри цикла `for` можно получить текущий индекс. 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 диапазон значений. Если параметр `<Шаг>` не указан, то используется значение `1`. Для примера умножим каждый элемент списка на 2:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] *= 2
print(arr)                       # Результат выполнения: [2, 4, 6, 8]
```

Можно также воспользоваться функцией `enumerate(<Объект>[, start=0])`, 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
```

Кроме того, перебрать элементы можно с помощью цикла `while`. Но в этом случае следует помнить, что цикл `while` работает медленнее цикла `for`. Для 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 range(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, польза здесь также и в том, что 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, — инструкция, выполняемая внутри цикла, находится перед циклом. Обратите внимание и 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. Для 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 внутри не квадратных, а 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. Функции *map()*, *zip()*, *filter()* и *reduce()*

Встроенная функция `map()` позволяет применить функцию к каждому элементу последовательности. Функция имеет следующий формат:

```
map(<Функция>, <Последовательность1>[, ..., <ПоследовательностьN>])
```

Функция `map()` возвращает объект, поддерживающий итерации (а не список, как это было ранее в Python 2). Чтобы получить список в версии Python 3, необходимо результат передать в функцию `list()`.

В качестве параметра `<Функция>` указывается ссылка на функцию (название функции без круглых скобок), которой будет передаваться текущий элемент последовательности. Внутри функции обратного вызова необходимо вернуть новое значение. Для примера прибавим к каждому элементу списка число 10 (листинг 8.3).

Листинг 8.3. Функция `map()`

```
def func(elem):
    """ Увеличение значения каждого элемента списка """
    return elem + 10 # Возвращаем новое значение

arr = [1, 2, 3, 4, 5]
print( list( map(func, 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 после-

довательностях на одинаковом смещении. Просуммируем элементы трех списков (листинг 8.4).

Листинг 8.4. Суммирование элементов трех списков

```
def func(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]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222, 333, 444, 555]
```

Если количество элементов в последовательностях будет разным, то в качестве ограничения выбирается последовательность с минимальным количеством элементов:

```
def func(e1, e2, e3):
    """ Суммирование элементов трех разных списков """
    return e1 + e2 + e3

arr1 = [1, 2, 3, 4, 5]
arr2 = [10, 20]
arr3 = [100, 200, 300, 400, 500]
print( list( map(func, arr1, arr2, arr3) ) )
# Результат выполнения: [111, 222]
```

Встроенная функция `zip()` **на каждой итерации возвращает кортеж, содержащий элементы последовательностей, которые расположены на одинаковом смещении. Функция возвращает объект, поддерживающий итерации (а не список, как это было ранее в Python 2). Чтобы получить список в версии Python 3, необходимо результат передать в функцию** `list()`. **Формат функции:**

```
zip(<Последовательность1>[, ..., <ПоследовательностьN>])
```

Пример:

```
>>> zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
<zip object at 0x00FCAC88>
>>> list(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 результат попадут только элементы, которые существуют во всех последовательностях на одинаковом смещении:

```
>>> list(zip([1, 2, 3], [4, 6], [7, 8, 9, 10]))
[(1, 4, 7), (2, 6, 8)]
```

В качестве еще одного примера переделаем нашу программу (см. 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]
arr = [x + y + z for (x, y, z) in zip(arr1, arr2, arr3)]
print(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. Функция возвращает объект, поддерживающий итерации (а не список или кортеж, как это было ранее в Python 2). Чтобы получить список в версии Python 3, необходимо результат передать в функцию** `list()`**. Пример:**

```
>>> filter(None, [1, 0, None, [], 2])
<filter object at 0x00FD58B0>
>>> list(filter(None, [1, 0, None, [], 2]))
[1, 2]
```

Аналогичная операция с использованием генераторов списков выглядит так:

```
>>> [ i for i in [1, 0, None, [], 2] if i ]
[1, 2]
```

В первом параметре можно указать ссылку на функцию. В эту функцию в качестве параметра будет передаваться текущий элемент последовательности. Если элемент нужно 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 (листинг 8.6).**

Листинг 8.6. Пример использования функции filter()

```
def func(elem):
    return elem >= 0

arr = [-1, 2, -3, 4, 0, -20, 10]
arr = list(filter(func, arr))
print(arr)                              # Результат: [2, 4, 0, 10]
# Использование генераторов списков
arr = [-1, 2, -3, 4, 0, -20, 10]
arr = [ i for i in arr if func(i) ]
print(arr)                              # Результат: [2, 4, 0, 10]
```

Функция `reduce()` **из модуля** `functools` **применяет указанную функцию к парам элементов и накапливает результат. Функция имеет следующий формат:**

```
reduce(<Функция>, <Последовательность>[, <Начальное значение>])
```

В функцию обратного вызова в качестве параметров передаются два элемента: первый элемент будет содержать результат предыдущих вычислений, а второй — значение текущего элемента. Получим сумму всех элементов списка (листинг 8.7).

Листинг 8.7. Пример использования функции reduce()

```
from functools import reduce # Подключаем модуль

def func(x, y):
    print("({0}, {1})".format(x, y), end=" ")
    return x + y

arr = [1, 2, 3, 4, 5]
summa = reduce(func, arr)
# Последовательность: (1, 2) (3, 3) (6, 4) (10, 5)
print(summa) # Результат выполнения: 15
summa = reduce(func, arr, 10)
# Последовательность: (10, 1) (11, 2) (13, 3) (16, 4) (20, 5)
print(summa) # Результат выполнения: 25
summa = reduce(func, [], 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 или оператора +=:

```
>>> arr = [1, 2, 3]
>>> arr + [4, 5, 6]             # Возвращает новый список
[1, 2, 3, 4, 5, 6]
```

```
>>> arr += [4, 5, 6]            # Изменяет текущий список
>>> arr
[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. Если элемента с указанным индексом нет, или список пустой, возбуждается исключение `IndexError`. 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#3>", line 1, in <module>
    arr.remove(5)    # Такого элемента нет
ValueError: list.remove(x): x not in list
```

- `clear()` — **удаляет все элементы списка, очищая его. Никакого результата при этом не возвращается. Поддержка этого метода появилась в Python 3.3. Пример:**

```
>>> arr = [1, 2, 3, 1, 1]
>>> arr.clear()
>>> arr
[]
```

Если необходимо удалить все повторяющиеся элементы списка, то можно преобразовать список во множество, а затем множество обратно преобразовать в список. Обратите внимание на то, что список должен содержать только неизменяемые объекты (например, числа, строки или кортежи). 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
{1, 2, 3}
>>> arr = list(s)         # Преобразуем множество в список
>>> arr                   # Все повторы были удалены
[1, 2, 3]
```

8.8. Поиск элемента в списке и получение сведений о значениях, входящих в список

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`. **Аналогичный оператор** `not in` **выполняет проверку на *невхождение* элемента в список: если элемент отсутствует в списке, возвращается** `True`, **в противном случае —** `False`. **Примеры:**

```
>>> 2 in [1, 2, 3, 4, 5], 6 in [1, 2, 3, 4, 5] # Проверка на вхождение
(True, False)
```

```
>>> 2 not in [1, 2, 3, 4, 5], 6 not in [1, 2, 3, 4, 5] # Проверка на невхождение
(False, True)
```

Тем не менее, оба этих оператора не дают 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 и до конца 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#16>", line 1, in <module>
    arr.index(3)
ValueError: 3 is 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)
<list_reverseiterator object at 0x00FD5150>
>>> list(reversed(arr))              # Использование функции list()
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> for i in reversed(arr): print(i, end=" ") # 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()`. Он имеет следующий формат:

```
sort([key=None][, reverse=False])
```

Все параметры не являются обязательными. 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]
```

Надо заметить, что стандартная сортировка зависит от регистра символов (листинг 8.8).

Листинг 8.8. Стандартная сортировка

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort()
for i in arr:
    print(i, end=" ")
# 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

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=str.lower)                 # Указываем метод lower()
for i in arr:
    print(i, end=" ")
# Результат выполнения: единица1 Единица2 Единый
```

В параметре key можно указать функцию, выполняющую какое-либо действие над каждым элементом списка. В качестве единственного параметра она должна принимать значение очередного элемента списка, а в качестве результата — возвращать результат действий над ним. Этот результат будет участвовать в процессе сортировки, но значения самих элементов списка не изменятся.

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>[, 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 8.10.

Листинг 8.10. 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"]
>>> sorted(arr, key=str.lower)
['единица1', 'Единица2', 'Единый']
```

8.12. Заполнение списка числами

Создать список, содержащий диапазон чисел, можно с помощью функции range(). Эта функция возвращает диапазон, который преобразуется в список вызовом функции list(). Функция range() имеет следующий формат:

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

Первый параметр задает начальное значение — если он не указан, используется значение 0. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон. Если параметр <Шаг> не указан, то используется значение 1. В качестве примера заполним список числами от 0 до 10:

```
>>> list(range(11))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Создадим список, состоящий из диапазона чисел от 1 до 15:

```
>>> list(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:

```
>>> list(range(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(range(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#69>", line 1, in <module>
    " – ".join(arr)
TypeError: sequence item 3: expected str instance, 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, как и списки, являются упорядоченными последовательностями элементов. Они 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. Пример:

```
>>> 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!
<class 'int'>
>>> t = ("str"); type(t)   # Получили строку, а не кортеж!
<class 'str'>
```

Четвертая строка в исходном 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`) и невхождение (оператор `not in`). Пример:

```
>>> 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, как уже неоднократно отмечалось, 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#95>", line 1, in <module>
    t[0] = 50             # Изменить элемент по индексу нельзя!
TypeError: 'tuple' object does not support item assignment
```

Кортежи поддерживают уже знакомые нам по спискам функции `len()`, `min()`, `max()`, **методы** `index()` **и** `count()`. **Примеры:**

```
>>> t = (1, 2, 3)           # Создаем кортеж
>>> len(t)                  # Получаем количество элементов
3
>>> t = (1, 2, 1, 2, 1)
>>> t.index(1), t.index(2) # Ищем элементы в кортеже
(0, 1)
```

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()` — объединяет два множества:

```
>>> 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()` — вычисляет разницу 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 элементы, которые присутствуют в обоих этих множествах:

```
>>> 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 те, что присутствуют в обоих этих множествах:

```
>>> 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
(True, False)
```

- not in — проверка отсутствия элемента во множестве:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 in s, 12 in s
(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)
```

- `a.isdisjoint(b)` — **проверяет, являются ли множества** `a` **и** `b` **полностью разными, т. е. не содержащими ни одного совпадающего элемента:**

```
>>> s = set([1, 2, 3])
>>> s.isdisjoint(set([4, 5, 6]))
True
>>> s.isdisjoint(set([1, 3, 5]))
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. Если указанный элемент не существует, никакого исключения не возбуждается:**

```
>>> 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([])
```

Помимо генераторов списков и генераторов словарей, язык Python 3 поддерживает *генераторы множеств*. Синтаксис генераторов множеств похож на синтаксис генераторов списков, но выражение заключается в фигурные скобки, а не в квадратные. Так как результатом является множество, все повторяющиеся элементы будут удалены. Пример:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3]}
{1, 2, 3}
```

Генераторы множеств могут иметь сложную структуру. Например, состоять из нескольких вложенных циклов `for` **и (или) содержать оператор ветвления** `if` **после цикла. Создадим из элементов исходного списка множество, содержащее только уникальные элементы с четными значениями:**

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3] if x % 2 == 0}
{2}
```

Язык Python поддерживает 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()`.

8.16. Диапазоны

Диапазоны, как следует из самого их названия, — это диапазоны целых чисел с заданными начальным и конечным значением и шагом (промежутком между соседними числами). Как и списки, кортежи и множества, диапазоны представляют собой последовательности и, подобно кортежам, являются неизменяемыми.

Важнейшим преимуществом диапазонов перед другими видами последовательностей является их компактность — вне зависимости от количества входящих в него элементов-чисел, диапазон всегда отнимает один и тот же объем оперативной памяти. Однако в диапазон могут входить лишь последовательно стоящие друг за другом числа — сформировать диапазон на основе произвольного набора чисел или данных другого типа, даже чисел с плавающей точкой, невозможно.

Диапазоны чаще всего используются для проверки вхождения значения в какой-либо интервал и для организации циклов.

Для создания диапазона применяется функция `range()`:

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

Первый параметр задает начальное значение — если он не указан, используется значение `0`. Во втором параметре указывается конечное значение. Следует заметить, что это значение не входит в возвращаемый диапазон. Если параметр `<Шаг>` не указан, то используется значение `1`. Примеры:

```
>>> r = range(1, 10)
>>> for i in r: print(i, end = " ")
1 2 3 4 5 6 7 8 9
>>> r = range(10, 110, 10)
>>> for i in r: print(i, end = " ")
10 20 30 40 50 60 70 80 90 100
>>> r = range(10, 1, -1)
>>> for i in r: print(i, end = " ")
10 9 8 7 6 5 4 3 2
```

Преобразовать диапазон в список, кортеж, обычное или неизменяемое множество можно с помощью функций `list()`, `tuple()`, `set()` или `frozenset()` соответственно:

```
>>> list(range(1, 10))              # Преобразуем в список
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(1, 10))             # Преобразуем в кортеж
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> set(range(1, 10))               # Преобразуем в множество
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Множества поддерживают доступ к элементу по индексу, получение среза (в результате возвращается также диапазон), проверку на вхождение и невхождение, функции `len()`, `min()`, `max()`, методы `index()` и `count()`. Примеры:

```
>>> r = range(1, 10)
>>> r[2], r[-1]
(3, 9)
>>> r[2:4]
range(3, 5)
>>> 2 in r, 12 in r
(True, False)
>>> 3 not in r, 13 not in r
(False, True)
>>> len(r), min(r), max(r)
(9, 1, 9)
>>> r.index(4), r.count(4)
(3, 1)
```

В Python 3.3 появилась поддержка операторов, позволяющих сравнить два диапазона:

- `==` — возвращает `True`, если диапазоны равны, и `False` в противном случае. Диапазоны считаются равными, если они содержат одинаковые последовательности чисел. Примеры:

```
>>> range(1, 10) == range(1, 10, 1)
True
>>> range(1, 10, 2) == range(1, 11, 2)
True
>>> range(1, 10, 2) == range(1, 12, 2)
False
```

- `!=` — возвращает `True`, если диапазоны не равны, и `False` в противном случае:

```
>>> range(1, 10, 2) != range(1, 12, 2)
True
>>> range(1, 10) != range(1, 10, 1)
False
```

А в Python 3.4 диапазоны стали поддерживать свойства `start`, `stop` и `step`, возвращающие, соответственно, начальную, конечную границы диапазона и его шаг:

```
>>> r = range(1, 10)
>>> r.start, r.stop, r.step
(1, 10, 1)
```

8.17. Модуль *itertools*

Модуль `itertools` содержит функции, позволяющие генерировать различные последовательности на основе других последовательностей, производить фильтрацию элементов и др. Все функции возвращают объекты, поддерживающие итерации. Прежде чем использовать функции, необходимо подключить модуль с помощью инструкции:

```
import itertools
```

8.17.1. Генерация неопределенного количества значений

Для генерации неопределенного количества значений предназначены следующие функции:

- `count([start=0][, step=1])` — создает бесконечно нарастающую последовательность значений. Начальное значение задается параметром `start`, а шаг — параметром `step`. Пример:

```
>>> import itertools
>>> for i in itertools.count():
        if i > 10: break
        print(i, end=" ")

0 1 2 3 4 5 6 7 8 9 10
>>> list(zip(itertools.count(), "абвгд"))
[(0, 'а'), (1, 'б'), (2, 'в'), (3, 'г'), (4, 'д')]
>>> list(zip(itertools.count(start=2, step=2), "абвгд"))
[(2, 'а'), (4, 'б'), (6, 'в'), (8, 'г'), (10, 'д')]
```

- `cycle(<Последовательность>)` — на каждой итерации возвращается очередной элемент последовательности. Когда будет достигнут конец последовательности, перебор начнется сначала, и так бесконечно. Пример:

```
>>> n = 1
>>> for i in itertools.cycle("абв"):
        if n > 10: break
        print(i, end=" ")
        n += 1

а б в а б в а б в а
>>> list(zip(itertools.cycle([0, 1]), "абвгд"))
[(0, 'а'), (1, 'б'), (0, 'в'), (1, 'г'), (0, 'д')]
```

- `repeat(<Объект>[, <Количество повторов>])` — возвращает объект указанное количество раз. Если количество повторов не указано, то объект возвращается бесконечно. Пример:

```
>>> list(itertools.repeat(1, 10))
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> list(zip(itertools.repeat(5), "абвгд"))
[(5, 'а'), (5, 'б'), (5, 'в'), (5, 'г'), (5, 'д')]
```

8.17.2. Генерация комбинаций значений

Получить различные комбинации значений позволяют следующие функции:

- `combinations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. При этом элементы в кортеже гарантированно будут разными. Формат функции:

  ```
  combinations(<Последовательность>, <Количество элементов>)
  ```

  Примеры:

  ```
  >>> import itertools
  >>> list(itertools.combinations('абвг', 2))
  [('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'в'), ('б', 'г'),
   ('в', 'г')]
  >>> ["".join(i) for i in itertools.combinations('абвг', 2)]
  ['аб', 'ав', 'аг', 'бв', 'бг', 'вг']
  >>> list(itertools.combinations('вгаб', 2))
  [('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'а'), ('г', 'б'),
   ('а', 'б')]
  >>> list(itertools.combinations('абвг', 3))
  [('а', 'б', 'в'), ('а', 'б', 'г'), ('а', 'в', 'г'),
   ('б', 'в', 'г')]
  ```

- `combinations_with_replacement()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. При этом кортеж может содержать одинаковые элементы. Формат функции:

  ```
  combinations_with_replacement(<Последовательность>,
                                <Количество элементов>)
  ```

  Примеры:

  ```
  >>> list(itertools.combinations_with_replacement('абвг', 2))
  [('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'б'),
   ('б', 'в'), ('б', 'г'), ('в', 'в'), ('в', 'г'), ('г', 'г')]
  >>> list(itertools.combinations_with_replacement('вгаб', 2))
  [('в', 'в'), ('в', 'г'), ('в', 'а'), ('в', 'б'), ('г', 'г'),
   ('г', 'а'), ('г', 'б'), ('а', 'а'), ('а', 'б'), ('б', 'б')]
  ```

- `permutations()` — на каждой итерации возвращает кортеж, содержащий комбинацию из указанного количества элементов. Если количество элементов не указано, то используется длина последовательности. Формат функции:

  ```
  permutations(<Последовательность>[, <Количество элементов>])
  ```

  Примеры:

  ```
  >>> list(itertools.permutations('абвг', 2))
  [('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'), ('б', 'в'),
   ('б', 'г'), ('в', 'а'), ('в', 'б'), ('в', 'г'), ('г', 'а'),
   ('г', 'б'), ('г', 'в')]
  >>> ["".join(i) for i in itertools.permutations('абвг')]
  ['абвг', 'абгв', 'авбг', 'авгб', 'агбв', 'агвб', 'бавг',
  ```

```
 'багв', 'бваг', 'бвга', 'бгав', 'бгва', 'вабг', 'вагб',
 'вбаг', 'вбга', 'вгаб', 'вгба', 'габв', 'гавб', 'гбав',
 'гбва', 'гваб', 'гвба']
```

- `product()` — **на каждой итерации возвращает кортеж, содержащий комбинацию из элементов одной или нескольких последовательностей. Формат функции:**

```
product(<Последовательность1>[,..., <ПоследовательностьN>][,
        repeat=1])
```

Примеры:

```
>>> from itertools import product
>>> list(product('абвг', repeat=2))
[('а', 'а'), ('а', 'б'), ('а', 'в'), ('а', 'г'), ('б', 'а'),
 ('б', 'б'), ('б', 'в'), ('б', 'г'), ('в', 'а'), ('в', 'б'),
 ('в', 'в'), ('в', 'г'), ('г', 'а'), ('г', 'б'), ('г', 'в'),
 ('г', 'г')]
>>> ["".join(i) for i in product('аб', 'вг', repeat=1)]
['ав', 'аг', 'бв', 'бг']
>>> ["".join(i) for i in product('аб', 'вг', repeat=2)]
['авав', 'аваг', 'авбв', 'авбг', 'агав', 'агаг', 'агбв',
 'агбг', 'бвав', 'бваг', 'бвбв', 'бвбг', 'бгав', 'бгаг',
 'бгбв', 'бгбг']
```

8.17.3. Фильтрация элементов последовательности

Для фильтрации элементов последовательности предназначены следующие функции:

- `filterfalse(<Функция>, <Последовательность>)` — **возвращает элементы последовательности (по одному на каждой итерации), для которых функция, указанная в первом параметре, вернет значение** `False`. **Примеры:**

```
>>> import itertools
>>> def func(x): return x > 3

>>> list(itertools.filterfalse(func, [4, 5, 6, 0, 7, 2, 3]))
[0, 2, 3]
>>> list(filter(func, [4, 5, 6, 0, 7, 2, 3]))
[4, 5, 6, 7]
```

Если в первом параметре вместо названия функции указать значение `None`, **то каждый элемент последовательности будет проверен на соответствие значению** `False`. **Если элемент в логическом контексте возвращает значение** `True`, **то он не войдет в возвращаемый результат. Примеры:**

```
>>> list(itertools.filterfalse(None, [0, 5, 6, 0, 7, 0, 3]))
[0, 0, 0]
>>> list(filter(None, [0, 5, 6, 0, 7, 0, 3]))
[5, 6, 7, 3]
```

- `dropwhile(<Функция>, <Последовательность>)` — **возвращает элементы последовательности (по одному на каждой итерации), начиная с элемента, для которого функция, указанная в первом параметре, вернет значение** `False`.

Примеры:

```
>>> def func(x): return x > 3

>>> list(itertools.dropwhile(func, [4, 5, 6, 0, 7, 2, 3]))
[0, 7, 2, 3]
>>> list(itertools.dropwhile(func, [4, 5, 6, 7, 8]))
[]
>>> list(itertools.dropwhile(func, [1, 2, 4, 5, 6, 7, 8]))
[1, 2, 4, 5, 6, 7, 8]
```

- `takewhile(<Функция>, <Последовательность>)` — **возвращает элементы последовательности (по одному на каждой итерации), пока не встретится элемент, для которого функция, указанная в первом параметре, вернет значение** `False`. **Примеры:**

```
>>> def func(x): return x > 3

>>> list(itertools.takewhile(func, [4, 5, 6, 0, 7, 2, 3]))
[4, 5, 6]
>>> list(itertools.takewhile(func, [4, 5, 6, 7, 8]))
[4, 5, 6, 7, 8]
>>> list(itertools.takewhile(func, [1, 2, 4, 5, 6, 7, 8]))
[]
```

- `compress()` — **производит фильтрацию последовательности, указанной в первом параметре. Элемент возвращается, только если соответствующий элемент (с таким же индексом) из второй последовательности трактуется как истина. Сравнение заканчивается, когда достигнут конец одной из последовательностей. Формат функции:**

```
compress(<Фильтруемая последовательность>,
         <Последовательность логических значений>)
```

Примеры:

```
>>> list(itertools.compress('абвгде', [1, 0, 0, 0, 1, 1]))
['а', 'д', 'е']
>>> list(itertools.compress('абвгде', [True, False, True]))
['а', 'в']
```

8.17.4. Прочие функции

Помимо функций, которые мы рассмотрели в предыдущих подразделах, модуль `itertools` **содержит несколько дополнительных функций:**

- `islice()` — **на каждой итерации возвращает очередной элемент последовательности. Поддерживаются форматы:**

```
islice(<Последовательность>, <Конечная граница>)
```

и

```
islice(<Последовательность>, <Начальная граница>, <Конечная граница>[, <Шаг>])
```

Если `<Шаг>` **не указан, будет использовано значение** `1`. **Примеры:**

```
>>> list(itertools.islice("абвгдежж", 3))
['а', 'б', 'в']
```

```
>>> list(itertools.islice("абвгдезж", 3, 6))
['г', 'д', 'е']
>>> list(itertools.islice("абвгдезж", 3, 6, 2))
['г', 'е']
```

- `starmap(<Функция>, <Последовательность>)` — **формирует последовательность на основании значений, возвращенных указанной функцией. Исходная последовательность должна содержать в качестве элементов кортежи — именно над элементами этих кортежей функция и станет вычислять значения, которые войдут в генерируемую последовательность. Примеры суммирования значений:**

```
>>> import itertools
>>> def func1(x, y): return x + y

>>> list(itertools.starmap(func1, [(1, 2), (4, 5), (6, 7)]))
[3, 9, 13]
>>> def func2(x, y, z): return x + y + z

>>> list(itertools.starmap(func2, [(1, 2, 3), (4, 5, 6)]))
[6, 15]
```

- `zip_longest()` — **на каждой итерации возвращает кортеж, содержащий элементы последовательностей, которые расположены на одинаковом смещении. Если последовательности имеют разное количество элементов, то вместо отсутствующего элемента вставляется объект, указанный в параметре** `fillvalue`. **Формат функции:**

```
zip_longest(<Последовательность1>[, ..., <ПоследовательностьN>]
            [, fillvalue=None])
```

Примеры:

```
>>> list(itertools.zip_longest([1, 2, 3], [4, 5, 6]))
[(1, 4), (2, 5), (3, 6)]
>>> list(itertools.zip_longest([1, 2, 3], [4]))
[(1, 4), (2, None), (3, None)]
>>> list(itertools.zip_longest([1, 2, 3], [4], fillvalue=0))
[(1, 4), (2, 0), (3, 0)]
>>> list(zip([1, 2, 3], [4]))
[(1, 4)]
```

- `accumulate(<Последовательность>[, <функция>])` — **на каждой итерации возвращает результат, полученный выполнением определенного действия над текущим элементом и результатом, полученным на предыдущей итерации. Выполняемая операция задается параметром** `<Функция>`, **а если он не указан, выполняется операция сложения. Функция, выполняющая операцию, должна принимать два параметра и возвращать результат. На первой итерации всегда возвращается первый элемент переданной последовательности. Пример:**

```
>>> # Выполняем сложение
>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6]))
[1, 3, 6, 10, 15, 21]
>>> # [1, 1+2, 3+3, 6+4, 10+5, 15+6]
```

```
>>> # Выполняем умножение
>>> def func(x, y): return x * y

>>> list(itertools.accumulate([1, 2, 3, 4, 5, 6], func))
[1, 2, 6, 24, 120, 720]
>>> # [1, 1*2, 2*3, 6*4, 24*5, 120*6]
```

- `chain()` — **на каждой итерации возвращает элементы сначала из первой последовательности, затем из второй и т. д. Формат функции:**

  `chain(<Последовательность1>[, ..., <ПоследовательностьN>])`

  Примеры:

```
>>> arr1, arr2, arr3 = [1, 2, 3], [4, 5], [6, 7, 8, 9]
>>> list(itertools.chain(arr1, arr2, arr3))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(itertools.chain("abc", "defg", "hij"))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> list(itertools.chain("abc", ["defg", "hij"]))
['a', 'b', 'c', 'defg', 'hij']
```

- `chain.from_iterable(<Последовательность>)` — **аналогична функции** `chain()`**, но принимает одну последовательность, каждый элемент которой считается отдельной последовательностью. Примеры:**

```
>>> list(itertools.chain.from_iterable(["abc", "defg", "hij"]))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

- `tee(<Последовательность>[, <Количество>])` — **возвращает кортеж, содержащий несколько одинаковых итераторов для последовательности. Если второй параметр не указан, то возвращается кортеж из двух итераторов. Примеры:**

```
>>> arr = [1, 2, 3]
>>> itertools.tee(arr)
(<itertools.tee object at 0x00FD8760>,
<itertools.tee object at 0x00FD8738>)
>>> itertools.tee(arr, 3)
(<itertools.tee object at 0x00FD8710>,
<itertools.tee object at 0x00FD87D8>,
<itertools.tee object at 0x00FD87B0>)
>>> list(itertools.tee(arr)[0])
[1, 2, 3]
>>> list(itertools.tee(arr)[1])
[1, 2, 3]
```

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. Равно 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
>>> list(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» записываются 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, переданной первым параметром, а их значениями — величина, переданная вторым параметром. 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`. То есть, 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#49>", line 1, in <module>
    d["c"]             # Обращение к несуществующему элементу
KeyError: 'c'
```

Проверить существование ключа можно с помощью оператора `in`. Если ключ найден, то возвращается значение `True`, в противном случае — `False`. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d            # Ключ существует
True
>>> "c" in d            # Ключ не существует
False
```

Проверить, отсутствует ли какой-либо ключ в словаре, позволит оператор `not in`. Если ключ отсутствует, возвращается `True`, иначе — `False`. Примеры:

```
>>> d = { "a": 1, "b": 2 }
>>> "c" not in d           # Ключ не существует
True
>>> "a" not in d           # Ключ существует
False
```

Метод `get(<Ключ>[, <Значение по умолчанию>])` позволяет избежать возбуждения исключения `KeyError` при отсутствии в словаре указанного ключа. Если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается `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 в словаре создается 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, 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 словаря можно с помощью цикла `for`**, хотя словари и не являются последовательностями. Для примера выведем элементы словаря двумя способами. Первый способ использует метод** `keys()`**, возвращающий объект с ключами словаря. Во втором 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("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (y => 2) (x => 1) (z => 3)
print()                          # Вставляем символ перевода строки
for key in d:                    # Словари также поддерживают итерации
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (y => 2) (x => 1) (z => 3)
```

Поскольку 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 = list(d.keys())              # Получаем список ключей
k.sort()                        # Сортируем список ключей
for key in k:
    print("({0} => {1})".format(key, d[key]), end=" ")
# 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("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

Так как на каждой итерации возвращается ключ словаря, функции `sorted()` **можно сразу передать объект словаря, а не результат выполнения метода** `keys()`:

```
d = {"x": 1, "y": 2, "z": 3}
for key in sorted(d):
    print("({0} => {1})".format(key, d[key]), end=" ")
# Выведет: (x => 1) (y => 2) (z => 3)
```

9.4. Методы для работы со словарями

Для работы со словарями предназначены следующие методы:

- `keys()` — **возвращает объект** `dict_keys`, **содержащий все ключи словаря. Этот объект поддерживает итерации, а также операции над множествами. Пример:**

```
>>> d1, d2 = { "a": 1, "b": 2 }, { "a": 3, "c": 4, "d": 5 }
>>> d1.keys(), d2.keys() # Получаем объект dict_keys
(dict_keys(['a', 'b']), dict_keys(['a', 'c', 'd']))
>>> list(d1.keys()), list(d2.keys()) # Получаем список ключей
(['a', 'b'], ['a', 'c', 'd'])
>>> for k in d1.keys(): print(k, end=" ")

a b
>>> d1.keys() | d2.keys()    # Объединение
{'a', 'c', 'b', 'd'}
>>> d1.keys() - d2.keys()    # Разница
{'b'}
>>> d2.keys() - d1.keys()    # Разница
{'c', 'd'}
>>> d1.keys() & d2.keys()    # Одинаковые ключи
{'a'}
>>> d1.keys() ^ d2.keys()    # Уникальные ключи
{'c', 'b', 'd'}
```

- `values()` — **возвращает объект** `dict_values`, **содержащий все значения словаря. Этот объект поддерживает итерации. Примеры:**

```
>>> d = { "a": 1, "b": 2 }
>>> d.values()                  # Получаем объект dict_values
dict_values([1, 2])
>>> list(d.values())            # Получаем список значений
[1, 2]
>>> [ v for v in d.values() ]
[1, 2]
```

- `items()` — **возвращает объект** `dict_items`, **содержащий все ключи и значения в виде кортежей. Этот объект поддерживает итерации. Примеры:**

```
>>> d = { "a": 1, "b": 2 }
>>> d.items()                   # Получаем объект dict_items
dict_items([('a', 1), ('b', 2)])
>>> list(d.items())             # Получаем список кортежей
[('a', 1), ('b', 2)]
```

- `<Ключ> in <Словарь>` — **проверяет существование указанного ключа в словаре. Если ключ найден, то возвращается значение** `True`, **в противном случае —** `False`. **Примеры:**

```
>>> d = { "a": 1, "b": 2 }
>>> "a" in d                    # Ключ существует
True
>>> "c" in d                    # Ключ не существует
False
```

- `<Ключ> not in <Словарь>` — **проверяет отсутствие указанного ключа в словаре. Если такового ключа нет, то возвращается значение** `True`, **в противном случае —** `False`. **Примеры:**

```
>>> d = { "a": 1, "b": 2 }
>>> "c" not in d                    # Ключ не существует
True
>>> "a" not in d                    # Ключ существует
False
```

- `get(<Ключ>[, <Значение по умолчанию>])` — **если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается** `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 создает в словаре 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, возбуждается исключение** KeyError. **Примеры:**

```
>>> 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#40>", 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. **Примеры:**

```
>>> 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#45>", 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`.

9.5. Генераторы словарей

Помимо генераторов списков, язык Python 3 поддерживает генераторы словарей. Синтаксис генераторов словарей похож на синтаксис генераторов списков, но имеет два отличия:

- **выражение заключается в фигурные скобки, а не в квадратные;**
- **внутри выражения перед циклом** `for` **указываются два значения через двоеточие, а не одно. Значение, расположенное слева от двоеточия, становится ключом, а значение, расположенное справа от двоеточия, — значением элемента.**

Пример:

```
>>> keys = ["a", "b"]          # Список с ключами
>>> values = [1, 2]            # Список со значениями
>>> {k: v for (k, v) in zip(keys, values)}
{'a': 1, 'b': 2}
>>> {k: 0 for k in keys}
{'a': 0, 'b': 0}
```

Генераторы словарей могут иметь сложную структуру. Например, состоять из нескольких вложенных циклов `for` **и (или) содержать оператор ветвления** `if` **после цикла. Создадим новый словарь, содержащий только элементы с четными значениями, из исходного словаря:**

```
>>> d = { "a": 1, "b": 2, "c": 3, "d": 4 }
>>> {k: v for (k, v) in d.items() if v % 2 == 0}
{'b': 2, 'd': 4}
```

ГЛАВА 10

Работа с датой и временем

Для работы с датой и временем в языке Python предназначены следующие модули:

- `time` — позволяет получить текущие дату и время, а также произвести их форматированный вывод;
- `datetime` — позволяет манипулировать 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 текущих даты и времени

Получить текущие 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
1428057929.227704
```

- `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. Примеры:

```
>>> 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=2015, tm_mon=4, tm_mday=3, tm_hour=10, tm_min=48,
tm_sec=10, tm_wday=4, tm_yday=93, tm_isdst=0)
>>> time.gmtime(1428057929.0)         # Дата 03-04-2015
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=3, tm_hour=10, tm_min=45,
tm_sec=29, tm_wday=4, tm_yday=93, tm_isdst=0)
```

Получить значение конкретного атрибута можно, указав его название или индекс внутри объекта:

```
>>> d = time.gmtime()
>>> d.tm_year, d[0]
(2015, 2015)
>>> tuple(d)          # Преобразование в кортеж
(2015, 4, 3, 10, 50, 34, 4, 93, 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. Примеры:**

```
>>> time.localtime()                # Текущая дата и время
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=3, tm_hour=13, tm_min=51,
tm_sec=22, tm_wday=4, tm_yday=93, tm_isdst=0)
>>> time.localtime(1428057929.0)  # Дата 03-04-2015
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=3, tm_hour=13, tm_min=45,
tm_sec=29, tm_wday=4, tm_yday=93, 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:**

```
>>> d = time.localtime(1428057929.0)
>>> time.mktime(d)
1428057929.0
>>> tuple(time.localtime(1428057929.0))
(2015, 4, 3, 13, 45, 29, 4, 93, 0)
>>> time.mktime((2015, 4, 3, 13, 45, 29, 4, 93, 0))
1428057929.0
>>> 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 (указаны пары вида «имя атрибута — индекс — описание»):**

- `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 текущие дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 10.1).

Листинг 10.1. Вывод текущих даты и времени

```
# -*- coding: utf-8 -*-
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] ) )
input()
```

Примерный результат выполнения:

```
Сегодня:
пятница 3 апреля 2015 13:59:30
03.04.2015
```

10.2. Форматирование даты и времени

Форматирование даты и времени выполняют 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, будут выведены текущие 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. Примеры:**

```
>>> import time
>>> time.strftime("%d.%m.%Y") # Форматирование даты
'03.04.2015'
>>> time.strftime("%H:%M:%S") # Форматирование времени
'14:01:34'
>>> time.strftime("%d.%m.%Y", time.localtime(1321954972.0))
'22.11.2011'
```

- `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`. **Если строка формата не указана, используется строка** `"%a %b %d %H:%M:%S %Y"`. **Примеры:**

```
>>> time.strptime("Fri Apr 03 14:01:34 2015")
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=3, tm_hour=14, tm_min=1,
tm_sec=34, tm_wday=4, tm_yday=94, tm_isdst=-1)
```

```
>>> time.strptime("03.04.2015", "%d.%m.%Y")
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=3, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=4, tm_yday=93, tm_isdst=-1)
>>> time.strptime("03-04-2015", "%d.%m.%Y")
... Фрагмент опущен ...
ValueError: time data '03-04-2015' does not match format '%d.%m.%Y'
```

- `asctime([<Объект struct_time>])` — **возвращает строку формата** `"%a %b %d %H:%M:%S %Y"`. **Если параметр не указан, будут выведены текущие дата и время. Если в параметре указан объект** `struct_time` **или кортеж из девяти элементов, то дата будет соответствовать указанному значению. Примеры:**

```
>>> time.asctime()                                  # Текущая дата
'Fri Apr  3 14:06:12 2015'
>>> time.asctime(time.localtime(1321954972.0))  # Дата в прошлом
'Tue Nov 22 12:42:52 2011'
```

- `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
'Fri Apr  3 14:06:12 2015'
>>> time.ctime(1321954972.0)        # Дата в прошлом
'Tue Nov 22 12:42:52 2011'
```

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,** `"2011"`);
- `%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
03.04.2015
>>> print(time.strftime("%X"))    # Представление времени
14:10:19
>>> print(time.strftime("%c"))    # Дата и время
03.04.2015 14:10:19
```

- `%Z` — **название часового пояса или пустая строка (например,** `"Московское время"`**,** `"UTC"`**);**
- `%%` — **символ** `"%"`**.**

В качестве примера выведем текущие дату и время с помощью функции `strftime()` **(листинг 10.2).**

Листинг 10.2. Форматирование даты и времени

```
# -*- coding: utf-8 -*-
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))
input()
```

Примерный результат выполнения:

```
Сегодня:
пятница 03 апр 2015 14:12:09
03.04.2015
```

10.3. «Засыпание» скрипта

Функция `sleep(<Время в секундах>)` **из модуля** `time` **прерывает выполнение скрипта на указанное время, по истечении которого скрипт продолжит работу. В качестве параметра можно указать целое или вещественное число. Пример:**

```
>>> import time                  # Подключаем модуль time
>>> time.sleep(5)                # "Засыпаем" на 5 секунд
```

10.4. Модуль *datetime*. Манипуляции датой и временем

Модуль `datetime` позволяет манипулировать датой и временем. Например, выполнять 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 инструкции:

```
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 выполнять операции над датами — например: складывать, вычитать, сравнивать и др. Конструктор класса имеет следующий формат:

```
timedelta([days][, seconds][, microseconds][, milliseconds][, minutes]
          [, hours][, weeks])
```

Все параметры не являются обязательными и по умолчанию имеют значение `0`. Первые три параметра считаются 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 атрибутов:

- `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')
```

Получить результат в секундах позволяет метод `total_seconds()`:

```
>>> d = datetime.timedelta(minutes=1)
>>> d.total_seconds()
60.0
```

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))
>>> d2 / d1                                       # Деление
3.5
>>> d1 / 2, d2 / 2.5                              # Деление
(datetime.timedelta(1), datetime.timedelta(2, 69120))
>>> d2 // d1                                      # Деление
3
>>> d1 // 2, d2 // 2                              # Деление
(datetime.timedelta(1), datetime.timedelta(3, 43200))
>>> d2 % d1                                       # Остаток
datetime.timedelta(1)
```

```
>>> 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)
```

Также можно получать строковое представление объекта `timedelta` **с помощью функций** `str()` **и** `repr()`:

```
>>> d = datetime.timedelta(hours = 25, minutes = 5, seconds = 27)
>>> str(d)
'1 day, 1:05:27'
>>> repr(d)
'datetime.timedelta(1, 3927)'
```

Поддерживаются следующие атрибуты класса:

- `min` — **минимальное значение, которое может иметь объект** `timedelta`;
- `max` — **максимальное значение, которое может иметь объект** `timedelta`;
- `resolution` — **минимальное возможное различие между значениями** `timedelta`.

Выведем значения этих атрибутов:

```
>>> datetime.timedelta.min
datetime.timedelta(-999999999)
>>> datetime.timedelta.max
datetime.timedelta(999999999, 86399, 999999)
>>> datetime.timedelta.resolution
datetime.timedelta(0, 0, 1)
```

10.4.2. Класс *date*

Класс `date` **из модуля** `datetime` **позволяет выполнять операции над датами. Конструктор класса имеет следующий формат:**

```
date(<Год>, <Месяц>, <День>)
```

Все параметры являются обязательными. В параметрах можно указать следующий диапазон значений:

- <Год> — в виде числа, расположенного в диапазоне между значениями, хранящимися в константах MINYEAR и MAXYEAR класса datetime (о нем речь пойдет позже). 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(2015, 4, 3)
datetime.date(2015, 4, 3)
>>> datetime.date(2015, 13, 3)   # Неправильное значение для месяца
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.date(2015, 4, 3)
>>> repr(d), str(d)
('datetime.date(2015, 4, 3)', '2015-04-03')
```

Для создания объекта класса date также можно воспользоваться следующими методами этого класса:

- today() — возвращает текущую дату:

```
>>> datetime.date.today() # Получаем текущую дату
datetime.date(2015, 4, 3)
```

- fromtimestamp(<Количество секунд>) — возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи:

```
>>> import datetime, time
>>> datetime.date.fromtimestamp(time.time())  # Текущая дата
datetime.date(2015, 4, 3)
>>> datetime.date.fromtimestamp(1321954972.0) # Дата 22-11-2011
datetime.date(2011, 11, 22)
```

- fromordinal(<Количество дней с 1-го года>) — возвращает дату, соответствующую количеству дней, прошедших с первого года. В качестве параметра указывается число от 1 до datetime.date.max.toordinal(). Примеры:

```
>>> 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 атрибутов:

- 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 (3-04-2015)
>>> d.year, d.month, d.day
(2015, 4, 3)
```

Над экземплярами класса `date` **можно производить следующие операции:**

- `date2 = date1 + timedelta` — **прибавляет к дате указанный период в днях. Значения атрибутов** `timedelta.seconds` **и** `timedelta.microseconds` **игнорируются;**
- `date2 = date1 - timedelta` — **вычитает из даты указанный период в днях. Значения атрибутов** `timedelta.seconds` **и** `timedelta.microseconds` **игнорируются;**
- `timedelta = date1 - date2` — **возвращает разницу между датами (период в днях). Атрибуты** `timedelta.seconds` **и** `timedelta.microseconds` **будут иметь значение** `0`;
- **можно также сравнивать две даты с помощью операторов сравнения.**

Примеры:

```
>>> d1 = datetime.date(2015, 4, 3)
>>> d2 = datetime.date(2015, 1, 1)
>>> t = datetime.timedelta(days=10)
>>> d1 + t, d1 - t              # Прибавляем и вычитаем 10 дней
(datetime.date(2015, 4, 13), datetime.date(2015, 3, 24))
>>> d1 - d2                     # Разница между датами
datetime.timedelta(92)
>>> 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(2015, 4, 3)
>>> d.replace(2014, 12) # Заменяем год и месяц
datetime.date(2014, 12, 3)
>>> d.replace(year=2015, month=1, day=31)
datetime.date(2015, 1, 31)
>>> d.replace(day=30)   # Заменяем только день
datetime.date(2015, 1, 30)
```

- `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 2015, 4, 3)
>>> d.strftime "%d.%m.%Y")
'03.04.2015'
```

- `isoformat()` — **возвращает дату в формате** `ГГГГ-ММ-ДД`:

```
>>> d = datetime.date(2015, 4, 3)
>>> d.isoformat()
'2015-04-03'
```

- `ctime()` — **возвращает строку формата** `"%a %b %d %H:%M:%S %Y"`:

```
>>> d = datetime.date(2015, 4, 3)
>>> d.ctime()
'Fri Apr  3 00:00:00 2015'
```

- `timetuple()` — **возвращает объект** `struct_time` **с датой и временем**:

```
>>> d = datetime.date(2015, 4, 3)
>>> d.timetuple()
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=3, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=4, tm_yday=93, tm_isdst=-1)
```

- `toordinal()` — **возвращает количество дней, прошедших с 1-го года**:

```
>>> d = datetime.date(2015, 4, 3)
>>> d.toordinal()
735691
>>> datetime.date.fromordinal(735691)
datetime.date(2015, 4, 3)
```

- `weekday()` — **возвращает порядковый номер дня в неделе (**`0` **— для понедельника,** `6` **— для воскресенья)**:

```
>>> d = datetime.date(2015, 4, 3)
>>> d.weekday() # 4 — это пятница
4
```

- `isoweekday()` — **возвращает порядковый номер дня в неделе (**`1` **— для понедельника,** `7` **— для воскресенья)**:

```
>>> d = datetime.date(2015, 4, 3)
>>> d.isoweekday() # 5 — это пятница
5
```

- `isocalendar()` — **возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе)**:

```
>>> d = datetime.date(2015, 4, 3)
>>> d.isocalendar()
(2015, 14, 5)
```

Наконец, имеется поддержка следующих атрибутов класса:

- `min` — **минимально возможное значение даты;**
- `max` — **максимально возможное значение даты;**
- `resolution` — **минимальное возможное различие между значениями даты.**

Выведем значения этих атрибутов:

```
>>> datetime.date.min
datetime.date(1, 1, 1)
>>> datetime.date.max
datetime.date(9999, 12, 31)
>>> datetime.date.resolution
datetime.timedelta(1)
```

10.4.3. Класс *time*

Класс `time` **из модуля** `datetime` **позволяет выполнять операции над временем. Конструктор класса имеет следующий формат:**

```
time([hour][, minute][, second][, microsecond][, tzinfo])
```

Все параметры не являются обязательными. 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 атрибутов:

- `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 выполнять арифметические операции. Можно только производить сравнения. Примеры:

```
>>> 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. Примеры:

```
>>> 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'
```

Тип `time` поддерживает такие атрибуты класса:

- `min` — минимально возможное значение времени;
- `max` — максимально возможное значение времени;
- `resolution` — минимальное возможное различие между значениями времени.

Вот значения этих атрибутов:

```
>>> datetime.time.min
datetime.time(0, 0)
>>> datetime.time.max
datetime.time(23, 59, 59, 999999)
>>> datetime.time.resolution
datetime.timedelta(0, 0, 1)
```

> ***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 выполнять 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>` — **в виде числа, расположенного в диапазоне между значениями, хранящимися в константах** `MINYEAR` (`1`) **и** `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`**. Примеры:**

```
>>> import datetime
>>> datetime.datetime(2015, 4, 6)
datetime.datetime(2015, 4, 6, 0, 0)
>>> datetime.datetime(2015, 4, 6, hour=12, minute=55)
datetime.datetime(2015, 4, 6, 12, 55)
>>> datetime.datetime(2015, 32, 20)
... Фрагмент опущен ...
ValueError: month must be in 1..12
>>> d = datetime.datetime(2015, 4, 6, 16, 1, 5)
>>> repr(d), str(d)
('datetime.datetime(2015, 4, 6, 16, 1, 5)', '2015-04-06 16:01:05')
```

Для создания экземпляра класса можно также воспользоваться следующими методами:

- `today()` — **возвращает текущие дату и время:**

  ```
  >>> datetime.datetime.today()
  datetime.datetime(2015, 4, 6, 16, 2, 23, 944152)
  ```

- `now([<Зона>])` — **возвращает текущие дату и время. Если параметр не задан, то метод аналогичен методу** `today()`**. Пример:**

  ```
  >>> datetime.datetime.now()
  datetime.datetime(2015, 4, 6, 16, 2, 45, 144777)
  ```

- `utcnow()` — **возвращает текущее универсальное время (UTC):**

  ```
  >>> datetime.datetime.utcnow()
  datetime.datetime(2015, 4, 6, 13, 3, 9, 862255)
  ```

- `fromtimestamp(<Количество секунд>[, <Зона>])` — **возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи**:

```
>>> import datetime, time
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2015, 4, 6, 16, 3, 34, 978523)
>>> datetime.datetime.fromtimestamp(1421579037.0)
datetime.datetime(2015, 1, 18, 14, 3, 57)
```

- `utcfromtimestamp(<Количество секунд>)` — **возвращает дату, соответствующую количеству секунд, прошедших с начала эпохи, в универсальном времени (UTC). Примеры:**

```
>>> datetime.datetime.utcfromtimestamp(time.time())
datetime.datetime(2015, 4, 6, 13, 4, 45, 756479)
>>> datetime.datetime.utcfromtimestamp(1421579037.0)
datetime.datetime(2015, 1, 18, 11, 3, 57)
```

- `fromordinal(<Количество дней с 1-го года>)` — **возвращает дату, соответствующую количеству дней, прошедших с 1-го года. В качестве параметра указывается число от** `1` **до** `datetime.datetime.max.toordinal()`. **Примеры:**

```
>>> 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(2015, 4, 6)  # Экземпляр класса date
>>> t = datetime.time(16, 7, 22)   # Экземпляр класса time
>>> datetime.datetime.combine(d, t)
datetime.datetime(2015, 4, 6, 16, 7, 22)
```

- `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`. **Примеры:**

```
>>> datetime.datetime.strptime("06.04.2015", "%d.%m.%Y")
datetime.datetime(2015, 4, 6, 0, 0)
>>> datetime.datetime.strptime("06.04.2015", "%d-%m-%Y")
... Фрагмент опущен ...
ValueError: time data '06.04.2015'
does not match format '%d-%m-%Y'
```

Получить результат можно с помощью следующих атрибутов:

- `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`**).**

Примеры:

```
>>> d = datetime.datetime(2015, 4, 6, 16, 7, 22)
>>> d.year, d.month, d.day
(2015, 4, 6)
>>> d.hour, d.minute, d.second, d.microsecond
(16, 7, 22, 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(2015, 4, 6, 16, 7, 22)
>>> d2 = datetime.datetime(2015, 4, 1, 12, 31, 4)
>>> t = datetime.timedelta(days=10, minutes=10)
>>> d1 + t                      # Прибавляем 10 дней и 10 минут
datetime.datetime(2015, 4, 16, 16, 17, 22)
>>> d1 – t                      # Вычитаем 10 дней и 10 минут
datetime.datetime(2015, 3, 27, 15, 57, 22)
>>> d1 – d2                     # Разница между датами
datetime.timedelta(5, 12978)
>>> 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`**, хранящий дату:**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 10, 54)
>>> d.date()
datetime.date(2015, 4, 6)
```

- `time()` — **возвращает экземпляр класса** `time`**, хранящий время:**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 10, 54)
>>> d.time()
datetime.time(16, 10, 54)
```

- `timetz()` — **возвращает экземпляр класса** `time`**, хранящий время. Метод учитывает параметр** `tzinfo`**;**
- `timestamp()` — **возвращает вещественное число, представляющее количество секунд, прошедшее с начала эпохи (обычно с 1 января 1970 г.). Поддержка этого метода появилась в Python 3.3.**

Пример:

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.timestamp()
1428326052.0
```

- `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. Примеры:**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.replace(2014, 12)
datetime.datetime(2014, 12, 6, 16, 14, 12)
>>> d.replace(hour=12, month=10)
datetime.datetime(2015, 10, 6, 12, 14, 12)
```

- `timetuple()` — **возвращает объект** `struct_time` **с датой и временем:**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.timetuple()
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=6, tm_hour=16, tm_min=14,
tm_sec=12, tm_wday=0, tm_yday=96, tm_isdst=-1)
```

- `utctimetuple()` — **возвращает объект** `struct_time` **с датой в универсальном времени (UTC):**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.utctimetuple()
time.struct_time(tm_year=2015, tm_mon=4, tm_mday=6, tm_hour=16, tm_min=14,
tm_sec=12, tm_wday=0, tm_yday=96, tm_isdst=0)
```

- `toordinal()` — **возвращает количество дней, прошедшее с 1-го года:**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.toordinal()
735694
```

- `weekday()` — **возвращает порядковый номер дня в неделе (**`0` **— для понедельника,** `6` **— для воскресенья):**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.weekday() # 0 – это понедельник
0
```

- `isoweekday()` — **возвращает порядковый номер дня в неделе (**`1` **— для понедельника,** `7` **— для воскресенья):**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.isoweekday() # 1 – это понедельник
1
```

- `isocalendar()` — **возвращает кортеж из трех элементов (год, номер недели в году и порядковый номер дня в неделе):**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.isocalendar()
(2015, 15, 1)
```

- `isoformat([<Разделитель даты и времени>])` — **возвращает дату в формате ISO 8601. Если разделитель не указан, используется буква** `T`. **Примеры:**

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.isoformat()            # Разделитель не указан
'2015-04-06T16:14:12'
>>> d.isoformat(" ")         # Пробел в качестве разделителя
'2015-04-06 16:14:12'
```

- `ctime()` — **возвращает строку формата** `"%a %b %d %H:%M:%S %Y"`:

```
>>> d = datetime.datetime(2015, 4, 6, 16, 14, 12)
>>> d.ctime()
'Mon Apr  6 16:14:12 2015'
```

- `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(2015, 4, 6, 16, 14, 12)
>>> d.strftime("%d.%m.%Y %H:%M:%S")
'06.04.2015 16:14:12'
```

Поддерживаются также следующие атрибуты класса:

- `min` — **минимально возможные значения даты и времени;**
- `max` — **максимально возможные значения даты и времени;**
- `resolution` — **минимальное возможное различие между значениями даты и времени.**

Значения этих атрибутов:

```
>>> datetime.datetime.min
datetime.datetime(1, 1, 1, 0, 0)
>>> datetime.datetime.max
datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)
>>> datetime.datetime.resolution
datetime.timedelta(0, 0, 1)
```

> ***Примечание***
>
> **Экземпляры класса** `datetime` **также поддерживают 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` **формирует календарь в виде простого текста или HTML-кода. Прежде чем использовать модуль, необходимо подключить его с помощью инструкции:**

```
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 апреле 2015 года, распределенных по дням недели:

```
>>> import calendar
>>> c = calendar.Calendar(0)
>>> c.monthdayscalendar(2015, 4) # 4 — это апрель
[[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, 0, 0, 0]]
```

- `TextCalendar` — **позволяет вывести календарь в виде простого текста. Формат конструктора:**

```
TextCalendar([<Первый день недели>])
```

Выведем календарь на весь 2015 год:

```
>>> c = calendar.TextCalendar(0)
>>> print(c.formatyear(2015)) # Текстовый календарь на 2015 год
```

В качестве результата мы получим большую строку, содержащую календарь в виде отформатированного текста;

- `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 2015 год на русском языке:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2015))
```

- `HTMLCalendar` — **позволяет вывести календарь в формате HTML. Формат конструктора:**

```
HTMLCalendar([<Первый день недели>])
```

Выведем календарь на весь 2015 год:

```
>>> c = calendar.HTMLCalendar(0)
>>> print(c.formatyear(2011))
```

Результатом будет большая строка с HTML-кодом календаря, отформатированного в виде таблицы;

- `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 2015 год на русском языке в виде отдельной Web-страницы:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> xhtml = c.formatyearpage(2011, encoding="windows-1251")
>>> print(xhtml.decode("cp1251"))
```

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, поддерживаемые классом calendar. Изменить значение параметра позволяет метод setfirstweekday(<Первый день недели>). В качестве примера выведем текстовый календарь на январь 2015 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(2015, 1)) # Текстовый календарь на январь 2015 г.
```

10.5.1. Методы классов *TextCalendar* и *LocaleTextCalendar*

Экземпляры классов TextCalendar и LocaleTextCalendar поддерживают следующие методы:

- formatmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]]) — возвращает текстовый календарь на указанный месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов перевода строки между строками. Выведем календарь на апрель 2015 года:

```
>>> import calendar
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatmonth(2015, 4))
    Апрель 2015
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
```

- prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]]) — метод аналогичен методу formatmonth(), но не возвращает календарь в виде строки, а сразу выводит его на экран. Распечатаем календарь на апрель 2015 года, указав ширину поля с днем равной 4 символам:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.prmonth(2015, 4, 4)
           Апрель 2015
 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
```

- formatyear(<Год>[, w=2][, l=1][, c=6][, m=3]) — возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:
  - w — ширина поля с днем (по умолчанию 2);
  - l — количество символов перевода 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 2015 год. На одной строке выведем сразу четыре месяца и установим количество пробелов между месяцами:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2015, m=4, c=2))
```

- pryear(<Год>[, w=2][, l=1][, c=6][, m=3]) — метод аналогичен методу formatyear(), но не возвращает календарь в виде строки, а сразу выводит его. В качестве примера распечатаем календарь на 2015 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 перевода строки:

```
>>> c = calendar.LocaleTextCalendar(0, "Russian_Russia.1251")
>>> c.pryear(2015, 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 календарь на указанный месяц в году в виде HTML-кода. 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 атрибут cssclasses, который 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 апрель 2015 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(2015, 4, False))
```

- formatyear(<Год>[, <Количество месяцев на строке>]) — возвращает календарь на указанный год в виде HTML-кода. Календарь будет отформатирован с помощью нескольких HTML-таблиц. Для примера выведем календарь на 2015 год так, чтобы на одной строке выводились сразу четыре месяца:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> print(c.formatyear(2015, 4))
```

- `formatyearpage(<Год>[, width][, css][, encoding])` — **возвращает календарь на указанный год в виде отдельной Web-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. Для примера выведем календарь на 2015 год так, чтобы на одной строке выводилось сразу четыре месяца. Укажем при этом кодировку файла:

```
>>> c = calendar.LocaleHTMLCalendar(0, "Russian_Russia.1251")
>>> xhtml = c.formatyearpage(2015, 4, encoding="windows-1251")
>>> type(xhtml) # Возвращаемая строка имеет тип данных bytes
<class 'bytes'>
>>> print(xhtml.decode("cp1251"))
```

10.5.3. Другие полезные функции

Модуль `calendar` **предоставляет еще и несколько функций, которые позволяют вывести текстовый календарь без создания объекта соответствующего 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 перевода строки>]])` — **возвращает текстовый календарь на указанный месяц в году. Третий параметр позволяет указать ширину поля с днем, а четвертый параметр — количество символов перевода строки между строками. Выведем календарь на апрель 2015 года:**

```
>>> calendar.setfirstweekday(0)
>>> print(calendar.month(2015, 4)) # Апрель 2015 года
     April 2015
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
```

- prmonth(<Год>, <Месяц>[, <Ширина поля с днем>[, <Количество символов перевода строки>]]) — **функция аналогична функции** month(), **но не возвращает календарь в виде строки, а сразу выводит его. Выведем календарь на апрель 2015 года:**

```
>>> calendar.prmonth(2015, 4) # Апрель 2015 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 апреля 2015 года:**

```
>>> calendar.monthcalendar(2015, 4)
[[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, 0, 0, 0]]
```

- monthrange(<Год>, <Месяц>) — **возвращает кортеж из двух элементов: номера дня недели, приходящегося на первое число указанного месяца, и количества дней в месяце:**

```
>>> print(calendar.monthrange(2015, 4))
(2, 30)
>>> # Апрель 2015 года начинается со среды (2) и включает 30 дней
```

- calendar(<Год>[, w][, l][, c][, m]) — **возвращает текстовый календарь на указанный год. Параметры имеют следующее предназначение:**
  - w — **ширина поля с днем (по умолчанию** 2**);**
  - l — **количество символов перевода 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. Для примера выведем календарь на 2015 год так, чтобы на одной строке выводилось сразу четыре месяца. Установим при этом количество пробелов между месяцами:

```
>>> print(calendar.calendar(2015, 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. Для примера выведем календарь на 2015 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 перевода строки:**

```
>>> calendar.prcal(2015, 2, 1, 4, 2)
```

- weekheader(<n>) — **возвращает строку, которая содержит аббревиатуры дней недели с учетом текущей локали, разделенные пробелами. Единственный параметр задает длину каждой аббревиатуры в символах. Примеры:**

```
>>> calendar.weekheader(4)
'Mon  Tue  Wed  Thu  Fri  Sat  Sun '
>>> calendar.weekheader(2)
'Mo Tu We Th Fr Sa Su'
>>> import locale                    # Задаем другую локаль
>>> locale.setlocale(locale.LC_ALL, "Russian_Russia.1251")
'Russian_Russia.1251'
```

```
>>> calendar.weekheader(2)
'Пн Вт Ср Чт Пт Сб Вс'
```

- `isleap(<Год>)` — **возвращает значение** `True`**, если указанный год является високосным, в противном случае** — `False`:

```
>>> calendar.isleap(2015), 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, 2015) # 2012 — високосный год
1
>>> calendar.leapdays(2010, 2017) # 2012 и 2016 — високосные года
2
```

- `weekday(<Год>, <Месяц>, <День>)` — **возвращает номер дня недели (**`0` — **для понедельника,** `6` — **для воскресенья):**

```
>>> calendar.weekday(2015, 4, 7)
1
```

- `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(1321954972.0) # Дата 22-11-2011
>>> d
time.struct_time(tm_year=2011, tm_mon=11, tm_mday=22, tm_hour=9,
 tm_min=42, tm_sec=52, tm_wday=1, tm_yday=326, tm_isdst=0)
>>> tuple(d)
(2011, 11, 22, 9, 42, 52, 1, 326, 0)
>>> calendar.timegm(d)
1321954972
>>> calendar.timegm((2011, 11, 22, 9, 42, 52, 1, 326, 0))
1321954972
```

Модуль `calendar` **предоставляет также несколько атрибутов:**

- `day_name` — **список полных названий дней недели в текущей локали:**

```
>>> [i for i in calendar.day_name]
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
```

- `day_abbr` — **список аббревиатур названий дней недели в текущей локали:**

```
>>> [i for i in calendar.day_abbr]
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

- month_name — **список полных названий 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 — **список аббревиатур 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'
>>> [i for i in calendar.day_abbr]
['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Вс']
>>> [i for i in calendar.month_name]
['', 'Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', 'Июль',
 'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь']
>>> [i for i in calendar.month_abbr]
['', '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 инструкции:**

```
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), время выполнения которого предполагается измерить. 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** 10000 **тремя способами и выведем время выполнения каждого способа (листинг 10.4).**

Листинг 10.4. Измерение времени выполнения

```
# -*- coding: utf-8 -*-
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 range(1, 10001):
    j += i
"""
t2 = Timer(stmt=code2)
print("for:", t2.timeit(number=10000))
code3 = """\
j = sum(range(1, 10001))
"""
t3 = Timer(stmt=code3)
print("sum:", t3.timeit(number=10000))
input()
```

Примерный результат выполнения (зависит от мощности компьютера):

```
while: 19.892227524223074
for: 11.3560930317114
sum: 3.7005646209492618
```

Как видно из результата, цикл `for` работает почти 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: utf-8 -*-
from timeit import Timer
code1 = """\
arr1 = []
for i in range(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 range(1, 10001)]
"""
```

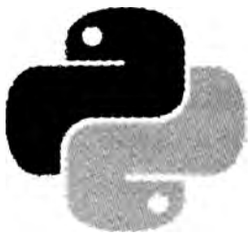
```
t2 = Timer(stmt=code2)
print("генератор:", t2.repeat(repeat=3, number=2000))
input()
```

Примерный результат выполнения:

```
append: [9.476708945758174, 10.29404489890369, 9.806380879254462]
генератор: [7.581222872765533, 7.621238914260893, 7.806058897124785]
```

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. Определение функции и ее вызов

Функция создается (или, как говорят программисты, определяется) с помощью ключевого слова `def` по следующей схеме:

```
def <Имя функции>([<Параметры>]):
    ["""  Строка документирования """]
    <Тело функции>
    [return <Результат>]
```

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 также имеет значение.

После имени функции в круглых скобках можно указать один или несколько параметров через запятую, а если функция не принимает параметры, указываются только круглые скобки. После круглых скобок ставится двоеточие.

Тело функции представляет собой составную конструкцию. Как и в любой составной конструкции, инструкции внутри функции выделяются одинаковым количеством пробелов слева. Концом функции считается инструкция, перед которой находится меньшее количество пробелов. Если тело функции не содержит инструкций, то внутри ее необходимо разместить оператор `pass`, который не выполняет никаких действий. Этот оператор удобно использовать на этапе отладки программы, когда мы определили функцию, а тело решили дописать позже. Пример функции, которая ничего не делает:

```
def func():
    pass
```

Необязательная инструкция `return` позволяет вернуть из функции какое-либо значение в качестве результата. После исполнения этой инструкции выполнение функции будет остановлено. Это означает, что инструкции, следующие после оператора `return`, никогда не будут выполнены. Пример:

```
def func():
    print("Текст до инструкции return")
    return "Возвращаемое значение"
    print("Эта инструкция никогда не будет выполнена")

print(func()) # Вызываем функцию
```

Результат выполнения:

```
Текст до инструкции return
Возвращаемое значение
```

Инструкции `return` может не быть вообще. В этом случае выполняются все инструкции внутри функции, и в качестве результата возвращается значение `None`.

Для примера создадим три функции (листинг 11.1).

Листинг 11.1. Определения функций

```
def print_ok():
    """ Пример функции без параметров """
    print("Сообщение при удачно выполненной операции")

def echo(m):
    """ Пример функции с параметром """
    print(m)

def summa(x, y):
    """ Пример функции с параметрами,
        возвращающей сумму двух переменных """
    return x + y
```

При вызове функции значения передаются внутри круглых скобок через запятую. Если функция не принимает параметров, то указываются только круглые скобки. Необходимо также заметить, что количество параметров в определении функции должно совпадать с количеством параметров при вызове, иначе будет выведено сообщение об ошибке. Вызвать функции из листинга 11.1 можно способами, указанными в листинге 11.2.

Листинг 11.2. Вызов функций

```
print_ok()              # Вызываем функцию без параметров
echo("Сообщение")       # Функция выведет сообщение
x = summa(5, 2)         # Переменной x будет присвоено значение 7
a, b = 10, 50
y = summa(a, b)         # Переменной y будет присвоено значение 60
```

Как видно из последнего примера, имя переменной в вызове функции может не совпадать с именем переменной в определении функции. Кроме того, *глобальные* переменные `x` и `y` не

конфликтуют с одноименными переменными в определении функции, т. к. они расположены в разных областях видимости. Переменные, указанные в определении функции, являются локальными и доступны только внутри функции. Более подробно области видимости мы рассмотрим в *разд. 11.9*.

Оператор +, используемый в функции `summa()`, применяется не только для сложения чисел, но и позволяет объединить последовательности. То есть, функция `summa()` может использоваться не только для сложения чисел. В качестве примера выполним конкатенацию строк и объединение списков:

```
def summa(x, y):
    return x + y

print(summa("str", "ing"))    # Выведет: string
print(summa([1, 2], [3, 4])) # Выведет: [1, 2, 3, 4]
```

Как вы уже знаете, все в языке Python представляет собой объекты: строки, списки и даже сами типы данных. Не являются исключением и функции. 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 summa(x, y):
    return x + y

f = summa                          # 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 summa(x, y):
    return x + y

def func(f, a, b):
    """ Через переменную f будет доступна ссылка на
        функцию summa() """
    return f(a, b) # Вызываем функцию summa()

# Передаем ссылку на функцию в качестве параметра
v = func(summa, 10, 20)
```

Объекты функций поддерживают множество атрибутов, обратиться к которым можно, указав атрибут после названия функции через точку. Например, через атрибут `__name__` можно получить название функции в виде строки, через атрибут `__doc__` — строку документиро-

вания и т. д. Для примера выведем названия всех атрибутов функции с помощью встроенной функции `dir()`:

```
>>> def summa(x, y):
        """ Суммирование двух чисел """
        return x + y

>>> dir(summa)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__',
'__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__',
'__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> summa.__name__
'summa'
>>> summa.__code__.co_varnames
('x', 'y')
>>> summa.__doc__
' Суммирование двух чисел '
```

11.2. Расположение определений функций

Все инструкции в программе выполняются последовательно сверху вниз. Это означает, что, прежде чем использовать в программе идентификатор, его необходимо предварительно определить, присвоив ему значение. Поэтому определение функции должно быть расположено перед вызовом функции.

Правильно:

```
def summa(x, y):
    return x + y
v = summa(10, 20) # Вызываем после определения. Все нормально
```

Неправильно:

```
v = summa(10, 20) # Идентификатор еще не определен. Это ошибка!!!
def summa(x, y):
    return x + y
```

В последнем случае будет выведено сообщение об ошибке: `NameError: name 'summa' is not defined`. Чтобы избежать ошибки, определение функции размещают в самом начале программы после подключения модулей или в отдельном модуле (о них речь пойдет в *главе 12*).

С помощью оператора ветвления `if` можно изменить порядок выполнения программы — например, 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: utf-8 -*-
n = input("Введите 1 для вызова первой функции: ")
```

```
if n == "1":
    def echo():
        print("Вы ввели число 1")
else:
    def echo():
        print("Альтернативная функция")

echo() # Вызываем функцию
input()
```

При вводе числа 1 мы получим сообщение `"Вы ввели число 1"`, в противном случае — `"Альтернативная функция"`.

Помните, что инструкция `def` всего лишь присваивает ссылку на объект функции идентификатору, расположенному после ключевого слова `def`. Если определение одной функции встречается в программе несколько раз, то будет использоваться функция, которая была определена последней. Пример:

```
def echo():
    print("Вы ввели число 1")
def echo():
    print("Альтернативная функция")
echo() # 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 summa(x, y=2):          # y — необязательный параметр
    return x + y
a = summa(5)               # Переменной a будет присвоено значение 7
b = summa(10, 50)          # Переменной b 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 summa(x, y):
    return x + y
print(summa(10, 20))        # Выведет: 30
```

Переменной `x` при сопоставлении будет присвоено значение `10`, а переменной `y` — значение `20`. Но язык Python позволяет также передать значения в функцию, используя сопостав-

ление по ключам. Для этого при вызове функции параметрам присваиваются значения, причем последовательность указания параметров в этом случае может быть произвольной (листинг 11.7).

Листинг 11.7. Сопоставление по ключам

```
def summa(x, y):
    return x + y
print(summa(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 summa(a=2, b=3, c=4): # Все параметры являются необязательными
    return a + b + c
print(summa(2, 3, 20))    # Позиционное присваивание
print(summa(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 summa(a, b, c):
    return a + b + c
t1, arr = (1, 2, 3), [1, 2, 3]
print(summa(*t1))                  # Распаковываем кортеж
print(summa(*arr))                 # Распаковываем список
t2 = (2, 3)
print(summa(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 summa(a, b, c):
    return a + b + c
d1 = {"a": 1, "b": 2, "c": 3}
print(summa(**d1))                 # Распаковываем словарь
t, d2 = (1, 2), {"c": 3}
print(summa(*t, **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 func(a, b):
    a, b = 20, "str"
x, s = 80, "test"
```

```
func(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 func(a, b):
    a[0], b["a"] = "str", 800
x = [1, 2, 3]               # Список
y = {"a": 1, "b": 2}        # Словарь
func(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, поскольку список и словарь относятся к изменяемым типам. Чтобы избежать изменения значений, внутри функции следует создать копию объекта (листинг 11.10).

Листинг 11.10. Передача изменяемого объекта в функцию

```
def func(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
func(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:

```
func(x[:], y.copy())
```

Если указать объект, имеющий изменяемый тип, в качестве значения по умолчанию, то этот объект будет сохраняться между вызовами функции. Пример:

```
def func(a=[]):
    a.append(2)
    return a
print(func())                   # Выведет: [2]
print(func())                   # Выведет: [2, 2]
print(func())                   # Выведет: [2, 2, 2]
```

Как видно из примера, значения накапливаются внутри списка. Обойти эту проблему можно, например, следующим образом:

```
def func(a=None):
    # Создаем новый список, если значение равно None
    if a is None:
        a = []
    a.append(2)
    return a
print(func())                   # Выведет: [2]
print(func([1]                  # Выведет: [1, 2]
print(func())                   # Выведет: [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. Для примера напишем функцию суммирования произвольного количества чисел (листинг 11.11).

Листинг 11.11. Сохранение переданных данных в кортеже

```
def summa(*t):
    """ Функция принимает произвольное количество параметров """
    res = 0
    for i in t:       # Перебираем кортеж с переданными параметрами
        res += i
    return res
print(summa(10, 20))                    # Выведет: 30
print(summa(10, 20, 30, 40, 50, 60))    # Выведет: 210
```

Можно также вначале указать несколько обязательных параметров и параметров, имеющих значения по умолчанию:

```
def summa(x, y=5, *t): # Комбинация параметров
    res = x + y
    for i in t:        # Перебираем кортеж с переданными параметрами
        res += i
    return res
print(summa(10))                        # Выведет: 15
print(summa(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 func(**d):
    for i in d:     # Перебираем словарь с переданными параметрами
        print("{0} => {1}".format(i, d[i]), end=" ")
func(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 func(*t, **d):
    """ Функция примет любые параметры """
    for i in t:
        print(i, end=" ")
```

```
    for i in d:    # Перебираем словарь с переданными параметрами
        print("{0} => {1}".format(i, d[i]), end=" ")
func(35, 10, a=1, b=2, c=3) # Выведет: 35 10 a => 1 c => 3 b => 2
func(10)                    # Выведет: 10
func(a=1, b=2)              # Выведет: a => 1 b => 2
```

В определении функции можно указать, что некоторые параметры передаются только по именам. Для этого параметры должны указываться после параметра с одной звездочкой, но перед параметром с двумя звездочками. Именованные параметры могут иметь значения по умолчанию. Пример:

```
def func(*t, a, b=10, **d):
    print(t, a, b, d)
func(35, 10, a=1, c=3) # Выведет: (35, 10) 1 10 {'c': 3}
func(10, a=5)          # Выведет: (10,) 5 10 {}
func(a=1, b=2)         # Выведет: () 1 2 {}
func(1, 2, 3)          # Ошибка. Параметр a обязателен!
```

В этом примере переменная `t` примет любое количество значений, которые будут объединены в кортеж. Переменные `a` и `b` должны передаваться только по именам, причем переменной `a` обязательно нужно передать значение при вызове функции. Переменная `b` имеет значение по умолчанию, поэтому при вызове допускается не передавать ей значение, но если значение передается, то оно должно быть указано после названия параметра и оператора `=`. Переменная `d` примет любое количество именованных параметров и сохранит их в словаре. Обратите внимание на то, что, хотя переменные `a` и `b` являются именованными, они не попадут в этот словарь.

Параметра с двумя звездочками может не быть в определении функции, а вот параметр с одной звездочкой при указании параметров, передаваемых только по именам, должен присутствовать обязательно. Если функция не должна принимать переменного количества параметров, но необходимо использовать переменные, передаваемые только по именам, то можно указать только звездочку без переменной:

```
def func(x=1, y=2, *, a, b=10):
    print(x, y, a, b)
func(35, 10, a=1)      # Выведет: 35 10 1 10
func(10, a=5)          # Выведет: 10 2 5 10
func(a=1, b=2)         # Выведет: 1 2 1 2
func(a=1, y=8, x=7)    # Выведет: 7 8 1 10
func(1, 2, 3)          # Ошибка. Параметр a обязателен!
```

В этом примере значения переменным `x` и `y` можно передавать как по позициям, так и по именам. Поскольку переменные имеют значения по умолчанию, допускается вообще не передавать им значений. Переменные `a` и `b` расположены после параметра с одной звездочкой, поэтому передать значения при вызове можно только по именам. Так как переменная `b` имеет значение по умолчанию, допускается не передавать ей значение при вызове, а вот переменная `a` обязательно должна получить значение, причем только по имени.

11.5. Анонимные функции

Помимо обычных, язык Python позволяет использовать анонимные функции, которые также 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 таких функций нет имени; по этой причине их и называют анонимными.

В качестве значения анонимная функция возвращает ссылку на объект-функцию, которую можно сохранить в переменной или передать в качестве параметра в другую функцию. Вызвать анонимную функцию можно, как и обычную, с помощью круглых скобок, внутри которых расположены передаваемые параметры. Пример использования анонимных функций приведен в листинге 11.14.

Листинг 11.14. Пример использования анонимных 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 анонимных функций могут быть необязательными. Для этого параметрам в определении функции присваивается значение по умолчанию (листинг 11.15).

Листинг 11.15. Необязательные параметры в анонимных функциях

```
f = lambda x, y=2: x + y
print(f(5))                             # Выведет: 7
print(f(5, 6))                          # Выведет: 11
```

Чаще всего анонимную функцию не сохраняют в переменной, а сразу передают в качестве параметра в другую функцию. Например, метод списков `sort()` позволяет указать пользовательскую функцию в параметре `key`. Отсортируем список без учета регистра символов, указав в качестве параметра функцию (листинг 11.16).

Листинг 11.16. Сортировка без учета регистра символов

```
arr = ["единица1", "Единый", "Единица2"]
arr.sort(key=lambda s: s.lower())
for i in arr:
    print(i, end=" ")
# Результат выполнения: 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 func(x, y):
    for i in range(1, x+1):
        yield i ** y

for n in func(10, 2):
    print(n, end=" ")     # Выведет: 1 4 9 16 25 36 49 64 81 100
print()                   # Вставляем пустую строку
for n in func(10, 3):
    print(n, end=" ")     # 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 func(x, y):
    for i in range(1, x+1):
       yield i ** y

i = func(3, 3)
print(i.__next__())       # Выведет: 1 (1 ** 3)
print(i.__next__())       # Выведет: 8 (2 ** 3)
print(i.__next__())       # Выведет: 27 (3 ** 3)
print(i.__next__())       # Исключение StopIteration
```

Получается, что 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, т. к. не понадобится загружать весь список со значениями в память.

В Python 3.3 появилась возможность вызвать одну функцию-генератор из другой. Для этого применяется расширенный синтаксис ключевого слова `yield`:

```
yield from <Вызываемая функция-генератор>
```

Рассмотрим следующий пример. Пусть у нас есть список чисел, и нам требуется получить другой список, включающий числа в диапазоне от 1 до каждого из чисел в первом списке. Чтобы создать такой список, мы напишем код, показанный в листинге 11.19.

Листинг 11.19. Вызов одной функции-генератора из другой (простой случай)

```
def gen(l):
    for e in l:
        yield from range(1, e + 1)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Здесь мы в функции-генераторе `gen` перебираем переданный ей в качестве параметра список и для каждого его элемента вызываем другую функцию-генератор. В качестве последней выступает выражение, создающее диапазон от 1 до значения этого элемента, увеличенного на единицу (чтобы это значение вошло в диапазон). В результате на выходе мы получим вполне ожидаемый результат:

```
1 2 3 4 5 1 2 3 4 5 6 7 8 9 10
```

Усложним задачу, включив в результирующий список числа, умноженные на 2. Код, выполняющий эту задачу, показан в листинге 11.20.

Листинг 11.20. Вызов одной функции-генератора из другой (сложный случай)

```
def gen2(n):
    for e in range(1, n + 1):
        yield e * 2

def gen(l):
    for e in l:
        yield from gen2(e)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Здесь мы вызываем из функции-генератора `gen` написанную нами самими функцию-генератор `gen2`. Она создает диапазон, перебирает все входящие в него числа и возвращает их умноженными на 2. Результат работы приведенного в листинге кода таков:

```
2 4 6 8 10 2 4 6 8 10 12 14 16 18 20
```

Что и требовалось нам получить.

11.7. Декораторы функций

Декораторы позволяют изменить поведение обычных функций — например, выполнить какие-либо действия перед выполнением функции. Рассмотрим это на примере (листинг 11.21).

Листинг 11.21. Декораторы функций

```
def deco(f):                              # Функция-декоратор
    print("Вызвана функция func()")
    return f                              # Возвращаем ссылку на функцию
```

```
@deco
def func(x):
    return "x = {0}".format(x)

print(func(10))
```

Выведет:

```
Вызвана функция func()
x = 10
```

В этом примере перед определением функции `func()` указывается название функции `deco()` с предваряющим символом `@`:

```
@deco
```

Таким образом, функция `deco()` становится декоратором функции `func()`. 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 deco(f):
    print("Вызвана функция func()")
    return f
def func(x):
    return "x = {0}".format(x)
# Вызываем функцию func() через функцию deco()
print(deco(func)(10))
```

Перед определением функции можно указать сразу несколько функций-декораторов. Для примера обернем функцию `func()` в два декоратора: `deco1()` и `deco2()` (листинг 11.22).

Листинг 11.22. Указание нескольких декораторов

```
def deco1(f):
    print("Вызвана функция deco1()")
    return f
def deco2(f):
    print("Вызвана функция deco2()")
    return f
@deco1
@deco2
def func(x):
    return "x = {0}".format(x)
print(func(10))
```

Выведет:

```
Вызвана функция deco2()
Вызвана функция deco1()
x = 10
```

Использование двух декораторов эквивалентно следующему коду:

```
func = deco1(deco2(func))
```

Здесь сначала будет вызвана функция `deco2()`, а затем функция `deco1`. Результат выполнения будет присвоен идентификатору `func`.

В качестве еще одного примера использования декораторов рассмотрим выполнение функции только при правильно введенном пароле (листинг 11.23).

Листинг 11.23. Ограничение доступа с помощью декоратора

```
passw = input("Введите пароль: ")

def test_passw(p):
    def deco(f):
        if p == "10":          # Сравниваем пароли
            return f
        else:
            return lambda: "Доступ закрыт"
    return deco                # Возвращаем функцию-декоратор

@test_passw(passw)
def func():
    return "Доступ открыт"
print(func())                  # Вызываем функцию
```

В этом примере после символа `@` указана не ссылка на функцию, а выражение, которое возвращает декоратор. Иными словами, декоратором является не функция `test_passw()`, а результат ее выполнения (функция `deco()`). Если введенный пароль является правильным, то выполнится функция `func()`, 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 для выполнения неопределенного количества операций. В качестве примера рассмотрим вычисление факториала (листинг 11.24).

Листинг 11.24. Вычисление факториала

```
def factorial(n):
    if n == 0 or n == 1: return 1
    else:
        return n * factorial(n - 1)

while True:
    x = input("Введите число: ")
```

```
    else:
        print("Вы ввели не число!")
print("Факториал числа {0} = {1}".format(x, factorial(x)))
```

Впрочем, проще всего для вычисления факториала 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 11.25).**

Листинг 11.25. Глобальные переменные

```
def func(glob2):
    print("Значение глобальной переменной glob1 =", glob1)
    glob2 += 10
    print("Значение локальной переменной glob2 =", glob2)

glob1, glob2 = 10, 5
func(77) # Вызываем функцию
print("Значение глобальной переменной glob2 =", glob2)
```

Результат выполнения:

```
Значение глобальной переменной glob1 = 10
Значение локальной переменной glob2 = 87
Значение глобальной переменной glob2 = 5
```

Переменной `glob2` **внутри функции присваивается значение, переданное при вызове функции. В результате создается новая переменная с тем же именем:** `glob2`, **которая является локальной. Все изменения этой переменной внутри функции не затронут значение одноименной глобальной переменной.**

Итак, ***локальные переменные*** — **это переменные, объявляемые внутри функций. Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной переменной не изменяется. Локальные переменные видны только внутри тела функции (листинг 11.26).**

Листинг 11.26. Локальные переменные

```
def func():
    local1 = 77                    # Локальная переменная
    glob1 = 25                     # Локальная переменная
    print("Значение glob1 внутри функции =", glob1)
glob1 = 10                         # Глобальная переменная
func()                             # Вызываем функцию
```

```
print("Значение glob1 вне функции =", 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 `func()`, 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 11.27).

Листинг 11.27. Ошибка при обращении к переменной до присваивания значения

```
def func():
    # Локальная переменная еще не определена
    print(glob1)                     # Эта строка вызовет ошибку!!!
    glob1 = 25                       # Локальная переменная
glob1 = 10                           # Глобальная переменная
func()                               # 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 11.28).

Листинг 11.28. Использование ключевого слова global

```
def func():
    # Объявляем переменную glob1 глобальной
    global glob1
    glob1 = 25          # Изменяем значение глобальной переменной
    print("Значение glob1 внутри функции =", glob1)
glob1 = 10              # Глобальная переменная
print("Значение glob1 вне функции =", glob1)
func()                  # Вызываем функцию
print("Значение glob1 после функции =", 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, классы 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!!!
func = lambda: x
x = 80                          # Изменили значение
print(func())                   # Выведет: 80, а не 5
```

Если необходимо сохранить именно текущее значение переменной, то можно воспользоваться способом, приведенным в листинге 11.29.

Листинг 11.29. Сохранение значения переменной

```
x = 5
# Сохраняется значение переменной x
func = (lambda y: lambda: y)(x)
x = 80                                  # Изменили значение
print(func())                           # 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 также, указав глобальную переменную в качестве значения параметра по умолчанию в определении функции (листинг 11.30).

Листинг 11.30. Сохранение значения с помощью параметра по умолчанию

```
x = 5
# Сохраняется значение переменной x
func = lambda x=x: x
x = 80                                  # Изменили значение
print(func())                           # 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 func():
    local1 = 54
    glob1 = 25
    print "Глобальные идентификаторы внутри функции")
    print(sorted globals().keys())
```

```
    print("Локальные идентификаторы внутри функции")
    print(sorted(locals().keys()))
glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(globals().keys()))
```

Результат выполнения:

```
Глобальные идентификаторы внутри функции
['__builtins__', '__cached__', '__doc__', '__file__', '__name__',
 '__package__', 'func', 'glob1', 'glob2']
Локальные идентификаторы внутри функции
['glob2', 'local1']
Глобальные идентификаторы вне функции
['__builtins__', '__cached__', '__doc__', '__file__', '__name__',
 '__package__', 'func', 'glob1', 'glob2']
```

- `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 func():
    local1 = 54
    glob2 = 25
    print("Локальные идентификаторы внутри функции")
    print(sorted(vars().keys()))
glob1, glob2 = 10, 88
func()
print("Глобальные идентификаторы вне функции")
print(sorted(vars().keys()))
print("Указание объекта в качестве параметра")
print(sorted(vars(dict).keys()))
print("Альтернативный вызов")
print(sorted(dict.__dict__.keys()))
```

11.10. Вложенные функции

Одну функцию можно вложить в другую функцию, причем уровень вложенности не ограничен. При этом вложенная функция получает свою собственную локальную область видимости и имеет доступ к переменным, объявленным внутри функции, в которую она вложена (функции-родителя). Рассмотрим вложение функций на примере (листинг 11.31).

Листинг 11.31. Вложенные функции

```
def func1(x):
    def func2():
        print(x)
    return func2
```

```
f1 = func1(10)
f2 = func1(99)
f1()                    # Выведет: 10
f2()                    # Выведет: 99
```

Здесь мы определили функцию `func1()`, принимающую один параметр. Внутри функции `func1()` определена вложенная функция `func2()`. Результатом выполнения функции `func1()` будет ссылка на эту вложенную функцию. Внутри функции `func2()` мы производим вывод значения переменной `x`, которая является локальной в функции `func1()`. Таким образом, помимо локальной, глобальной и встроенной областей видимости, добавляется *вложенная область видимости*. При этом поиск идентификаторов вначале производится внутри вложенной функции, затем внутри функции-родителя, далее в функциях более высокого уровня и лишь потом в глобальной и встроенных областях видимости. В нашем примере переменная `x` будет найдена в области видимости функции `func1()`.

Следует учитывать, что сохраняются лишь ссылки на переменные, а не их значения в момент определения функции. Например, если после определения функции `func2()` произвести изменение переменной `x`, то будет использоваться это новое значение:

```
def func1(x):
    def func2():
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                    # Выведет: 30
f2()                    # Выведет: 30
```

Обратите внимание на результат выполнения. В обоих случаях мы получили значение `30`. Если необходимо сохранить именно значение переменной при определении вложенной функции, следует передать значение как значение по умолчанию:

```
def func1(x):
    def func2(x=x): # Сохраняем текущее значение, а не ссылку
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1()                    # Выведет: 10
f2()                    # Выведет: 99
```

Теперь попробуем из вложенной функции `func2()` изменить значение переменной `x`, объявленной внутри функции `func1()`. Если внутри функции `func2()` присвоить значение переменной `x`, то будет создана новая локальная переменная с таким же именем. Если внутри функции `func2()` объявить переменную как глобальную и присвоить ей значение, то изменится значение глобальной переменной, а не значение переменной `x` внутри функции `func1()`. Таким образом, ни один из изученных ранее способов не позволяет из вложенной функции изменить значение переменной, объявленной внутри функции-родителя. Чтобы

решить эту проблему, следует объявить необходимые переменные с помощью ключевого слова `nonlocal` (листинг 11.32).

Листинг 11.32. Ключевое слово nonlocal

```
def func1(a):
    x = a
    def func2(b):
        nonlocal x     # Объявляем переменную как nonlocal
        print(x)
        x = b          # Можем изменить значение x в func1()
    return func2

f = func1(10)
f(5)                    # Выведет: 10
f(12)                   # Выведет: 5
f(3)                    # Выведет: 12
```

При использовании ключевого слова `nonlocal` следует помнить, что переменная обязательно должна существовать внутри функции-родителя. В противном случае будет выведено сообщение об ошибке.

11.11. Аннотации функций

В Python 3 функции могут содержать *аннотации*, которые вводят новый способ документирования. Теперь в заголовке функции допускается указывать предназначение каждого параметра, данные какого типа он может принимать, а также тип возвращаемого функцией значения. Аннотации имеют следующий формат:

```
def <Имя функции>(
    [<Параметр1>[: <Выражение>] [= <Значение по умолчанию>][, ...,
     <ПараметрN>[: <Выражение>] [= <Значение по умолчанию>]]]
    ) -> <Возвращаемое значение>:
    <Тело функции>
```

В параметрах `<Выражение>` и `<Возвращаемое значение>` можно указать любое допустимое выражение языка Python. Это выражение будет выполнено при создании функции. Пример указания аннотаций:

```
def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
    print(a, b)
```

В этом примере для переменной `a` создано описание `"Параметр1"`. Для переменной `b` выражение `10 + 5` является описанием, а число `3` — значением параметра по умолчанию. Кроме того, после закрывающей круглой скобки указан тип возвращаемого функцией значения: `None`. После создания функции все выражения будут выполнены, и результаты сохранятся в виде словаря в атрибуте `__annotations__` объекта функции. Для примера выведем значение этого атрибута:

```
>>> def func(a: "Параметр1", b: 10 + 5 = 3) -> None: pass

>>> func.__annotations__
{'a': 'Параметр1', 'b': 15, 'return': None}
```

ГЛАВА 12

Модули и пакеты

Модулем в языке Python называется любой файл с программным кодом. Каждый модуль может импортировать другой модуль, получая, таким образом, доступ к атрибутам (переменным, функциям и классам), объявленным внутри импортированного модуля. Следует заметить, что импортируемый модуль может содержать программу не только на языке Python — так, 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__`. Для запускаемого модуля он 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 обращались к этой инструкции 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 имен модулей с ключевыми словами, встроенными идентификаторами и названиями модулей, входящих в стандартную библиотеку.

За один раз можно импортировать сразу несколько модулей, записав их через запятую. Для 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 hasattr_math(attr):
    if hasattr(math, attr):
        return "Атрибут существует"
```

```
    else:
        return "Атрибут не существует"
print(hasattr_math("pi"))        # Атрибут существует
print(hasattr_math("x"))         # Атрибут не существует
```

Если название модуля слишком длинное, и его неудобно указывать каждый раз для доступа к атрибутам 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 содержимое импортированного модуля доступно 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. Для примера создадим модуль под названием tests.py, в котором определим переменную `x` (листинг 12.6).

Листинг 12.6. Содержимое модуля tests.py

```
# -*- coding: utf-8 -*-
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: utf-8 -*-
import tests             # Подключаем файл tests.py
x = 22
print(tests.x)           # Значение переменной x внутри модуля
print(x)                 # Значение переменной x в основной программе
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, поскольку одноименные переменные расположены в разных пространствах имен.

Как говорилось еще в *главе 1*, перед собственно выполнением каждый модуль Python компилируется, преобразуясь в особое внутреннее представление (байт-код), — это делается для ускорения выполнения кода. Файлы с откомпилированным кодом хранятся в папке `__pycache__`, автоматически создающейся в папке, где находится сам файл с исходным, неоткомпилированным кодом модуля, и имеют имена вида <имя файла с исходным кодом>.cpython-<первые две цифры номера версии Python>.pyc. Так, при запуске на исполнение нашего файла tests.py откомпилированный код будет сохранен в файле tests.cpython-34.pyc.

Следует заметить, что для импортирования модуля достаточно иметь только файл с откомпилированным кодом, файл с исходным кодом в этом случае не нужен. Для примера переименуйте файл tests.py (например, в tests1.py), скопируйте файл tests.cpython-34.pyc из папки __pycache__ в папку с основной программой и переименуйте его в tests.pyc, а затем 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` (или `from`, речь о которой пойдет позже). 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. Для примера выведем ключи словаря `modules`, предварительно отсортировав их (листинг 12.8).

Листинг 12.8. Вывод ключей словаря `modules`

```
# -*- coding: utf-8 -*-
import tests, sys                    # Подключаем модули tests и sys
print(sorted(sys.modules.keys()))
input()
```

Инструкция `import` требует явного указания объекта модуля. Так, нельзя передать название модуля в виде строки. Чтобы подключить модуль, название которого создается динамически в зависимости от определенных условий, следует воспользоваться функцией `__import__()`. Для примера подключим модуль tests.py с помощью функции `__import__()` (листинг 12.9).

Листинг 12.9. Использование функции `__import__()`

```
# -*- coding: utf-8 -*-
s = "test" + "s"              # Динамическое создание названия модуля
m = __import__(s)             # Подключение модуля tests
print(m.x)                    # Вывод значения атрибута x
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: utf-8 -*-
import tests
print(dir(tests))
print(sorted(tests.__dict__.keys()))
input()
```

12.2. Инструкция *from*

Для импортирования только определенных идентификаторов из модуля можно воспользоваться инструкцией `from`. Ее формат таков:

```
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 варианта позволяют импортировать модуль и сделать доступными только указанные идентификаторы. Для длинных имен можно назначить псевдонимы, указав их 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: utf-8 -*-
from math import pi, floor as f
print(pi)                                  # Вывод числа pi
# Вызываем функцию floor() через идентификатор f
print(f(5.49))                             # Выведет: 5
input()
```

Идентификаторы можно разместить на нескольких строках, указав их названия через запятую внутри круглых скобок:

```
from math import (pi, floor,
                  sin, cos)
```

Третий вариант формата инструкции `from` позволяет импортировать из модуля все идентификаторы. Для примера импортируем все идентификаторы из модуля `math` (листинг 12.12).

Листинг 12.12. Импорт всех идентификаторов из модуля

```
# -*- coding: utf-8 -*-
from math import *     # Импортируем все идентификаторы из модуля math
print(pi)              # Вывод числа pi
print(floor(5.49))     # Вызываем функцию floor()
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: utf-8 -*-
s = "Значение из модуля module1"
```

Содержимое файла module2.py приведено в листинге 12.14.

Листинг 12.14. Содержимое файла module2.py

```
# -*- coding: utf-8 -*-
s = "Значение из модуля module2"
```

Исходный код основной программы приведен в листинге 12.15.

Листинг 12.15. Исходный код основной программы

```
# -*- coding: utf-8 -*-
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"
input()
```

Итак, в обоих модулях определена переменная с именем `s`. Размещаем все файлы в одной папке, а затем запускаем основную программу с помощью двойного щелчка на значке файла. 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 *`. Идентификаторы внутри списка указываются в виде строки. Создадим файл module3.py (листинг 12.16).

Листинг 12.16. Использование атрибута __all__

```
# -*- coding: utf-8 -*-
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: utf-8 -*-
from module3 import *
print(sorted(vars().keys())) # Получаем список всех идентификаторов
input()
```

После запуска основной программы (с помощью двойного щелчка на значке файла) получим следующий результат:

```
['__builtins__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', '_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__', '__loader__', '__name__', '__package__',
'__spec__', 'x', 'y', 'z']
```

Обратите внимание на то, что переменная _s в этом случае не импортируется, т. к. ее имя начинается с символа подчеркивания.

12.3. Пути поиска модулей

До сих пор мы размещали модули в одной папке с файлом основной программы. 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 кодом основной программы;
- значение переменной окружения PYTHONPATH. Для добавления переменной в меню **Пуск** выбираем пункт **Панель управления** (или **Настройка | Панель управления**). В открывшемся окне выбираем пункт **Система** и, если мы пользуемся Windows Vista или более новой версией этой системы, щелкаем на ссылке **Дополнительные параметры системы**. 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. Закончив, не забудем нажать кнопки **OK** обоих открытых окон. 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:\Python34\Lib\site-packages. Названия таких файлов могут быть произвольными, главное, чтобы они имели расширение pth. Каждый путь (абсолютный или относительный) должен быть расположен на отдельной строке. Для примера создайте файл mypath.pth в каталоге C:\Python34\Lib\site-packages со следующим содержимым:

```
# Это комментарий
C:\folder1
C:\folder2
```

> ***Примечание***
>
> 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 его начало — с помощью метода insert() (листинг 12.18).

Листинг 12.18. Изменение списка путей поиска модулей

```
# -*- coding: utf-8 -*-
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)
input()
```

В этом примере мы добавили папку C:\folder2 в начало списка. Теперь, если в каталогах C:\folder1 и C:\folder2 существуют одноименные модули, будет использоваться модуль из папки C:\folder2, а не из папки 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 в указании пути необходимо удвоить (экранировать) каждый встреченный слеш:

```
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() из модуля imp. Формат функции:

```
from imp import reload
reload(<Объект модуля>)
```

В качестве примера создадим модуль tests2.py со следующим содержимым:

```
# -*- coding: utf-8 -*-
x = 150
```

Подключим этот модуль в окне **Python Shell** редактора IDLE и выведем текущее значение переменной x:

```
>>> import sys
>>> sys.path.append(r"C:\book") # Добавляем путь к папке с модулем
>>> import tests2               # Подключаем модуль tests2.py
>>> print(tests2.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 tests2
>>> print(tests2.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

```
>>> from imp import reload
>>> reload(tests2)                # Перезагружаем модуль
<module 'tests2' from 'C:\book\tests2.py'>
>>> print(tests2.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, на языке 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: utf-8 -*-
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: utf-8 -*-
msg = "Модуль {0}".format(__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: utf-8 -*-

# 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

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:\ должен быть записан 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 столь 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 folder1\folder2\:

```
# -*- coding: utf-8 -*-
__all__ = ["module2", "module3"]
```

Теперь изменим содержимое основного файла main.py (листинг 12.23) и запустим его.

Листинг 12.23. Содержимое файла main.py

```
# -*- coding: utf-8 -*-
from folder1.folder2 import *
print(module2.msg)          # Выведет: Модуль folder1.folder2.module2
print(module3.msg)          # Выведет: Модуль folder1.folder2.module3
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 указывать одни 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: utf-8 -*-

# Импорт модуля module2.py из текущего каталога
from . import module2 as m1
var1 = "Значение из: {0}".format(m1.msg)
from .module2 import msg as m2
var2 = "Значение из: {0}".format(m2)

# Импорт модуля module1.py из родительского каталога
from .. import module1 as m3
var3 = "Значение из: {0}".format(m3.msg)
from ..module1 import msg as m4
var4 = "Значение из: {0}".format(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: utf-8 -*-
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
input()
```

При импортировании модуля внутри пакета с помощью инструкции `import` важно помнить, что в Python 3 производится *абсолютный импорт*. Если при запуске Python-программы с помощью двойного щелчка на ее файле автоматически добавляется путь к каталогу с исполняемым файлом, то при импорте внутри пакета этого не происходит. Поэтому если изменить содержимое модуля module3.py показанным далее способом, то мы получим сообщение об ошибке или загрузим совсем другой модуль:

```
# -*- coding: utf-8 -*-
import module2        # Ошибка! Поиск модуля по абсолютному пути
var1 = "Значение из: {0}".format(module2.msg)
var2 = var3 = var4 = 0
```

В этом примере мы попытались импортировать модуль module2.py из модуля module3.py. При этом с помощью двойного щелчка мы запускаем файл main.py (см. листинг 12.25). Поскольку импорт внутри пакета выполняется по абсолютному пути, поиск модуля module2.py не будет производиться в папке folder1\folder2\. В результате модуль не будет найден. Если в путях поиска модулей находится модуль с таким же именем, то будет импортирован модуль, который мы и не предполагали подключать.

Чтобы подключить модуль, расположенный в той же папке внутри пакета, необходимо воспользоваться относительным импортом с помощью инструкции `from`:

```
from . import module2
```

Или указать полный путь относительно корневого каталога пакета:

```
import folder1.folder2.module2 as module2
```

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 удобных сущностей — объектов, а также организовать связи между этими объектами.

Основным «кирпичиком» ООП является класс. *Класс* — это сложный тип данных, включающий набор переменных и функций для управления значениями, хранящимися в этих переменных. Переменные называют *атрибутами*, а функции — *методами*. Класс является фабрикой объектов, т. е. позволяет создать неограниченное количество экземпляров, 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 именования переменных. После названия класса в круглых скобках можно указать один или несколько базовых классов через запятую. Если же класс не наследует базовые классы, то круглые скобки можно не указывать. Следует заметить, что все выражения внутри инструкции `class` выполняются при создании класса, а не его экземпляра. Для примера создадим класс, внутри которого просто выводится сообщение (листинг 13.1).

Листинг 13.1. Создание определения класса

```
# -*- coding: utf-8 -*-
class MyClass:
    """ Это строка документирования """
    print("Инструкции выполняются сразу")
input()
```

Этот пример содержит лишь определение класса `MyClass` и не создает экземпляр класса. Как только поток выполнения достигнет инструкции `class`, сообщение, указанное в функции `print()`, будет сразу выведено.

Создание атрибута класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, — с помощью инструкции `def`. Методам класса в первом параметре, который обязательно следует указать явно, автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть именем `self`, хотя это и не обязательно. Доступ к атрибутам и методам класса внутри определяемого метода производится через переменную `self` с помощью точечной нотации — к атрибуту `x` из метода класса можно обратиться так: `self.x`.

Чтобы использовать атрибуты и методы класса, необходимо создать экземпляр класса согласно следующему синтаксису:

```
<Экземпляр класса> = <Название класса>([<Параметры>])
```

При обращении к методам класса используется такой 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>
```

Определим класс `MyClass` с атрибутом `x` и методом `print_x()`, выводящим значение этого атрибута, а затем создадим экземпляр класса и вызовем метод (листинг 13.2).

Листинг 13.2. Создание атрибута и метода

```
class MyClass:
    def __init__(self):    # Конструктор
        self.x = 10        # Атрибут экземпляра класса
    def print_x(self):     # self — это ссылка на экземпляр класса
        print(self.x)      # Выводим значение атрибута
c = MyClass()              # Создание экземпляра класса
                           # Вызываем метод print_x()
c.print_x()                # self не указывается при вызове метода
print(c.x)                 # К атрибуту можно обратиться непосредственно
```

Для доступа к атрибутам и методам можно использовать и следующие функции:

- `getattr()` — возвращает значение атрибута по его названию, заданному в виде строки. С помощью этой функции можно сформировать имя 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>)`

  Вторым параметром метода `setattr()` можно передать имя несуществующего атрибута — в этом случае атрибут с указанным именем будет создан;

- `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 MyClass:
    def __init__(self):
        self.x = 10
    def get_x(self):
        return self.x
c = MyClass()                   # Создаем экземпляр класса
print(getattr(c, "x"))          # Выведет: 10
print(getattr(c, "get_x")())    # Выведет: 10
print(getattr(c, "y", 0))       # Выведет: 0, т. к. атрибут не найден
setattr(c, "y", 20)             # Создаем атрибут y
print(getattr(c, "y", 0))       # Выведет: 20
delattr(c, "y")                 # Удаляем атрибут y
print(getattr(c, "y", 0))       # Выведет: 0, т. к. атрибут не найден
print(hasattr(c, "x"))          # Выведет: True
print(hasattr(c, "y"))          # Выведет: False
```

Все атрибуты класса в языке Python являются открытыми (public), т. е. доступными для непосредственного изменения как из самого класса, так и из других классов и из основного кода программы.

Кроме того, атрибуты допускается 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 объекта класса и экземпляра класса

```
class MyClass:                  # Определяем пустой класс
    pass
MyClass.x = 50                  # Создаем атрибут объекта класса
c1, c2 = MyClass(), MyClass()   # Создаем два экземпляра класса
c1.y = 10                       # Создаем атрибут экземпляра класса
c2.y = 20                       # Создаем атрибут экземпляра класса
print(c1.x, c1.y)               # Выведет: 50 10
print(c2.x, c2.y)               # Выведет: 50 20
```

В этом примере мы определяем пустой класс, разместив в нем оператор pass. Далее создаем атрибут объекта класса (x). Этот атрибут будет доступен всем создаваемым экземплярам класса. Затем создаем два экземпляра класса и добавляем одноименные атрибуты (y). Значения этих атрибутов будут разными в каждом экземпляре класса. Но если создать новый экземпляр (например, c3), то атрибут y в нем определен не будет. Таким образом, с помощью классов можно имитировать типы данных, поддерживаемые другими языками программирования (например, тип struct, доступный в языке C).

Очень важно понимать разницу между атрибутами объекта класса и атрибутами экземпляра класса. *Атрибут объекта класса* доступен всем экземплярам класса, но после изменения атрибута значение изменится во всех экземплярах класса. *Атрибут экземпляра класса* может хранить уникальное значение для каждого экземпляра, и изменение его в одном экземпляре класса не затронет значения одноименного атрибута в других экземплярах того же класса. Рассмотрим это на примере, создав класс с атрибутом объекта класса (x) и атрибутом экземпляра класса (y):

```
class MyClass:
    x = 10                          # Атрибут объекта класса
    def __init__(self):
        self.y = 20                 # Атрибут экземпляра класса
```

Теперь создадим два экземпляра этого класса:

```
c1 = MyClass()                      # Создаем экземпляр класса
c2 = MyClass()                      # Создаем экземпляр класса
```

Выведем значения атрибута x, а затем изменим значение и опять произведем вывод:

```
print(c1.x, c2.x)                   # 10 10
MyClass.x = 88                      # Изменяем атрибут объекта класса
print(c1.x, c2.x)                   # 88 88
```

Как видно из примера, изменение атрибута объекта класса затронуло значение в двух экземплярах класса сразу. Теперь произведем аналогичную операцию с атрибутом y:

```
print(c1.y, c2.y)                   # 20 20
c1.y = 88                           # Изменяем атрибут экземпляра класса
print(c1.y, c2.y)                   # 88 20
```

В этом случае изменилось значение атрибута только в экземпляре c1.

Следует также учитывать, что в одном классе могут одновременно существовать атрибут объекта и атрибут экземпляра с одним именем. Изменение атрибута объекта класса мы производили следующим образом:

```
MyClass.x = 88                      # Изменяем атрибут объекта класса
```

Если после этой инструкции вставить инструкцию

```
c1.x = 200                          # Создаем атрибут экземпляра
```

то будет создан атрибут экземпляра класса, а не изменено значение атрибута объекта класса. Чтобы увидеть разницу, выведем значения атрибута:

```
print(c1.x, MyClass.x)              # 200 88
```

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>]]):
    <Инструкции>
```

С помощью метода `__init__()` можно присвоить начальные значения атрибутам класса. При создании экземпляра класса параметры этого метода 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 MyClass:
    def __init__(self, value1, value2): # Конструктор
        self.x = value1
        self.y = value2
c = MyClass(100, 300)                    # Создаем экземпляр класса
print(c.x, c.y)                          # 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. Впрочем, поскольку интерпретатор самостоятельно заботится об удалении объектов, использование деструктора в языке Python не имеет особого смысла.

Листинг 13.6. Метод __del__()

```
class MyClass:
    def __init__(self): # Конструктор класса
        print("Вызван метод __init__()")
    def __del__(self):  # Деструктор класса
        print("Вызван метод __del__()")
c1 = MyClass()          # Выведет: Вызван метод __init__()
del c1                  # Выведет: Вызван метод __del__()
c2 = MyClass()          # 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 реализован доступ ко всем атрибутам и методам класса `Class1` (листинг 13.7).

Листинг 13.7. Наследование

```
class Class1:          # Базовый класс
    def func1(self):
        print("Метод func1() класса Class1")
    def func2(self):
        print("Метод func2() класса Class1")

class Class2(Class1): # Класс Class2 наследует класс Class1
    def func3(self):
        print("Метод func3() класса Class2")

c = Class2()          # Создаем экземпляр класса Class2
c.func1()             # Выведет: Метод func1() класса Class1
c.func2()             # Выведет: Метод func2() класса Class1
c.func3()             # Выведет: Метод 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* или *суперклассом*, а класс `Class2` — *производным* 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 func1(self):
        print("Метод 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 func1(self):
        print("Метод func1() класса Class2")
        Class1.func1(self)     # Вызываем метод базового класса

c = Class2()                   # Создаем экземпляр класса Class2
c.func1()                      # Вызываем метод func1()
```

Выведет:

```
Конструктор производного класса
Конструктор базового класса
Метод func1() класса Class2
Метод func1() класса Class1
```

Внимание!

Конструктор базового класса автоматически не вызывается, если он переопределен в производном классе.

Чтобы вызвать одноименный метод из базового класса, можно также воспользоваться функцией `super()`. Формат функции:

```
super([<Класс>, <Указатель self>])
```

С помощью функции `super()` инструкцию

```
Class1.__init__(self)          # Вызываем конструктор базового класса
```

можно записать так:

```
super().__init__()             # Вызываем конструктор базового класса
```

или так:

```
super(Class2, self).__init__() # Вызываем конструктор базового класса
```

Обратите внимание на то, что при использовании функции `super()` не нужно явно передавать указатель `self` в вызываемый метод. Кроме того, в первом параметре функции `super()` указывается производный класс, а не базовый. Поиск идентификатора будет производиться во всех базовых классах. Результатом поиска станет первый найденный идентификатор в цепочке наследования.

Примечание

В последних версиях Python 2 существовало два типа классов: «классические» классы и классы нового стиля. Классы нового стиля должны были явно наследовать класс `object`. В Python 3 все классы являются классами нового стиля, но наследуют класс `object` неявно. Таким образом, все классы верхнего уровня являются наследниками этого класса, даже если он не указан в списке наследования. «Классические» классы (в понимании Python 2) в Python 3 больше не поддерживаются.

13.4. Множественное наследование

В определении класса в круглых скобках можно указать сразу несколько базовых классов через запятую. Рассмотрим множественное наследование на примере (листинг 13.9).

Листинг 13.9. Множественное наследование

```
class Class1:          # Базовый класс для класса Class2
    def func1(self):
        print("Метод func1() класса Class1")

class Class2(Class1): # Класс Class2 наследует класс Class1
    def func2(self):
        print("Метод func2() класса Class2")

class Class3(Class1): # Класс Class3 наследует класс Class1
    def func1(self):
        print("Метод func1() класса Class3")
```

```
    def func2(self):
        print("Метод func2() класса Class3")
    def func3(self):
        print("Метод func3() класса Class3")
    def func4(self):
        print("Метод func4() класса Class3")

class Class4(Class2, Class3): # Множественное наследование
    def func4(self):
        print("Метод func4() класса Class4")

c = Class4()                # Создаем экземпляр класса Class4
c.func1()                   # Выведет: Метод func1() класса Class3
c.func2()                   # Выведет: Метод func2() класса Class2
c.func3()                   # Выведет: Метод func3() класса Class3
c.func4()                   # Выведет: Метод func4() класса Class4
```

Метод `func1()` определен в двух классах: `Class1` и `Class3`. Так как вначале просматриваются все базовые классы, непосредственно указанные в определении текущего класса, метод `func1()` будет найден в классе `Class3` (поскольку он указан в числе базовых классов в определении `Class4`), а не в классе `Class1`.

Метод `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 нем. Чтобы наследовать метод из класса `Class3`, следует указать это явным образом. Переделаем определение класса `Class4` из предыдущего примера и наследуем метод `func2()` из класса `Class3` (листинг 13.10).

Листинг 13.10. Указание класса при наследовании метода

```
class Class4(Class2, Class3): # Множественное наследование
    # Наследуем func2() из класса Class3, а не из класса Class2
    func2 = Class3.func2
    def func4(self):
        print("Метод func4() класса Class4")
```

Вернемся к листингу 13.9. Метод `func3()` определен только в классе `Class3`, поэтому метод наследуется от этого класса. Метод `func4()`, определенный в классе `Class3`, переопределяется в производном классе.

Если же искомый метод найден в производном классе, то вся иерархия наследования просматриваться не будет.

Для получения перечня базовых классов можно воспользоваться атрибутом `__bases__`. В качестве значения атрибут возвращает кортеж. В качестве примера выведем базовые классы для всех классов из предыдущего примера:

```
print(Class1.__bases__)
print(Class2.__bases__)
print(Class3.__bases__)
print(Class4.__bases__)
```

Выведет:

```
(<class 'object'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class2'>, <class '__main__.Class3'>)
```

Рассмотрим порядок поиска идентификаторов при сложной иерархии множественного наследования (листинг 13.11).

Листинг 13.11. Поиск идентификаторов при множественном наследовании

```
class Class1: x = 10
class Class2(Class1): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c = Class7()
print(c.x)
```

Последовательность поиска атрибута `x` будет такой:

```
Class7 -> Class4 -> Class3 – > Class6 -> Class5 -> Class2 -> Class1
```

Получить всю цепочку наследования позволяет атрибут `__mro__`:

```
print(Class7.__mro__)
```

Результат выполнения:

```
(<class '__main__.Class7'>, <class '__main__.Class4'>,
 <class '__main__.Class3'>, <class '__main__.Class6'>,
 <class '__main__.Class5'>, <class '__main__.Class2'>,
 <class '__main__.Class1'>, <class 'object'>)
```

13.4.1. Примеси и их использование

Множественное наследование, поддерживаемое Python, позволяет реализовать интересный способ расширения функциональности классов с помощью так называемых *примесей* (mixins). Примесь — это класс, включающий какие-либо атрибуты и методы, которые необходимо добавить к другим классам. Объявляются они точно так же, как и обычные классы (листинг 13.12).

Листинг 13.12. Класс-примесь

```
class Mixin:                          # Определяем сам класс-примесь
    attr = 0                          # Определяем атрибут примеси
    def mixin_method(self):           # Определяем метод примеси
        print("Метод примеси")
```

Теперь объявим два класса, добавим к их функциональности ту, что определена в классе-примеси `Mixin`, и проверим ее в действии (листинг 13.13).

Листинг 13.13. Расширение функциональности классов посредством примеси

```
class Class1 (Mixin):
    def method1(self):
        print("Метод класса Class1")

class Class2 (Class1, Mixin):
    def method2(self):
        print("Метод класса Class2")

c1 = Class1()
c1.method1()
c1.mixin_method()                    # Class1 поддерживает метод примеси

c2 = Class2()
c2.method1()
c2.method2()
c2.mixin_method()                    # Class2 также поддерживает метод примеси
```

Вот результат выполнения кода, приведенного в листинге 13.13:

```
Метод класса Class1
Метод примеси
Метод класса Class1
Метод класса Class2
Метод примеси
```

Примеси активно применяются в различных дополнительных библиотеках — в частности, в популярной библиотеке Web-программирования Django.

13.5. Специальные методы

Классы поддерживают следующие специальные методы:

- **`__call__()` — позволяет обработать вызов экземпляра класса как вызов функции. Формат метода:**

  ```
  __call__(self[, <Параметр1>[, ..., <ПараметрN>]])
  ```

  Пример:

  ```
  class MyClass:
      def __init__(self, m):
          self.msg = m
      def __call__(self):
          print(self.msg)
  c1 = MyClass("Значение1") # Создание экземпляра класса
  c2 = MyClass("Значение2") # Создание экземпляра класса
  c1()                      # Выведет: Значение1
  c2()                      # Выведет: Значение2
  ```

- `__getattr__(self, <Атрибут>)` — **вызывается при обращении к несуществующему атрибуту класса. Пример:**

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattr__(self, attr):
        print("Вызван метод __getattr__()")
        return 0
c = MyClass()
# Атрибут i существует
print(c.i)      # Выведет: 20. Метод __getattr__() не вызывается
# Атрибут s не существует
print(c.s)      # Выведет: Вызван метод __getattr__() 0
```

- `__getattribute__(self, <Атрибут>)` — **вызывается при обращении к любому атрибуту класса. Необходимо учитывать, что использование точечной нотации (для обращения к атрибуту класса) внутри этого метода приведет к зацикливанию. Чтобы избежать зацикливания, следует вызывать метод** `__getattribute__()` **объекта** `object`. **Внутри метода нужно вернуть значение атрибута или возбудить исключение** `AttributeError`. **Пример:**

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattribute__(self, attr):
        print("Вызван метод __getattribute__()")
        return object.__getattribute__(self, attr) # Только так!!!
c = MyClass()
print(c.i)      # Выведет: Вызван метод __getattribute__() 20
```

- `__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, что приведет к зацикливанию. Пример:**

```
class MyClass:
    def __setattr__(self, attr, value):
        print("Вызван метод __setattr__()")
        self.__dict__[attr] = value                # Только так!!!
c = MyClass()
c.i = 10        # Выведет: Вызван метод __setattr__()
print(c.i)      # Выведет: 10
```

- `__delattr__(self, <Атрибут>)` — **вызывается при удалении атрибута с помощью инструкции** `del <Экземпляр класса>.<Атрибут>`;

- `__len__(self)` — **вызывается при использовании функции** `len()`, **а также для проверки объекта на логическое значение при отсутствии метода** `__bool__()`. **Метод должен возвращать положительное целое число. Пример:**

```
class MyClass:
    def __len__(self):
        return 50
```

```
c = MyClass()
print(len(c))              # Выведет: 50
```

- `__bool__(self)` — **вызывается при использовании функции** `bool()`;
- `__int__(self)` — **вызывается при преобразовании объекта в целое число с помощью функции** `int()`;
- `__float__(self)` — **вызывается при преобразовании объекта в вещественное число с помощью функции** `float()`;
- `__complex__(self)` — **вызывается при преобразовании объекта в комплексное число с помощью функции** `complex()`;
- `__round__(self, n)` — **вызывается при использовании функции** `round()`;
- `__index__(self)` — **вызывается при использовании функций** `bin()`, `hex()` **и** `oct()`;
- `__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 функции** `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 MyClass:
    def __init__(self, m):
        self.msg = m
    def __repr__(self):
        return "Вызван метод __repr__() {0}".format(self.msg)
    def __str__(self):
        return "Вызван метод __str__() {0}".format(self.msg)
c = MyClass("Значение")
print(repr(c)) # Выведет: Вызван метод __repr__() Значение
print(str(c))  # Выведет: Вызван метод __str__() Значение
print(c)       # Выведет: Вызван метод __str__() Значение
```

- `__hash__(self)` — **этот метод следует переопределить, если экземпляр класса планируется использовать в качестве ключа словаря или внутри множества. Пример:**

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __hash__(self):
        return hash(self.x)
c = MyClass(10)
d = {}
d[c] = "Значение"
print(d[c]) # Выведет: Значение
```

Классы поддерживают еще несколько специальных методов, которые применяются лишь в особых случаях и будут рассмотрены в *главе 15*.

13.6. 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.__truediv__(y)`;
- `y / x` — **деление (экземпляр класса справа):** `x.__rtruediv__(y)`;
- `x /= y` — **деление и присваивание:** `x.__itruediv__(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 13.14.

Листинг 13.14. Пример перегрузки математических операторов

```
class MyClass:
    def __init__(self, y):
        self.x = y
    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")
        self.x += y
        return self
c = MyClass(50)
print(c + 10)                  # Выведет: Экземпляр слева 60
print(20 + c)                  # Выведет: Экземпляр справа 70
c += 30                        # Выведет: Сложение с присваиванием
print(c.x)                     # 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` — **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)`;
- `y in x` — **проверка на вхождение:** `x.__contains__(y)`.

Пример перегрузки операторов сравнения приведен в листинге 13.15.

Листинг 13.15. Пример перегрузки операторов сравнения

```
class MyClass:
    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
c = MyClass()
print("Равно" if c == 50 else "Не равно") # Выведет: Равно
print("Равно" if c == 51 else "Не равно") # Выведет: Не равно
print("Есть" if 5 in c else "Нет")        # Выведет: Есть
```

13.7. Статические методы и методы класса

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса (статический метод). 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 13.16.

Листинг 13.16. Статические методы

```
class MyClass:
    @staticmethod
    def func1(x, y):                # Статический метод
        return x + y
    def func2(self, x, y):          # Обычный метод в классе
        return x + y
    def func3(self, x, y):
        return MyClass.func1(x, y) # Вызов из метода класса

print(MyClass.func1(10, 20))        # Вызываем статический метод
c = MyClass()
print(c.func2(15, 6))               # Вызываем метод класса
print(c.func1(50, 12))              # Вызываем статический метод
                                    # через экземпляр класса
print(c.func3(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 13.17.

Листинг 13.17. Методы класса

```
class MyClass:
    @classmethod
    def func(cls, x): # Метод класса
        print(cls, x)
MyClass.func(10)      # Вызываем метод через название класса
c = MyClass()
c.func(50)            # Вызываем метод класса через экземпляр
```

13.8. 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 13.18).

Листинг 13.18. Абстрактные методы

```
class Class1:
    def func(self, x):      # Абстрактный метод
        # Возбуждаем исключение с помощью raise
        raise NotImplementedError("Необходимо переопределить метод")

class Class2(Class1):       # Наследуем абстрактный метод
    def func(self, x):      # Переопределяем метод
        print(x)

class Class3(Class1):       # Класс не переопределяет метод
    pass

c2 = Class2()
c2.func(50)                 # Выведет: 50
c3 = Class3()
try:                        # Перехватываем исключения
    c3.func(50)             # Ошибка. Метод func() не переопределен
except NotImplementedError as msg:
    print(msg)              # Выведет: Необходимо переопределить метод
```

В состав стандартной библиотеки входит модуль `abc`. В этом модуле определен декоратор `@abstractmethod`, который позволяет указать, что метод, перед которым расположен декора-

тор, является абстрактным. При попытке создать экземпляр класса-потомка, в котором не переопределен абстрактный метод, возбуждается исключение `TypeError`. Рассмотрим использование декоратора `@abstractmethod` на примере (листинг 13.19).

Листинг 13.19. Использование декоратора `@abstractmethod`

```
from abc import ABCMeta, abstractmethod
class Class1(metaclass=ABCMeta):
    @abstractmethod
    def func(self, x):      # Абстрактный метод
        pass

class Class2(Class1):       # Наследуем абстрактный метод
    def func(self, x):      # Переопределяем метод
        print(x)

class Class3(Class1):       # Класс не переопределяет метод
    pass

c2 = Class2()
c2.func(50)                 # Выведет: 50
try:
    c3 = Class3()           # Ошибка. Метод func() не переопределен
    c3.func(50)
except TypeError as msg:
    print(msg)              # Can't instantiate abstract class Class3
                            # with abstract methods func
```

Имеется возможность создания абстрактных статических методов и абстрактных методов класса, для чего необходимые декораторы указываются одновременно, друг за другом. Для примера объявим класс с абстрактными статическим методом и методом класса (листинг 13.20).

Листинг 13.20. Абстрактный статический метод и абстрактный метод класса

```
from abc import ABCMeta, abstractmethod
class MyClass(metaclass=ABCMeta):
    @staticmethod
    @abstractmethod
    def static_func(self, x):      # Абстрактный статический метод
        pass

    @classmethod
    @abstractmethod
    def class_func(self, x):       # Абстрактный метод класса
        pass
```

Примечание

В версиях Python, предшествующих 3.3, для определения абстрактного статического метода предлагалось использовать декоратор `@abstractstaticmethod`, а для определения

абстрактного метода класса — декоратор `@abstractclassmethod`. Оба этих декоратора определены в том же модуле `abc`. Однако, начиная с Python 3.3, эти декораторы объявлены нерекомендованными к использованию.

13.9. 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 лишь внутри класса, но никак не вне его. 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 можно видеть, здесь перед идентификатором добавляется название класса с предваряющим символом подчеркивания. Приведем пример использования псевдочастных идентификаторов (листинг 13.21).

Листинг 13.21. Псевдочастные идентификаторы

```
class MyClass:
    def __init__(self, x):
        self.__privateVar = x
    def set_var(self, x):          # Изменение значения
        self.__privateVar = x
    def get_var(self):             # Получение значения
        return self.__privateVar
c = MyClass(10)                    # Создаем экземпляр класса
print(c.get_var())                 # Выведет: 10
c.set_var(20)                      # Изменяем значение
print(c.get_var())                 # Выведет: 20
try:                               # Перехватываем ошибки
    print(c.__privateVar)          # Ошибка!!!
except AttributeError as msg:
    print(msg)                     # Выведет: 'MyClass' object has
                                   # no attribute '__privateVar'
c._MyClass__privateVar = 50        # Значение псевдочастных атрибутов
                                   # все равно можно изменить
print(c.get_var())                 # Выведет: 50
```

Можно также 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 13.22).

Листинг 13.22. Атрибут __slots__

```
class MyClass:
    __slots__ = ["x", "y"]
    def __init__(self, a, b):
        self.x, self.y = a, b
c = MyClass(1, 2)
print(c.x, c.y)                  # Выведет: 1 2
c.x, c.y = 10, 20                # Изменяем значения атрибутов
print(c.x, c.y)                  # Выведет: 10 20
try:                             # Перехватываем исключения
    c.z = 50                     # Атрибут z не указан в __slots__,
                                 # поэтому возбуждается исключение
except AttributeError as msg:
    print(msg)                   # 'MyClass' object has no attribute 'z'
```

13.10. Свойства класса

Внутри класса можно создать идентификатор, через который в дальнейшем будут производиться операции получения и изменения значения какого-либо атрибута, а также его удаления. 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 13.23).**

Листинг 13.23. Свойства класса

```
class MyClass:
    def __init__(self, value):
        self.__var = value
    def get_var(self):           # Чтение
        return self.__var
    def set_var(self, value):    # Запись
        self.__var = value
    def del_var(self):           # Удаление
        del self.__var
    v = property(get_var, set_var, del_var, "Строка документирования")
c = MyClass(5)
c.v = 35                         # Вызывается метод set_var()
print(c.v)                       # Вызывается метод get_var()
del c.v                          # Вызывается метод del_var()
```

Python поддерживает альтернативный метод определения свойств — с помощью методов `property()`, `setter()` и `deleter()`, которые используются в декораторах. Пример их использования приведен в листинге 13.24.

Листинг 13.24. Методы `getter()`, `setter()` и `deleter()`

```
class MyClass:
    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
c = MyClass(5)
c.v = 35                                  # Запись
print(c.v)                                # Чтение
del c.v                                   # Удаление
```

Имеется возможность определить абстрактное свойство — в этом случае все реализующие его методы должны быть переопределены в подклассе. Выполняется это с помощью знакомого нам декоратора `@abstractmethod` из модуля `abc`. Пример определения абстрактного свойства показан в листинге 13.25.

Листинг 13.25. Определение абстрактного свойства

```
from abc import ABCMeta, abstractmethod
class MyClass1(metaclass=ABCMeta):
    def __init__(self, value):
        self.__var = value
    @property
    @abctractmethod
    def v(self):                          # Чтение
        return self.__var
    @v.setter
    @abctractmethod
    def v(self, value):                   # Запись
        self.__var = value
    @v.deleter
    @abctractmethod
    def v(self):                          # Удаление
        del self.__var
```

Примечание

В версиях Python, предшествующих 3.3, для определения абстрактного свойства применялся декоратор `@abstractproperty`, определенный в модуле `abc`. Однако, начиная с Python 3.3, этот декоратор объявлен нерекомендованным к использованию.

13.11. Декораторы классов

В языке Python, помимо декораторов функций, поддерживаются также *декораторы классов*, которые позволяют изменить поведение самих классов. В качестве параметра декоратор принимает ссылку на объект класса, поведение которого необходимо изменить, и должен возвращать ссылку на тот же класс или какой-либо другой. Пример декорирования класса показан в листинге 13.26.

Листинг 13.26. Декораторы классов

```
def deco(C):                        # Принимает объект класса
    print("Внутри декоратора")      # Производит какие-то действия
    return C                        # Возвращает объект класса

@deco
class MyClass:
    def __init__(self, value):
        self.v = value
c = MyClass(5)
print(c.v)
```

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 выполнение программы 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 программы, которые можно выявить только по результатам ее работы. Как правило, интерпретатор не предупреждает о наличии такой ошибки, и программа будет успешно выполняться, но результат ее выполнения будет не тем, на который мы рассчитывали. Выявить и исправить такие ошибки достаточно трудно;
- *ошибки времени выполнения* — это ошибки, которые возникают во время работы программы. Причиной являются события, не предусмотренные программистом. Классическим примером служит деление на ноль:

```
>>> def test(x, y): return x / y

>>> test(4, 2)                          # Нормально
2.0
>>> 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: division 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>[ as <Объект исключения>]]:
    <Блок, выполняемый при возникновении исключения>
[...
except [<ИсключениеN>[ as <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 уровня. Если исключение в программе вообще нигде не обрабатывается, оно 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 инструкции, расположенной сразу после обработчика. В нашем примере управление будет передано инструкции, выводящей значение переменной `x`: (`print(x)`). Обратите внимание на то, что инструкция `print("Выражение после вложенного обработчика")` выполнена 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) as err:
    print(err.__class__.__name__) # Название класса исключения
    print(err)                      # Текст сообщения об ошибке
```

Результат выполнения:

```
ZeroDivisionError
division by zero
```

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 14.5.

Листинг 14.5. 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 примера показан в листинге 14.6.

Листинг 14.6. Результат выполнения листинга 14.5

```
Type:  <class 'ZeroDivisionError'>
Value: division by zero
Trace: <traceback object at 0x031DE3C8>

 ----------print_exception()-----------
Traceback (most recent call last):
  File "<pyshell#1>", line 2, in <module>
ZeroDivisionError: division by zero

 --------------print_tb()--------------
  File "<pyshell#1>", line 2, in <module>

 ----------format_exception()----------
['Traceback (most recent call last):\n', '  File "<pyshell#1>", line 2,
in <module>\n', 'ZeroDivisionError: division by zero\n']

 --------format_exception_only()---------
['ZeroDivisionError: division 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 инструкции 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`. Для 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` инструкции 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#17>", line 2, in <module>
    x = 10 / 0
ZeroDivisionError: division by zero
```

В качестве примера переделаем нашу программу суммирования произвольного количества целых чисел, введенных пользователем (см. листинг 4.16), 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: utf-8 -*-
print("Введите слово 'stop' для получения результата")
summa = 0
while True:
    x = 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)
input()
```

Процесс ввода значений и получения результата выглядит так (значения, введенные пользователем, выделены полужирным шрифтом):

```
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
```

14.2. Инструкция *with...as*

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.

Для работы с протоколом предназначена инструкция `with...as`. Инструкция имеет следующий формат:

```
with <Выражение1>[ as <Переменная>][, ...,
     <ВыражениеN>[ as <Переменная>]]:
    <Блок, в котором перехватываем исключения>
```

Вначале вычисляется `<Выражение1>`, 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 инструкции 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 инструкций, 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

```
class MyClass:
    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 MyClass():
    print("Блок внутри with")
print("\nПоследовательность при наличии исключения:")
with MyClass() 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:\book\test.py", line 20, in <module>
    raise TypeError("Исключение TypeError")
TypeError: Исключение TypeError
```

Некоторые встроенные объекты поддерживают протокол по умолчанию — 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

```
with open("test.txt", "a", encoding="utf-8") 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 представляют собой классы. Иерархия встроенных классов исключений показана в листинге 14.12.

Листинг 14.12. Иерархия встроенных классов исключений

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError, OverflowError, ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
        LookupError
            IndexError, KeyError
        MemoryError
        NameError
            UnboundLocalError
```

```
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError, ConnectionAbortedError, ConnectionRefusedError,
                ConnectionResetError
            FileExistsError
            FileNotFoundError
            InterruptedError
            IsADirectoryError
            NotADirectoryError
            PermissionError
            ProcessLookupError
            TimeoutError
        ReferenceError
        RuntimeError
            NotImplementedError
        SyntaxError
            IndentationError
                TabError
        SystemError
        TypeError
        ValueError
            UnicodeError
                UnicodeDecodeError, UnicodeEncodeError
                UnicodeTranslateError
        Warning
            BytesWarning, DeprecationWarning, FutureWarning, ImportWarning,
            PendingDeprecationWarning, ResourceWarning, RuntimeWarning,
            SyntaxWarning, UnicodeWarning, UserWarning
```

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`, **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` — **является классом самого верхнего уровня и базовым для всех прочих классов исключений;**
- `Exception` — **базовый класс для большинства встроенных в Python исключений. Именно его, а не** `BaseException`, **необходимо наследовать при создании пользовательского класса исключения;**

- `AssertionError` — возбуждается инструкцией `assert`;
- `AttributeError` — попытка обращения к несуществующему атрибуту объекта;
- `EOFError` — возбуждается функцией `input()` при достижении конца файла;
- `ImportError` — невозможно импортировать 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>;
- `MemoryError` — интерпретатору существенно не хватает оперативной памяти;
- `NameError` — попытка обращения к идентификатору до его определения;
- `NotImplementedError` — должно возбуждаться в абстрактных методах;
- `OSError` — базовый класс для всех исключений, возбуждаемых в ответ на возникновение ошибок в операционной системе (отсутствие запрошенного файла, недостаток места на диске и пр.);
- `OverflowError` — число, получившееся в результате выполнения арифметической операции, слишком велико, чтобы Python смог его обработать;
- `RuntimeError` — неклассифицированная ошибка времени выполнения;
- `StopIteration` — возбуждается методом `__next__()` как сигнал об окончании итераций;
- `SyntaxError` — синтаксическая ошибка;
- `SystemError` — ошибка в самой программе интерпретатора Python;
- `TabError` — в исходном коде программы встретился символ табуляции, использование которого для создания отступов недопустимо;
- `TypeError` — тип объекта не соответствует ожидаемому;
- `UnboundLocalError` — внутри функции переменной присваивается значение после обращения к одноименной глобальной переменной;
- `UnicodeDecodeError` — ошибка преобразования последовательности байтов в строку;
- `UnicodeEncodeError` — ошибка преобразования строки в последовательность байтов;
- `UnicodeTranslationError` — ошибка преобразования строки в другую кодировку;
- `ValueError` — переданный параметр не соответствует ожидаемому значению;
- `ZeroDivisionError` — попытка деления на ноль.

14.4. Пользовательские исключения

Для возбуждения пользовательских исключений предназначены две инструкции:

- `raise`;
- `assert`.

Инструкция `raise` возбуждает заданное исключение. Она имеет несколько вариантов формата:

```
raise <Экземпляр класса>
raise <Название класса>
raise <Экземпляр или название класса> from <Объект исключения>
raise
```

В *первом варианте формата* 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 as 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 as 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:\book\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 as err:
    print(err)          # Выведет: Описание исключения
```

Во *втором варианте формата* инструкции `raise` в первом параметре задается объект класса, а не экземпляр. Пример:

```
try:
    raise ValueError # Эквивалентно: raise ValueError()
except ValueError:
    print("Сообщение об ошибке")
```

В *третьем варианте формата* инструкции `raise` в первом параметре задается экземпляр класса или просто название класса, а во втором параметре указывается объект исключения. В этом случае объект исключения сохраняется в атрибуте `__cause__`. При обработке вложенных исключений эти данные используются для вывода информации не только о последнем исключении, но и о первоначальном исключении. Пример:

```
try:
    x = 1 / 0
except Exception as err:
    raise ValueError() from err
```

Результат выполнения:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the
following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError
```

Как видно из результата, мы получили информацию не только по исключению `ValueError`, но и по исключению `ZeroDivisionError`. Следует заметить, что при отсутствии инструкции `from` информация сохраняется неявным образом. Если убрать инструкцию `from` в предыдущем примере, то получим следующий результат:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another
exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError
```

Четвертый вариант формата инструкции `raise` позволяет повторно возбудить последнее исключение и обычно применяется в коде, следующем за инструкцией `except`. Пример:

```
class MyError(Exception): pass
try:
    raise MyError("Сообщение об ошибке")
except MyError as err:
    print(err)
    raise               # Повторно возбуждаем исключение
```

Результат выполнения:

```
Сообщение об ошибке
Traceback (most recent call last):
  File "C:\book\test.py", line 5, in <module>
    raise MyError("Сообщение об ошибке")
MyError: Сообщение об ошибке
```

Инструкция `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 as err:
    print(err) # Выведет: Сообщение об ошибке
```

ГЛАВА 15

Итераторы, контейнеры и перечисления

Язык Python поддерживает средства для создания классов особого назначения: итераторов, контейнеров и перечислений.

Итераторы **— это классы, генерирующие последовательности каких-либо значений. Такие классы мы можем задействовать, например, в циклах** `for`**:**

```
class MyIterator:                # Определяем класс-итератор
    . . .
it = MyIterator()                # Создаем его экземпляр
for v in it:                     # и используем в цикле for
    . . .
```

Контейнеры **— классы, которые могут выступать как последовательности (списки или кортежи) или отображения (словари). Мы можем обратиться к любому элементу экземпляра такого класса через его индекс или ключ:**

```
class MyList:                    # Определяем класс-список
    . . .
class MyDict:                    # Определяем класс-словарь
    . . .
lst, dct = MyList(), MyDict()    # Используем их
lst[0] = 1
dct["first"] = 578
print(lst[1]), print(dct["second"])
```

Перечисления **— особые классы, представляющие наборы каких-либо именованных величин. В этом смысле они аналогичны подобным типам данных, доступным в других языках программирования, — например, в C:**

```
from enum import Enum            # Импортируем базовый класс Enum
class Versions(Enum):            # Определяем класс-перечисление
    Python2.7 = "2.7"
    Python3.4 = "3.4"
                                 # Используем его
if python_version == Versions.Python3.4:
    . . .
```

Поддержка перечислений появилась в Python 3.4.

15.1. Итераторы

Для того чтобы превратить класс в итератор, нам следует переопределить в нем два специальных метода:

- `__iter__(self)` — говорит о том, что этот класс является итератором (поддерживает итерационный протокол, как говорят Python-программисты). Должен возвращать сам экземпляр этого класса, а также при необходимости может выполнять всевозможные предустановки.

  Если в классе одновременно определены методы `__iter__()` и `__getitem__()` (о нем рассказывалось в *главе 13*), предпочтение отдается первому методу;

- `__next__(self)` — вызывается при выполнении каждой итерации и должен возвращать очередное значение из последовательности. Если последовательность закончилась, в этом методе следует возбудить исключение `StopIteration`, которое сообщит вызывающему коду об окончании итераций.

Для примера рассмотрим класс, хранящий строку и на каждой итерации возвращающий очередной ее символ, начиная с конца (листинг 15.1).

Листинг 15.1. Класс-итератор

```
class ReverseString:
    def __init__(self, s):
        self.__s = s
    def __iter__(self):
        self.__i = 0
        return self
    def __next__(self):
        if self.__i > len(self.__s) - 1:
            raise StopIteration
        else:
            a = self.__s[-self.__i - 1]
            self.__i = self.__i + 1
            return a
```

Проверим его в действии:

```
>>> s = ReverseString("Python")
>>> for a in s: print(a, end="")
nohtyP
```

Результат вполне ожидаем — строка, выведенная задом наперед.

Также мы можем переопределить специальный метод `__len()__`, который вернет количество элементов в последовательности. И, разумеется, мы можем переопределить специальные методы `__str()__` и `__repr()__`, возвращающие строковое представление итератора. (Все эти методы рассмотрены в *главе 13*.)

Перепишем код нашего класса-итератора, добавив в него определение методов `__len()__` и `__str()__` (листинг 15.2 — часть кода опущена).

Листинг 15.2. Расширенный класс-итератор

```
class ReverseString:
    . . .
    def __len__(self):
        return len(self.__s)
    def __str__(self):
        return self.__s[::-1]
```

Теперь мы можем получить длину последовательности, хранящейся в экземпляре класса `ReverseString`, и его строковое представление:

```
>>> s = ReverseString("Python")
>>> print(len(s))
6
>>> print(str(s))
nohtyP
```

15.2. Контейнеры

Python позволяет создать как контейнеры-последовательности, аналогичные спискам и кортежам, так и контейнеры-отображения, т. е. словари. Сейчас мы узнаем, как это делается.

15.2.1. Контейнеры-последовательности

Чтобы класс смог реализовать функциональность последовательности, нам следует переопределить в нем следующие специальные методы:

- `__getitem__(self, <Индекс>)` — вызывается при извлечении элемента последовательности по его индексу с помощью операции `<Экземпляр класса>[<Индекс>]`. Должен возвращать значение, расположенное по этому индексу. Если индекс не является целым числом или срезом, должно возбуждаться исключение `TypeError`, если такового индекса не существует, следует возбудить исключение `IndexError`;
- `__setitem__(self, <Индекс>, <Значение>)` — вызывается в случае присваивания нового значения элементу последовательности с заданным индексом (операция `<Экземпляр класса>[<Индекс>] = <Новое значение>`). Метод не должен возвращать результата. В случае задания индекса недопустимого типа и отсутствия такого индекса в последовательности следует возбуждать те же исключения, что и в случае метода `__getitem__()`;
- `__delitem__(self, <Ключ>)` — вызывается в случае удаления элемента последовательности с заданным индексом с помощью выражения `del <Экземпляр класса>[<Ключ>]`. Метод не должен возвращать результата. В случае задания индекса недопустимого типа и отсутствия такого индекса в последовательности следует возбуждать те же исключения, что и в случае метода `__getitem__()`;
- `__contains__(self, <Значение>)` — вызывается при проверке существования заданного значения в последовательности с применением операторов `in` и `not in`. Должен возвращать `True`, если такое значение есть, и `False` — в противном случае.

В классе-последовательности мы можем дополнительно реализовать функциональность итератора (см. *разд. 15.1*), переопределив специальные методы `__iter__()`, `__next__()` и `__len__()`. Чаще всего так и поступают.

Мы уже давно знаем, что строки в Python являются неизменяемыми. Давайте же напишем класс MutableString, представляющий строку, которую можно изменять теми же способами, что и список (листинг 15.3).

Листинг 15.3. Класс MutableString

```
class MutableString:
    def __init__(self, s):
        self.__s = list(s)

    # Реализуем функциональность итератора
    def __iter__(self):
        self.__i = 0
        return self
    def __next__(self):
        if self.__i > len(self.__s) - 1:
            raise StopIteration
        else:
            a = self.__s[self.__i]
            self.__i = self.__i + 1
            return a
    def __len__(self):
        return len(self.__s)

    def __str__(self):
        return "".join(self.__s)

    # Определяем вспомогательный метод, который будет проверять
    # корректность индекса
    def __iscorrectindex(self, i):
        if type(i) == int or type(i) == slice:
            if type(i) == int and i > self.__len__() - 1:
                raise IndexError
        else:
            raise TypeError

    # Реализуем функциональность контейнера-списка
    def __getitem__(self, i):
        self.__iscorrectindex(i)
        return self.__s[i]
    def __setitem__(self, i, v):
        self.__iscorrectindex(i)
        self.__s[i] = v
    def __delitem__(self, i):
        self.__iscorrectindex(i)
        del self.__s[i]
    def __contains__(self, v):
        return v in self.__s
```

Проверим свеженаписанный класс в действии:

```
>>> s = MutableString("Python")
>>> print(s[-1])
n
>>> s[0] = "J"
>>> del s[2:4]
>>> print(s)
Juon
```

Теперь проверим, как наш класс обрабатывает нештатные ситуации. Введем вот такой код, обращающийся к элементу с несуществующим индексом:

```
>>> s[9] = "u"
```

В ответ интерпретатор Python выдаст вполне ожидаемое сообщение об ошибке:

```
Traceback (most recent call last):
  File "C:/Users/dronov_va/Documents/Работа/Python Самое
  необходимое/Материалы/MutableString.py", line 50, in <module>
    s[9] = "u"
  File "C:/Users/dronov_va/Documents/Работа/Python Самое
  необходимое/Материалы/MutableString.py", line 29, in __setitem__
    raise IndexError
IndexError
```

15.2.2. Контейнеры-словари

Класс, реализующий функциональность перечисления, должен переопределять уже знакомые нам методы: `__getitem__()`, `__setitem__()`, `__delitem__()` и `__contains__()`. Разумеется, при этом следует сделать поправку на то, что вместо индексов здесь будут использоваться ключи произвольного типа (как правило, строкового).

Давайте исключительно для практики напишем класс `Version`, который будет хранить номер версии интерпретатора Python, разбитый на части: старшая цифра, младшая цифра и подрелиз. Причем доступ к частям номера версии мы будем получать по строковым ключам, как в обычном словаре Python (листинг 15.4). Ради простоты чтения кода функциональность итератора реализовывать не станем, а также заблокируем операцию удаления элемента словаря, возбудив в методе `__delitem__()` исключение `TypeError`.

Листинг 15.4. Класс `Version`

```
class Version:
    def __init__(self, major, minor, sub):
        self.__major = major                    # Старшая цифра
        self.__minor = minor                    # Младшая цифра
        self.__sub = sub                        # Подверсия
    def __str__(self):
        return str(self.__major) + "."  + str(self.__minor) + "." +
        str(self.__sub)

    # Реализуем функциональность словаря
    def __getitem__(self, k):
```

```
        if k == "major":
            return self.__major
        elif k == "minor":
            return self.__minor
        elif k == "sub":
            return self.__sub
        else:
            raise IndexError
    def __setitem__(self, k, v):
        if k == "major":
            self.__major = v
        elif k == "minor":
            self.__minor = v
        elif k == "sub":
            self.__sub = v
        else:
            raise IndexError
    def __delitem__(self, k):
        raise TypeError
    def __contains__(self, v):
        return v == "major" or v == "minor" or v == "sub"
```

Чтобы наш новый класс не бездельничал, дадим ему работу, введя такой код:

```
>>> v = Version(3, 4, 3)
>>> print(v["major"])
3
>>> v["sub"] = 4
>>> print(str(v))
3.4.4
```

Как видим, все работает как надо.

15.3. Перечисления

Перечисление — **это определенный самим программистом набор каких-либо именованных значений. Обычно они применяются для того, чтобы дать понятные имена каким-либо значениям, используемым в коде программы, — например, кодам ошибок, возвращаемым функциями Windows API.**

Инструменты для создания перечислений появились в Python 3.4. Это два класса, определенные в модуле `Enum`:

- `Enum` **— базовый класс для создания классов-перечислений, чьи элементы могут хранить значения произвольного типа:**

```
from enum import Enum
class Versions(Enum):
    V2_7 = "2.7"
    V3_4 = "3.4"
```

Здесь мы определили класс-перечисление `Versions`, имеющий два элемента: `V2_7` со значением `"2.7"` и `V3_4` со значением `"3.4"`. Отметим, что элементы перечислений представляют собой атрибуты объекта класса;

- `IntEnum` — базовый класс для создания перечислений, способных хранить лишь целочисленные значения:

```
from enum import IntEnum
class Colors(IntEnum):
    Red = 1
    Green = 2
    Blue = 3
```

Определяем перечисление `Colors` с тремя элементами, хранящими целые числа.

Имена элементов перечислений должны быть уникальны (что и неудивительно — ведь фактически это атрибуты объекта класса). Однако разные элементы все же могут хранить одинаковые значения:

```
from enum import Enum
class Versions(Enum):
    V2_7 = "2.7"
    V3_4 = "3.4"
    MostFresh = "3.4"
```

Чтобы объявить, что наше перечисление может хранить лишь уникальные значения, мы можем использовать декоратор `unique`, также определенный в модуле `enum`:

```
from enum import Enum, unique
@unique
class Versions(Enum):
    V2_7 = "2.7"
    V3_4 = "3.4"
```

Если мы попытаемся определить в классе, для которого был указан декоратор `unique`, элементы с одинаковыми значениями, то получим сообщение об ошибке.

Определив перечисление, можно использовать его элементы в вычислениях:

```
>>> e = Versions.V3_4
>>> e
<Versions.V3_4: '3.4'>
>>> e.value
'3.4'
>>> e == Versions.V2_7
False
```

Отметим, что для этого нам не придется создавать экземпляр класса. Это сделает сам Python, неявно создав экземпляр с тем же именем, что мы дали классу (вся необходимая для этого функциональность определена в базовых классах перечислений `Enum` и `IntEnum`).

Все классы перечислений принадлежат типу `EnumMeta` из модуля `enum`:

```
>>> type(Colors)
<class 'enum.EnumMeta'>
```

```
>>> from enum import EnumMeta
>>> type(Colors) == EnumMeta
True
```

Однако элементы перечислений уже являются экземплярами их классов:

```
>>> type(Colors.Red)
<enum 'Colors'>
>>> type(Colors.Red) == Colors
True
```

Над элементами перечислений можно производить следующие операции:

- обращаться к ним по их именам, использовав знакомую нам запись с точкой:

```
>>> Versions.V3_4
<Versions.V3_4: '3.4'>
>>> e = Versions.V3_4
>>> e
<Versions.V3_4: '3.4'>
```

- обращаться к ним в стиле словарей, использовав в качестве ключа имя элемента:

```
>>> Versions["V3_4"]
<Versions.V3_4: '3.4'>
```

- обращаться к ним по их значениям, указав их в круглых скобках после имени класса перечисления:

```
>>> Versions("3.4")
<Versions.V3_4: '3.4'>
```

- получать имена соответствующих им атрибутов класса и их значения, воспользовавшись свойствами `name` и `value` соответственно:

```
>>> Versions.V2_7.name, Versions.V2_7.value
('V2_7', '2.7')
```

- использовать в качестве итератора (необходимая для этого функциональность определена в базовых классах):

```
>>> list(Colors)
[<Colors.Red: 1>, <Colors.Green: 2>, <Colors.Blue: 3>]
>>> for c in Colors: print(c.value, end = " ")
1 2 3
```

- использовать в выражениях с применением операторов равенства, неравенства, `in` и `not in`:

```
>>> e = Versions.V3_4
>>> e == Versions.V3_4
True
>>> e != Versions.V2_7
True
>>> e in Versions
True
```

```
>>> e in Colors
False
```

Отметим, что элементы разных перечислений всегда не равны друг другу, даже если они и хранят одинаковые значения;

- использовать элементы перечислений — подклассов `IntEnum` в арифметических выражениях и в качестве индексов перечислений. В этом случае они будут автоматически преобразовываться в целые числа, соответствующие их значениям. Примеры:

```
>>> Colors.Red + 1                # Значение Colors.Red - 1
2
>>> Colors.Green != 3             # Значение Colors.Green - 2
True
>>> ["a", "b", "c"][Colors.Red]
'b'
```

Помимо элементов, классы перечислений могут включать атрибуты экземпляра класса и методы — как экземпляров, так и объектов класса. При этом методы экземпляра класса всегда вызываются у элемента перечисления (и, соответственно, первым параметром ему передается ссылка на экземпляр класса, представляющий элемент перечисления, у которого был вызван метод), а методы объекта класса — у самого класса перечисления. Для примера давайте рассмотрим код класса перечисления `VersionExtended` (листинг 15.5).

Листинг 15.5. Перечисление, включающее атрибуты и методы

```
from enum import Enum
class VersionExtended(Enum):
    V2_7 = "2.7"
    V3_4 = "3.4"

    # Методы экземпляра класса.
    # Вызываются у элемента перечисления
    def describe(self):
        return self.name, self.value
    def __str__(self):
        return str(__class__.__name__     "." + self.name + ": " +
        self.value

    # Метод объекта класса.
    # Вызывается у самого класса перечисления
    @classmethod
    def getmostfresh(cls):
        return cls.V3_4
```

В методе `__str__()` мы использовали встроенную переменную `__class__`, хранящую ссылку на объект текущего класса. Атрибут `__name__` этого объекта содержит имя класса в виде строки.

Осталось лишь проверить готовый класс в действии, для чего мы введем следующий код:

```
>>> d = VersionExtended.V2_7.describe()
>>> print(d[0] + ", " + d[1])
V2_7, 2.7
```

```
>>> print(VersionExtended.V2_7)
VersionExtended.V2_7: 2.7
>>> print(VersionExtended.getmostfresh())
VersionExtended.V3_4: 3.4
```

Осталось отметить одну важную деталь. На основе класса перечисления можно создавать подклассы только в том случае, если этот класс не содержит атрибутов объекта класса, т. е. собственно элементов перечисления. Если же класс перечисления содержит элементы, попытка определения его подкласса приведет к ошибке. Пример:

```
class ExtendedColors(Colors):
    pass

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    class ExtendedColors(Colors):
  File "C:\Python34\lib\enum.py", line 93, in __new__
    member_type, first_enum = metacls._get_mixins_(bases)
  File "C:\Python34\lib\enum.py", line 361, in _get_mixins_
    raise TypeError("Cannot extend enumerations")
TypeError: Cannot extend enumerations
```

Примечание

В составе стандартной библиотеки Python уже давно присутствует модуль `struct`, **позволяющий создавать нечто похожее на перечисления. Однако он не столь удобен в работе, как инструменты, предлагаемые модулем** `enum`.

ГЛАВА 16

Работа с файлами и каталогами

Очень часто нужно сохранить какие-либо данные. Для этого существуют два способа: запись 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.

16.1. Открытие файла

Прежде чем работать с файлом, необходимо создать объект файла с помощью функции `open()`. Функция имеет следующий формат:

```
open(<Путь к файлу>[, mode='r'][, buffering=-1][, encoding=None][,
     errors=None][, newline=None][, closefd=True])
```

В первом параметре указывается путь к файлу. Путь может быть абсолютным или относительным. При указании абсолютного пути в Windows следует учитывать, что в Python 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 из-за того, что слеши не удвоены, возникло присутствие сразу трех специальных символов: `\t`, `\n` и `\f` (отображается как `\x0c`). После преобразования этих 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 `OSError`:

```
>>> open("C:\temp\new\file.txt")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    open("C:\temp\new\file.txt")
OSError: [Errno 22] Invalid argument: '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 приводятся названия вложенных папок через слеш. Примеры:

```
>>> # 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. Примеры:

```
>>> # 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'
```

Как можно видеть, в абсолютном и относительном путях можно указать как прямые, так и обратные слеши. Все они будут автоматически преобразованы с учетом значения атрибута `sep` из модуля `os.path`. Значение этого атрибута зависит от используемой операционной системы. Выведем значение атрибута `sep` 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 это на примере, для чего 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 16.1.

Листинг 16.1. Содержимое файла C:\book\test.py

```
# -*- coding: utf-8 -*-
import os, sys
print("%-25s%s" % ("Файл:", os.path.abspath(__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.get_cwd()
```

Файл C:\book\folder1__init__.py создаем пустым. Как вы уже знаете, этот файл указывает интерпретатору Python, что данный каталог является пакетом с модулями. Содержимое файла C:\book\folder1\module1.py приведено в листинге 16.2.

Листинг 16.2. Содержимое файла C:\book\folder1\module1.py

```
# -*- coding: utf-8 -*-
import os, sys
def get_cwd():
    print("%-25s%s" % ("Файл:", os.path.abspath(__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`. Для примера изменим содержимое файла test.py (листинг 16.3).

Листинг 16.3. Пример использования функции chdir()

```
# -*- coding: utf-8 -*-
import os, sys
# Делаем каталог с исполняемым файлом текущим
os.chdir(os.path.dirname(os.path.abspath(__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 путь к исполняемому файлу вместе с названием файла. Атрибут `__file__` не всегда содержит полный путь к файлу. Например, если запуск осуществляется следующим образом:

```
C:\book>C:\Python34\python test.py,
```

то атрибут будет содержать только название файла без пути. Чтобы всегда получать полный путь к файлу, следует передать значение атрибута в функцию `abspath()` из модуля `os.path`. Далее мы 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 же 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 `mode` 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, возбуждается исключение `FileNotFoundError`;
- `r+` — чтение и запись. После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение `FileNotFoundError`;
- `w` — запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `w+` — чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
- `a` — запись. Если файл не существует, он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется;
- `a+` — чтение и запись. Если файл не существует, он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется;
- `x` — создание файла для записи. Если файл уже существует, возбуждается исключение `FileExistsError`;
- `x+` — создание файла для чтения и записи. Если файл уже существует, возбуждается исключение `FileExistsError`.

> ***Примечание***
>
> Поддержка последних двух режимов появилась в Python 3.3.

После указания режима может следовать модификатор:

- `b` — файл будет открыт в бинарном режиме. Файловые методы принимают и возвращают объекты типа `bytes`;
- `t` — файл будет открыт в текстовом режиме (значение по умолчанию в Windows). Файловые методы принимают и возвращают объекты типа `str`. В этом режиме будет автоматически выполняться обработка символа конца строки — так, в Windows при чтении вместо символов `\r\n` будет подставлен символ `\n`. Для 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
15
>>> f.close()                   # Закрываем файл
```

Поскольку мы указали режим w, то если файл не существует, он будет создан, а если существует, то будет перезаписан.

Теперь выведем содержимое файла в бинарном и текстовом режимах:

```
>>> # Бинарный режим (символ \r остается)
>>> with open(r"file.txt", "rb") as f:
        for line in f:
            print(repr(line))

b'String1\r\n'
b'String2'
>>> # Текстовый режим (символ \r удаляется)
>>> with open(r"file.txt", "r") as f:
        for line in f:
            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 или после вызова функции или метода `flush()`. В необязательном параметре `buffering` можно указать размер буфера. Если в качестве значения указан `0`, то данные будут сразу записываться в файл (значение допустимо только в бинарном режиме). Значение `1` используется при построчной записи в файл (значение допустимо только в текстовом режиме), другое положительное число задает примерный размер буфера, а отрицательное значение (или отсутствие значения) означает установку размера, применяемого в системе по умолчанию. По умолчанию текстовые файлы буферизуются построчно, а бинарные — частями, размер которых интерпретатор выбирает самостоятельно в диапазоне от 4096 до 8192 байтов.

При использовании текстового режима (задается по умолчанию) при чтении производится попытка преобразовать данные в кодировку Unicode, а при записи выполняется обратная операция — строка преобразуется в последовательность байтов в определенной кодировке. По умолчанию назначается кодировка, применяемая в системе. Если преобразование невозможно, то возбуждается исключение. Указать кодировку, которая будет использоваться при записи и чтении файла, позволяет параметр `encoding`. Для примера запишем данные в кодировке UTF-8:

```
>>> f = open(r"file.txt", "w", encoding="utf-8")
>>> f.write("Строка")  # Записываем строку в файл
6
>>> f.close()          # Закрываем файл
```

Для чтения этого файла следует явно указать кодировку при открытии файла:

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(line)

Строка
```

При работе с файлами в кодировках UTF-8, UTF-16 и UTF-32 следует учитывать, что в начале файла могут присутствовать служебные символы, называемые сокращенно BOM (Byte

Order Mark, метка порядка байтов). Для кодировки UTF-8 эти символы являются необязательными, и в предыдущем примере они не были добавлены в файл при записи. Чтобы символы были добавлены, в параметре `encoding` следует указать значение `utf-8-sig`. Запишем строку в файл в кодировке UTF-8 с BOM:

```
>>> f = open(r"file.txt", "w", encoding="utf-8-sig")
>>> f.write("Строка")  # Записываем строку в файл
6
>>> f.close()          # Закрываем файл
```

Теперь прочитаем файл с разными значениями в параметре `encoding`:

```
>>> with open(r"file.txt", "r", encoding="utf-8") as f:
        for line in f:
            print(repr(line))

'\ufeffСтрока'
>>> with open(r"file.txt", "r", encoding="utf-8-sig") as f:
        for line in f:
            print(repr(line))

'Строка'
```

В первом примере мы указали значение `utf-8`, поэтому маркер BOM был прочитан из файла вместе с данными. Во втором примере указано значение `utf-8-sig`, поэтому маркер BOM не попал в результат. Если неизвестно, есть ли маркер в файле, и необходимо получить данные без маркера, то следует всегда указывать значение `utf-8-sig` при чтении файла в кодировке UTF-8.

Для кодировок UTF-16 и UTF-32 маркер BOM является обязательным. При указании значений `utf-16` и `utf-32` в параметре `encoding` обработка маркера производится автоматически. При записи данных маркер автоматически вставляется в начало файла, а при чтении он не попадает в результат. Запишем строку в файл, а затем прочитаем ее из файла:

```
>>> with open(r"file.txt", "w", encoding="utf-16") as f:
        f.write("Строка")

6
>>> with open(r"file.txt", "r", encoding="utf-16") as f:
        for line in f:
            print(repr(line))

'Строка'
```

При использовании значений `utf-16-le`, `utf-16-be`, `utf-32-le` и `utf-32-be` маркер BOM необходимо самим добавить в начало файла, а при чтении удалить его.

В параметре `errors` можно указать уровень обработки ошибок. Возможные значения: `"strict"` (при ошибке возбуждается исключение `ValueError` — значение по умолчанию), `"replace"` (неизвестный символ заменяется символом вопроса или символом с кодом `\ufffd`), `"ignore"` (неизвестные символы игнорируются), `"xmlcharrefreplace"` (неизвестный символ заменяется последовательностью `&#xxxx;`) и `"backslashreplace"` (неизвестный символ заменяется последовательностью `\uxxxx`).

Параметр `newline` задает режим обработки символов конца строк. Поддерживаемые им значения таковы:

- `None` (значение по умолчанию) — выполняется стандартная обработка символов конца строки. Например, в Windows при чтении символы `\r\n` преобразуются в символ `\n`, а при записи производится обратное преобразование;
- `""` (пустая строка) — обработка символов конца строки не выполняется;
- `"<Специальный символ>"` — указанный специальный символ используется для обозначения конца строки, и никакая дополнительная обработка не выполняется. В качестве специального символа можно указать лишь `\r\n`, `\r` и `\n`.

16.2. Методы для работы с файлами

После открытия файла функция `open()` возвращает объект, с помощью которого производится дальнейшая работа с файлом. Тип объекта зависит от режима открытия файла и буферизации. Рассмотрим основные методы:

- `close()` — закрывает файл. Так как интерпретатор автоматически удаляет объект, когда на него отсутствуют ссылки, в небольших программах можно явно не закрывать файл. Тем не менее, явное закрытие файла является признаком хорошего стиля программирования. Кроме того, при наличии незакрытого файла генерируется предупреждающее сообщение: `"ResourceWarning: unclosed file"`.

  Язык Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует закрытие файла вне зависимости от того, произошло исключение внутри блока кода или нет. Пример:

```
with open(r"file.txt", "w", encoding="cp1251") as f:
    f.write("Строка") # Записываем строку в файл
# Здесь файл уже закрыт автоматически
```

- `write(<Данные>)` — записывает строку или последовательность байтов в файл. Если в качестве параметра указана строка, то файл должен быть открыт в текстовом режиме. Для записи последовательности байтов необходимо открыть файл в бинарном режиме. Помните, что нельзя записывать строку в бинарном режиме и последовательность байтов в текстовом режиме. Метод возвращает количество записанных символов или байтов. Пример записи в файл:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.write("Строка1\nСтрока2") # Записываем строку в файл
15
>>> f.close()                   # Закрываем файл
>>> # Бинарный режим
>>> f = open(r"file.txt", "wb")
>>> f.write(bytes("Строка1\nСтрока2", "cp1251"))
15
>>> f.write(bytearray("\nСтрока3", "cp1251"))
8
>>> f.close()
```

- `writelines(<Последовательность>)` — **записывает последовательность в файл. Если все элементы последовательности являются строками, то файл должен быть открыт в текстовом режиме. Если все элементы являются последовательностями байтов, то файл должен быть открыт в бинарном режиме. Пример записи элементов списка:**

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.writelines(["Строка1\n", "Строка2"])
>>> f.close()
>>> # Бинарный режим
>>> f = open(r"file.txt", "wb")
>>> arr = [bytes("Строка1\n", "cp1251"), bytes("Строка2", "cp1251")]
>>> f.writelines(arr)
>>> f.close()
```

- `writable()` — **возвращает** `True`, **если файл поддерживает запись, и** `False` — **в противном случае:**

```
>>> f = open(r"file.txt", "r")          # Открываем файл для чтения
>>> f.writable()
False
>>> f = open(r"file.txt", "w")          # Открываем файл для записи
>>> f.writable()
True
```

- `read([<Количество>])` — **считывает данные из файла. Если файл открыт в текстовом режиме, то возвращается строка, а если в бинарном — последовательность байтов. Если параметр не указан, возвращается содержимое файла от текущей позиции указателя до конца файла:**

```
>>> # Текстовый режим
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.read()

'Строка1\nСтрока2'
>>> # Бинарный режим
>>> with open(r"file.txt", "rb") as f:
        f.read()

b'\xd1\xf2\xf0\xee\xea\xe01\n\xd1\xf2\xf0\xee\xea\xe02'
```

Если в качестве параметра указать число, то за каждый вызов будет возвращаться указанное количество символов или байтов. Когда достигается конец файла, метод возвращает пустую строку. Пример:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.read(8)                  # Считываем 8 символов
'Строка1\n'
>>> f.read(8)                  # Считываем 8 символов
'Строка2'
>>> f.read(8)                  # Достигнут конец файла
''
>>> f.close()
```

- `readline([<Количество>])` — считывает из файла одну строку при каждом вызове. Если файл открыт в текстовом режиме, то возвращается строка, а если в бинарном — последовательность байтов. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка — если она не завершается символом перевода строки, то таковой добавлен не будет. При достижении конца файла возвращается пустая строка. Пример:

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.readline(), f.readline()
('Строка1\n', 'Строка2')
>>> f.readline()              # Достигнут конец файла
''
>>> f.close()
>>> # Бинарный режим
>>> f = open(r"file.txt", "rb")
>>> f.readline(), f.readline()
(b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02')
>>> f.readline()              # Достигнут конец файла
b''
>>> 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 символов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество символов, а если количество символов в строке больше, то возвращается указанное количество символов. Пример:

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.readline(2), f.readline(2)
('Ст', 'ро')
>>> f.readline(100) # Возвращается одна строка, а не 100 символов
'ка1\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. Если файл открыт в текстовом режиме, то возвращается список строк, а если в бинарном — список объектов типа `bytes`. Пример:

```
>>> # Текстовый режим
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.readlines()
['Строка1\n', 'Строка2']
>>> # Бинарный режим
>>> with open(r"file.txt", "rb") as f:
        f.readlines()

[b'\xd1\xf2\xf0\xee\xea\xe01\n', b'\xd1\xf2\xf0\xee\xea\xe02']
```

- `__next__()` — **считывает одну строку при каждом вызове. Если файл открыт в текстовом режиме, возвращается строка, а если в бинарном — последовательность байтов. При достижении конца файла возбуждается исключение** `StopIteration`. **Пример:**

```
>>> # Текстовый режим
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.__next__(), f.__next__()
('Строка1\n', 'Строка2')
>>> f.__next__() # Достигнут конец файла
Traceback (most recent call last):
  File "<pyshell#26>", 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__()`. **Для примера выведем все строки, предварительно удалив символ перевода строки:**

```
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> for line in f: print(line.rstrip("\n"), end=" ")

Строка1 Строка2
>>> f.close()
```

- `flush()` — **принудительно 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", encoding="cp1251")
>>> f.fileno()                    # Дескриптор файла
3
>>> f.close()
```

- `truncate([<Количество>])` — **обрезает файл до указанного количества символов (если задан текстовый режим) или байтов (в случае бинарного режима). Метод возвращает новый размер файла. Пример:**

```
>>> f = open(r"file.txt", "r+", encoding="cp1251")
>>> f.read()
'Строка1\nСтрока2'
>>> f.truncate(5)
5
>>> f.close()
>>> with open(r"file.txt", "r", encoding="cp1251") as f:
        f.read()

'Строк'
```

- `tell()` — возвращает позицию указателя относительно начала файла в виде 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:

```
>>> with open(r"file.txt", "w", encoding="cp1251") as f:
        f.write("String1\nString2")

15
>>> f = open(r"file.txt", "r", encoding="cp1251")
>>> f.tell()       # Указатель расположен в начале файла
0
>>> f.readline()   # Перемещаем указатель
'String1\n'
>>> f.tell()       # Возвращает 9 (8 + '\r'), а не 8 !!!
9
>>> f.close()
```

Чтобы избежать этого несоответствия, следует открывать файл в бинарном режиме, а не в текстовом:

```
>>> f = open(r"file.txt", "rb")
>>> f.readline()   # Перемещаем указатель
b'String1\r\n'
>>> f.tell()       # Теперь значение соответствует
9
>>> 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 `io` или соответствующие им значения:
  - `io.SEEK_SET` или `0` — начало файла (значение по умолчанию);
  - `io.SEEK_CUR` или `1` — текущая позиция указателя. Положительное значение смещения вызывает перемещение к концу файла, отрицательное — к его началу;
  - `io.SEEK_END` или `2` — конец файла.

Выведем значения этих атрибутов:

```
>>> import io
>>> io.SEEK_SET, io.SEEK_CUR, io.SEEK_END
(0, 1, 2)
```

Пример использования метода `seek()`:

```
>>> import io
>>> f = open(r"file.txt", "rb")
>>> f.seek(9, io.SEEK_CUR) # 9 байтов от указателя
9
>>> f.tell()
9
>>> f.seek(0, io.SEEK_SET) # Перемещаем указатель в начало
0
```

```
>>> f.tell()
0
>>> f.seek(-9, io.SEEK_END) # -9 байтов от конца файла
7
>>> f.tell()
7
>>> f.close()
```

- `seekable()` — возвращает `True`, если указатель файла можно сдвинуть в другую позицию, и `False` — в противном случае:

```
>>> f = open(r"C:\temp\new\file.txt", "r")
>>> f.seekable()
True
```

Помимо методов, объекты файлов поддерживают несколько атрибутов:

- `name` — имя 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', 'rb+', False)
>>> f.close()
>>> f.closed
True
```

- `encoding` — название кодировки, которая будет использоваться для преобразования строк перед записью в файл или при чтении. Атрибут доступен только в текстовом режиме. Обратите также внимание на то, что изменить значение атрибута нельзя, поскольку он доступен только для чтения. Пример:

```
>>> f = open(r"file.txt", "a", encoding="cp1251")
>>> f.encoding
'cp1251'
>>> f.close()
```

Стандартный вывод `stdout` также является файловым объектом. Атрибут `encoding` этого объекта всегда содержит кодировку устройства вывода, поэтому строка преобразуется в последовательность байтов 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'
```

- `buffer` — позволяет получить доступ к буферу. Атрибут доступен только в текстовом режиме. С помощью этого объекта можно записать последовательность байтов в текстовый поток.

Пример:

```
>>> f = open(r"file.txt", "w", encoding="cp1251")
>>> f.buffer.write(bytes("Строка", "cp1251"))
6
>>> f.close()
```

16.3. 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 и более поздних версий атрибут имеет значение `"nt"`:

```
>>> import os
>>> os.name                  # Значение в ОС Windows 8
'nt'
```

Для доступа к файлам предназначены следующие функции из модуля `os`:

- `open(<Путь к файлу>, <Режим>[, mode=0o777])` — открывает файл и возвращает целочисленный дескриптор, с помощью которого производится дальнейшая работа с файлом. Если файл открыть не удалось, возбуждается исключение `OSError` или одно из исключений, являющихся подклассами класса `OSError` (мы поговорим о них в конце этой главы). 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 и если не указан флаг `os.O_EXCL`;
  - `os.O_EXCL` — при использовании совместно с `os.O_CREAT` указывает, что создаваемый файл изначально не должен существовать, в противном случае будет сгенерировано исключение `FileExistsError`;
  - `os.O_TEMPORARY` — при использовании совместно с `os.O_CREAT` указывает, что создается временный файл, который будет автоматически удален сразу после закрытия;
  - `os.O_SHORT_LIVED` — то же самое, что `os.O_TEMPORARY`, но созданный файл по возможности будет храниться лишь в оперативной памяти, а не на диске;
  - `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, b"String1\n") # Записываем данные
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, b"String2\n") # Записываем данные
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
b'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
b'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)
(b'Strin', b'g1\nS', b'tring', b'2\n')
>>> os.read(f, 5)            # Достигнут конец файла
b''
>>> os.close(f)              # Закрываем файл
```

- `write(<Дескриптор>, <Последовательность байтов>)` — **записывает последовательность байтов 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
18
```

```
>>> os.lseek(f, 0, os.SEEK_SET) # Перемещение в начало файла
0
>>> os.lseek(f, 9, os.SEEK_CUR) # Относительно указателя
9
>>> os.lseek(f, 0, os.SEEK_CUR) # Текущее положение указателя
9
>>> 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()
```

16.4. Классы *StringIO* и *BytesIO*

Класс `StringIO` **из модуля** `io` **позволяет работать со строкой как с файловым объектом. Все операции с этим файловым объектом (будем называть его далее «файл») производятся в оперативной памяти. Формат конструктора класса:**

```
StringIO([<Начальное значение>][, newline=None])
```

Если первый 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 io                # Подключаем модуль
>>> f = io.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 = io.StringIO("String1\n")
>>> f.tell()         # Позиция указателя
0
>>> f.seek(0, 2)     # Перемещаем указатель в конец «файла»
8
>>> f.tell()         # Позиция указателя
8
>>> f.seek(0)        # Перемещаем указатель в начало «файла»
0
>>> f.tell()         # Позиция указателя
0
>>> f.close()        # Закрываем файл
```

- `write(<Строка>)` — **записывает строку в «файл»:**

```
>>> f = io.StringIO("String1\n")
>>> f.seek(0, 2)          # Перемещаем указатель в конец «файла»
8
>>> f.write("String2\n") # Записываем строку в «файл»
8
>>> f.getvalue()          # Получаем содержимое «файла»
'String1\nString2\n'
>>> f.close()             # Закрываем «файл»
```

- `writelines(<Последовательность>)` — **записывает последовательность в «файл»:**

```
>>> f = io.StringIO()
>>> f.writelines(["String1\n", "String2\n"])
>>> f.getvalue()          # Получаем содержимое «файла»
'String1\nString2\n'
>>> f.close()             # Закрываем «файл»
```

- `read([<Количество символов>])` — **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 символов. Когда достигается конец «файла», метод возвращает пустую строку. Пример:**

```
>>> f = io.StringIO("String1\nString2\n")
>>> f.read()
'String1\nString2\n'
>>> f.seek(0) # Перемещаем указатель в начало «файла»
0
>>> 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 символов>])` — 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 = io.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 символов. Иными словами, если количество символов в строке меньше значения параметра, то будет считана одна строка, а не указанное количество символов. Если количество символов в строке больше, то возвращается указанное количество символов. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3\n")
>>> f.readline(5), f.readline(5)
('Strin', 'g1\n')
>>> f.readline(100) # Возвращается одна строка, а не 100 символов
'String2\n'
>>> f.close()       # Закрываем «файл»
```

- `readlines([<Примерное количество символов>])` — 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 таковой добавлен не будет. Пример:

```
>>> f = io.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 символов плюс фрагмент до символа конца строки `\n`. Затем эта строка разбивается и добавляется построчно в список. Пример:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.readlines(14)
['String1\n', 'String2\n']
>>> f.seek(0) # Перемещаем указатель в начало «файла»
0
>>> 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 = io.StringIO("String1\nString2")
>>> f.__next__(), f.__next__()
('String1\n', 'String2')
```

```
>>> f.__next__()
... 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 = io.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 символов>])` — **обрезает «файл» до указанного количества символов. Пример:**

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.truncate(15)      # Обрезаем «файл»
15
>>> f.getvalue()        # Получаем содержимое «файла»
'String1\nString2'
>>> f.close()           # Закрываем «файл»
```

Если параметр не указан, то «файл» обрезается до текущей позиции указателя:

```
>>> f = io.StringIO("String1\nString2\nString3")
>>> f.seek(15)          # Перемещаем указатель
15
>>> f.truncate()        # Обрезаем «файл» до указателя
15
>>> f.getvalue()        # Получаем содержимое «файла»
'String1\nString2'
>>> f.close()           # Закрываем «файл»
```

Описанные ранее методы `writable()` **и** `seekable()`, **вызванные у объекта класса** `StringIO`, **всегда возвращают** `True`.

Класс `StringIO` **работает только со строками. Чтобы выполнять аналогичные операции с «файлами», представляющими собой последовательности байтов, следует использовать класс** `BytesIO` **из модуля** `io`. **Формат конструктора класса:**

```
BytesIO([<Начальное значение>])
```

Класс `BytesIO` **поддерживает такие же методы, что и класс** `StringIO`, **но в качестве значений методы принимают и возвращают последовательности байтов, а не строки. Рассмотрим основные операции на примере:**

```
>>> import io              # Подключаем модуль
>>> f = io.BytesIO(b"String1\n")
>>> f.seek(0, 2)           # Перемещаем указатель в конец «файла»
8
```

```
>>> f.write(b"String2\n") # Пишем в «файл»
8
>>> f.getvalue()         # Получаем содержимое «файла»
b'String1\nString2\n'
>>> f.seek(0)            # Перемещаем указатель в начало «файла»
0
>>> f.read()             # Считываем данные
b'String1\nString2\n'
>>> f.close()            # Закрываем «файл»
```

Класс `BytesIO` поддерживает также метод `getbuffer()`, который возвращает ссылку на объект `memoryview`. С помощью этого объекта можно получать и изменять данные по индексу или срезу, преобразовывать данные в список целых чисел (с помощью метода `tolist()`) или в последовательность байтов (с помощью метода `tobytes()`). Пример:

```
>>> f = io.BytesIO(b"Python")
>>> buf = f.getbuffer()
>>> buf[0]               # Получаем значение по индексу
b'P'
>>> buf[0] = b"J"        # Изменяем значение по индексу
>>> f.getvalue()         # Получаем содержимое
b'Jython'
>>> buf.tolist()         # Преобразуем в список чисел
[74, 121, 116, 104, 111, 110]
>>> buf.tobytes()        # Преобразуем в тип bytes
b'Jython'
>>> f.close()            # Закрываем «файл»
```

16.5. Права доступа к файлам и каталогам

В операционных системах семейства UNIX каждому объекту (файлу или каталогу) назначаются права доступа, предоставляемые той или иной разновидности пользователей: владельцу, группе и прочим. 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 в нем поиск файлов.

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 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 — также 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`).

Права доступа могут обозначаться и 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. 16.1).

Таблица 16.1. *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`

Теперь понятно, что, согласно данным этой таблицы, права доступа `rw-r--r--` можно записать так: 110 100 100, что и переводится в число 644. 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 комбинацию символов 0o (это соответствует восьмеричной записи числа):

```
>>> os.chmod(r"file.txt", 0o777) # Полный доступ к файлу
```

Вместо числа можно указать комбинацию констант из модуля stat. За дополнительной информацией обращайтесь к документации по модулю.

16.6. 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 OSError или одно из исключений, являющихся подклассом этого класса. 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 ...
FileNotFoundError: [Errno 2] No such file or directory:
'C:\\book2\\file2.txt'
```

  Исключение FileNotFoundError является подклассом класса OSError и возбуждается, если указанный файл не найден. Более подробно классы исключений, возбуждаемых при файловых операциях, мы рассмотрим в конце этой главы.

  Начиная с Python 3.3, функция copyfile() в качестве результата возвращает путь файла, куда были скопированы данные;

- copy(<Копируемый файл>, <Куда копируем>) — позволяет скопировать файл вместе с правами доступа. Если файл существует, то он будет перезаписан. Если файл не удалось скопировать, возбуждается исключение OSError или одно из исключений, являющихся подклассом этого класса. Пример:

```
>>> shutil.copy(r"file.txt", r"file3.txt")
```

  Начиная с Python 3.3, функция copy() в качестве результата возвращает путь скопированного файла;

- 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 OSError или одно из исключений, являющихся подклассом этого класса. Пример:

```
>>> shutil.copy2(r"file.txt", r"file4.txt")
```

Начиная с Python 3.3, функция `copy2()` в качестве результата возвращает путь скопированного файла;

- `move(<Путь к файлу>, <Куда перемещаем>)` — **перемещает файл в указанное место с удалением исходного файла. Если файл существует, то он будет перезаписан. Если файл не удалось переместить, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса. Пример перемещения файла file4.txt в каталог C:\book\test:**

```
>>> shutil.move(r"file4.txt", r"C:\book\test")
```

Начиная с Python 3.3, функция `move()` в качестве результата возвращает путь перемещенного файла.

Для переименования и удаления файлов предназначены следующие функции из модуля `os`:

- `rename(<Старое имя>, <Новое имя>)` — **переименовывает файл. Если файл не удалось переименовать, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса. Пример переименования файла с обработкой исключений:**

```
import os # Подключаем модуль
try:
    os.rename(r"file3.txt", "file4.txt")
except 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 не удалось удалить, возбуждается исключение `OSError` или одно из исключений, являющихся подклассом этого класса. 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. Опишем 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)
```

Начиная с Python 3.3, в качестве параметра можно передать целочисленный дескриптор открытого файла, возвращенный функцией `open()` из того же модуля `os`;

- `getsize(<Путь к файлу>)` — **возвращает размер файла в байтах. Если файл не существует, возбуждается исключение `OSError`:**

```
>>> os.path.getsize(r"file.txt")  # Файл существует
18
>>> os.path.getsize(r"file2.txt") # Файл не существует
... Фрагмент опущен ...
OSError: [Error 2] Не удается найти указанный файл: '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, возбуждается исключение** `OSError`. **Пример:**

```
>>> import time     # Подключаем модуль time
>>> t = os.path.getatime(r"file.txt")
>>> t
1304111982.96875
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'30.04.2011 01:19:42'
```

- `getctime(<Путь к файлу>)` — **позволяет узнать дату создания файла. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, возбуждается исключение** `OSError`. **Пример:**

```
>>> t = os.path.getctime(r"file.txt")
>>> t
1304028509.015625
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 02:08:29'
```

- `getmtime(<Путь к файлу>)` — **возвращает время последнего изменения файла. В качестве значения функция возвращает количество секунд, прошедших с начала эпохи. Если файл не существует, возбуждается исключение** `OSError`. **Пример:**

```
>>> t = os.path.getmtime(r"file.txt")
>>> t
1304044731.265625
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 06:38:51'
```

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 16.4.**

Листинг 16.4. Пример использования функции stat()

```
>>> import os, time
>>> s = os.stat(r"file.txt")
>>> s
nt.stat_result(st_mode=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304111982,
st_mtime=1304044731, st_ctime=1304028509)
>>> s.st_size       # Размер файла
18
>>> t = s.st_atime # Время последнего доступа к файлу
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'30.04.2011 01:19:42'
>>> t = s.st_ctime # Время создания файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 02:08:29'
```

```
>>> t = s.st_mtime # Время последнего изменения файла
>>> time.strftime("%d.%m.%Y %H:%M:%S", time.localtime(t))
'29.04.2011 06:38:51'
```

Обновить время последнего доступа и время изменения файла позволяет функция `utime()` **из модуля** `os`. **Функция имеет два варианта формата:**

```
utime(<Путь к файлу>, None)
utime(<Путь к файлу>, (<Последний доступ>, <Изменение файла>))
```

Начиная с Python 3.3, в качестве первого параметра можно указывать не только строковый путь, но и целочисленный дескриптор открытого файла, возвращенный функцией `open()` **из модуля** `os`. **Если в качестве второго параметра указано значение** `None`, **то время доступа и изменения файла будет текущим. Во втором варианте формата функции** `utime()` **указывается кортеж из новых значений в виде количества секунд, прошедших с начала эпохи. Если файл не существует, возбуждается исключение** `OSError`. **Пример использования функции** `utime()` **приведен в листинге 16.5.**

Листинг 16.5. Пример использования функции `utime()`

```
>>> import os, time
>>> os.stat(r"file.txt")          # Первоначальные значения
nt.stat_result(st_mode=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304111982,
st_mtime=1304044731, st_ctime=1304028509)
>>> 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=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304112906,
st_mtime=1304112906, st_ctime=1304028509)
>>> os.utime(r"file.txt", None)   # Текущее время
>>> os.stat(r"file.txt")
nt.stat_result(st_mode=33060, st_ino=2251799813878096, st_dev=0,
st_nlink=1, st_uid=0, st_gid=0, st_size=18, st_atime=1304113557,
st_mtime=1304113557, st_ctime=1304028509)
```

16.7. 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. Примеры:**

```
>>> 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'
```

Как уже отмечалось ранее, в относительном пути можно указать как прямые, так и обратные слеши. Все они будут автоматически преобразованы с учетом значения атрибута `sep` из модуля `os.path`. Значение этого атрибута зависит от используемой операционной системы. Выведем значение атрибута `sep` 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 (экранировать) 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 данном случае лучше использовать обычные строки:

```
>>> "C:\\temp\\new\\"             # Правильно
'C:\\temp\\new\\'
>>> r"C:\temp\new\\"[:-1]         # Можно и удалить слеш
'C:\\temp\\new\\'
```

- `isabs(<Путь>)` — возвращает `True`, если путь является абсолютным, и `False` — в противном случае:

```
>>> os.path.isabs(r"C:\book\file.txt")
True
>>> os.path.isabs("file.txt")
False
```

- `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 папке, где хранится файл:**

```
>>> 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 папке, где хранится файл, 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, при необходимости вставляя между ними разделители:**

```
>>> 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'
```

16.8. 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` **поддерживают метод** `write()`, **предназначенный для вывода сообщений, а поток** `stdin` **— метод** `readline()`, **служащий для получения вво-**

димых пользователем данных. Если этим потокам присвоить ссылку на объект, поддерживающий файловые методы, то можно перенаправить стандартные потоки в соответствующий файл. Для примера так и сделаем (листинг 16.6).

Листинг 16.6. 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.

Функция `print()` напрямую поддерживает перенаправление вывода. Для этого используется параметр `file`, который по умолчанию ссылается на стандартный поток вывода. Например, записать строку в файл можно так:

```
>>> f = open(r"file.txt", "a")
>>> print("Пишем строку в файл", file=f)
>>> f.close()
```

Параметр `flush`, поддержка которого появилась в Python 3.3, позволяет указать, когда следует выполнять непосредственное сохранение данных из промежуточного буфера в файл. Если его значение равно `False` (это, кстати, значение по умолчанию), сохранение будет выполнено лишь после закрытия файла или после вызова метода `flush()`. Чтобы указать интерпретатору Python выполнять сохранение после каждого вызова функции `print()`, следует присвоить этому параметру значение `True`. Пример:

```
>>> f = open(r"file.txt", "a")
>>> print("Пишем строку в файл", file = f, flush = True)
>>> print("Пишем другую строку в файл", file = f, flush = True)
>>> f.close()
```

Стандартный ввод `stdin` также можно перенаправить. В этом случае функция `input()` будет читать одну строку из файла при каждом вызове. При достижении конца файла возбуждается исключение `EOFError`. Для примера выведем содержимое файла с помощью перенаправления потока ввода (листинг 16.7).

Листинг 16.7. Перенаправление потока ввода

```
# -*- coding: utf-8 -*-
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 = 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
input()
```

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. Примеры:

```
>>> 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. Для примера создадим в папке C:\book файл tests.py с кодом, приведенным в листинге 16.8.

Листинг 16.8. Содержимое файла tests.py

```
# -*- coding: utf-8 -*-
while True:
    try:
        line = input()
        print(line)
    except EOFError:
        break
```

Запускаем командную строку и переходим в папку со скриптом, выполнив команду: `cd C:\book`. Теперь выведем содержимое созданного ранее текстового файла file.txt (его содержимое может быть любым), выполнив команду:

```
C:\Python34\python.exe tests.py < file.txt
```

Перенаправить стандартный вывод в файл можно аналогичным образом. Только в этом случае символ < необходимо заменить символом >. Изменим файл tests.py следующим образом:

```
# -*- coding: utf-8 -*-
print("String")              # Эта строка будет записана в файл
```

Теперь перенаправим вывод в файл file.txt, выполнив команду:

```
C:\Python34\python.exe tests.py > file.txt
```

В этом режиме файл file.txt будет перезаписан. Если необходимо добавить результат в конец файла, следует использовать символы >>. Пример дозаписи в файл:

```
C:\Python34\python.exe tests.py >> file.txt
```

С помощью стандартного вывода stdout можно создать индикатор выполнения процесса непосредственно 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 16.9).

Листинг 16.9. Индикатор выполнения процесса

```
# -*- coding: utf-8 -*-
import sys, time
for i in range(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")
input()
```

Сохраняем код в файл и запускаем его с помощью двойного щелчка. В окне консоли записи будут заменять друг друга на одной строке каждую секунду. Так как данные перед выводом будут помещаться в буфер, мы сбрасываем их на диск явным образом с помощью метода flush().

16.9. 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>][, fix_imports=True]) — производит сериализацию объекта и записывает данные в указанный файл. В параметре <Файл> указывается файловый объект, открытый на запись в бинарном режиме. Пример сохранения объекта в файл:

  ```
  >>> import pickle
  >>> f = open(r"file.txt", "wb")
  >>> obj = ["Строка", (2, 3)]
  >>> pickle.dump(obj, f)
  >>> f.close()
  ```

- load() — читает данные из файла и преобразует их в объект. В параметре <Файл> указывается файловый объект, открытый на чтение в бинарном режиме. Формат функции:

  ```
  load(<Файл>[, fix_imports=True][, encoding="ASCII"]
             [, errors="strict"])
  ```

  Пример восстановления объекта из файла:

  ```
  >>> f = open(r"file.txt", "rb")
  >>> obj = pickle.load(f)
  ```

```
>>> obj
['Строка', (2, 3)]
>>> f.close()
```

В один файл можно сохранить сразу несколько объектов, последовательно вызывая функцию `dump()`. **Пример сохранения нескольких объектов приведен в листинге 16.10.**

Листинг 16.10. 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 16.11).**

Листинг 16.11. 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
(['Строка', (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>][, fix_imports=True])
```

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>[, fix_imports=True][, encoding="ASCII"]
                [, errors="strict"])
```

Пример восстановления объекта из файла:

```
>>> f = open(r"file.txt", "rb")
>>> obj = pickle.Unpickler(f).load()
>>> obj
['Строка', (2, 3)]
>>> f.close()
```

Модуль pickle позволяет также преобразовать объект в последовательность байтов и восстановить объект из таковой. Для этого предназначены две функции:

- dumps(<Объект>[, <Протокол>][, fix_imports=True]) — производит сериализацию объекта и возвращает последовательность байтов специального формата. Формат зависит от указанного протокола — числа от 0 до 4 в порядке от более старых к более новым и совершенным. По умолчанию используется протокол 4, поддержка которого появилась в Python 3.4. Пример преобразования списка и кортежа:

```
>>> obj1 = [1, 2, 3, 4, 5]       # Список
>>> obj2 = (6, 7, 8, 9, 10)      # Кортеж
>>> pickle.dumps(obj1)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
>>> pickle.dumps(obj2)
b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.'
```

- loads(<Последовательность байтов>[, fix_imports=True][, encoding="ASCII"][, errors="strict"]) — преобразует последовательность байтов специального формата в объект. Пример восстановления списка и кортежа:

```
>>> pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.')
[1, 2, 3, 4, 5]
>>> pickle.loads(b'\x80\x03(K\x06K\x07K\x08K\tK\ntq\x00.')
(6, 7, 8, 9, 10)
```

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 dbm. Все эти действия модуль shelve производит самостоятельно.

Открыть файл с набором объектов поможет 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. Для примера создадим файл и сохраним в нем список и кортеж:

```
>>> import shelve                    # Подключаем модуль
>>> db = shelve.open("db1")          # 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()` — **возвращает объект с ключами;**
- `values()` — **возвращает объект со значениями;**
- `items()` — **возвращает объект-итератор, который на каждой итерации генерирует кортеж, содержащий ключ и значение. Пример:**

```
>>> db = shelve.open("db1")
>>> db.keys(), db.values()
(KeysView(<shelve.DbfilenameShelf object at 0x00FE81B0>),
 ValuesView(<shelve.DbfilenameShelf object at 0x00FE81B0>))
>>> list(db.keys()), list(db.values())
(['obj1', 'obj2'], [[1, 2, 3, 4, 5], (6, 7, 8, 9, 10)])
>>> db.items()
ItemsView(<shelve.DbfilenameShelf object at 0x00FE81B0>)
>>> list(db.items())
[('obj1', [1, 2, 3, 4, 5]), ('obj2', (6, 7, 8, 9, 10))]
>>> 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, создается новый элемент со значением, указанным во втором параметре, и в качестве результата возвращается 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, 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, а также операторами** `in` **и** `not in` **для проверки существования или несуществования ключа.**

Пример:

```
>>> db = shelve.open("db1")
>>> len(db)              # Количество элементов
2
>>> "obj1" in db
True
>>> del db["obj1"]       # Удаление элемента
>>> "obj1" in db
False
>>> "obj1" not in db
True
>>> db.close()
```

16.10. 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 восьмеричным числом (значение по умолчанию `0o777`). 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, возбуждается исключение — подкласс класса `OSError`. 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. На каждой итерации через этот объект доступен кортеж из трех элементов: текущего каталога, списка каталогов и списка файлов, находящихся в нем. Если произвести изменения в списке каталогов во время выполнения, 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 параметре `topdown` указано значение `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 имеющиеся в нем 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\\")
```

- `normcase(<Каталог>)` — преобразует заданный путь к каталогу к виду, подходящему для использования в текущей операционной системе. В Windows преобразует все прямые слеши в обратные. Также во всех системах приводит все буквы пути к нижнему регистру. Пример:

```
>>> from os.path import normcase
>>> normcase(r"c:/BoOk/fIlE.TxT")
'c:\\book\\file.txt'
```

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 дефис.

В качестве значения функция возвращает список путей к объектам, совпадающим с шаблоном. Пример использования функции `glob()` приведен в листинге 16.12.

Листинг 16.12. 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.

16.11. Исключения, возбуждаемые файловыми операциями

В этой главе неоднократно говорилось, что функции и методы, осуществляющие файловые операции, при возникновении нештатных ситуаций возбуждают исключение класса `OSError` или одно из исключений, являющееся его подклассом. Настало время познакомиться с ними.

Исключений-подклассов класса `OSError` довольно много. Вот те из них, что затрагивают именно операции с файлами и папками:

- `BlockingIOError` — не удалось заблокировать объект (файл или поток ввода/вывода);
- `ConnectionError` — ошибка сетевого соединения. Может возникнуть при открытии файла по сети. Является базовым классом для ряда других исключений более высокого уровня, описанных в документации по Python;
- `FileExistsError` — файл или папка с заданным именем уже существуют;
- `FileNotFoundError` — файл или папка с заданным именем не обнаружены;
- `InterruptedError` — файловая операция неожиданно прервана по какой-либо причине;
- `IsADirectoryError` — вместо пути к файлу указан путь к папке;
- `NotADirectoryError` — вместо пути к папке указан путь к файлу;
- `PermissionError` — отсутствуют права на доступ к указанному файлу или папке;
- `TimeoutError` — истекло время, отведенное системой на выполнение операции.

Пример кода, обрабатывающего некоторые из указанных исключений, приведен в листинге 16.13.

Листинг 16.13. Обработка исключений, возбуждаемых при файловых операциях

```
. . .
try
    open("C:\temp\new\file.txt")
```

```
except FileNotFoundError:
    print("Файл отсутствует")
except IsADirectoryError:
    print("Это не файл, а папка")
except PermissionError:
    print("Отсутствуют права на доступ к файлу"
except OSError:
    print("Неустановленная ошибка открытия файла")
. . .
```

Примечание

В версиях Python, предшествующих 3.3, поддерживалась другая иерархия исключений. В частности, поддерживались классы исключений `IOError` (возникали при файловых операциях в любых операционных системах) и `WindowsError` (возникали лишь в системе Windows). Начиная с Python 3.3, эти классы являются синонимами класса `OSError`. За подробностями обращайтесь к документации по языку Python.

ГЛАВА 17

Основы SQLite

В предыдущей главе мы освоили 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 модуля `dbm` для записи получившейся строки по ключу в файл. Если необходимо сохранять в файл просто строки, то можно сразу воспользоваться модулем `dbm`. Однако если объем сохраняемых данных велик и требуется удобный доступ к ним, то вместо этого модуля лучше использовать базы данных.

В состав стандартной библиотеки Python входит модуль `sqlite3`, позволяющий работать с базой данных SQLite. И для этого даже нет необходимости устанавливать сервер, ожидающий запросы на каком-либо порту, т. к. SQLite работает с файлом базы данных напрямую. Все что нужно для работы с SQLite, — это библиотека sqlite3.dll (расположена в папке C:\Python34\DLLs) и язык программирования, позволяющий использовать эту библиотеку (например, Python). Следует 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.

Поскольку 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.

Итак, на странице **http://www.sqlite.org/download.html** находим раздел **Precompiled Binaries for Windows**, загружаем оттуда архив с этой программой, а затем распаковываем его в текущую папку. Далее копируем хранящийся в этом архиве файл sqlite3.exe в каталог, с которым будем в дальнейшем работать, — например, в C:\book.

17.1. Создание базы данных

Попробуем создать новую базу данных, для чего прежде всего запускаем командную строку, выбрав в меню **Пуск** пункт **Выполнить**. В открывшемся окне набираем команду `cmd` и нажимаем кнопку **OK** — откроется черное окно с приглашением для ввода команд. Переходим в папку C:\book, выполнив команду:

```
cd C:\book
```

В командной строке должно появиться 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 также можно установить и 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, новая база данных с этим именем будет создана и открыта для дальнейшей работы. Если такая база данных уже существует, то она просто откроется без удаления содержимого. Результат выполнения команды будет выглядеть так:

```
SQLite version 3.8.9 2015-04-08 12:16:33
Enter ".help" for usage hints.
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.

Строка `sqlite>` здесь является приглашением для ввода SQL-команд. Каждая SQL-команда должна завершаться точкой с запятой. Если точку с запятой не ввести и нажать клавишу <Enter>, то приглашение примет вид `...>`. В качестве примера получим версию SQLite:

```
sqlite> SELECT sqlite_version();
3.8.9
sqlite> SELECT sqlite_version()
   ...> ;
3.8.9
```

SQLite позволяет использовать *комментарии*. Однострочный комментарий начинается с двух тире и заканчивается 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
3.8.9
sqlite> SELECT sqlite_version(); /* Комментарий после SQL-команды */
3.8.9
```

Чтобы завершить работу с SQLite и закрыть базу данных, следует выполнить команду `.exit` **или** `.quit`.

17.2. 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 такие 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. И если таблица уже существует, а ключевые слова** `IF NOT EXISTS` **не указаны, то будет выведено сообщение об ошибке:**

```
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, значе-**

ние 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 параметрах <Название таблицы> и <Название поля> 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 избежать и удается, 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, попытка создать таблицу при наличии уже существующей одноименной таблицы приведет к выводу сообщения об ошибке. SQL-команда DROP TABLE позволяет удалить как обычную, 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 (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 их 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, содер-**

жащую 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, 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|13:21:01|2015-04-14|2015-04-14 13:21:01
sqlite> /* Текущая дата на компьютере: 2015-04-14 16:21:01 */
sqlite> DROP TABLE tmp3;
```

- COLLATE <Функция> — **задает функцию сравнения для класса** TEXT. **Могут быть указаны функции** BINARY **(обычное сравнение — значение по умолчанию),** NOCASE **(сравнение без учета регистра) и** RTRIM **(предварительное удаление лишних пробелов справа). 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 функции** 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 (первичные ключи служат в качестве идентификатора, однозначно обозначающего запись). 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 из хранящихся в поле чисел. 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 получит значение этого поля, на единицу большее максимального из хранящихся во всех записях таблицы. Если удалить последнюю запись, а затем вставить новую запись, то запись будет иметь такое же значение идентификатора, что и удаленная. Чтобы идентификатор 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 идентификатор последней вставленной записи — 3, а не 2, как это было в предыдущем примере. Таким образом, идентификатор новой записи всегда будет уникальным.

Если в таблице не существует поля с первичным ключом, то получить идентификатор 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 уникальный набор значений;
- 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 в транзакции предыдущими командами, 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 в транзакции предыдущими командами, 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 для поля не задано, 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;
```

17.3. 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>, ...) | DEFAULT VALUES;
```

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.

Вместо конструкции VALUES(<Список полей>) можно указать DEFAULT VALUES. В этом случае будет создана новая запись, все поля которой получат значения по умолчанию или NULL, если таковые не были заданы при создании таблицы.

Создадим 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 в таблицах дублирования данных — 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 названия рубрик расположены 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 узнаем 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. Поскольку поле id_user не указано, то ему присваивается значение по умолчанию. В таблице user поле id_user объявлено как первичный ключ, поэтому туда 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 так делать всегда. Тогда в дальнейшем можно будет изменить структуру таблицы (например, добавить поле) без необходимости изменять все SQL-запросы — достаточно будет для нового поля указать значение по умолчанию, а все старые запросы останутся 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 идентификатором (или значение индекса UNIQUE не уникально), то такая SQL-команда приводит к ошибке. Когда необходимо, чтобы имеющиеся 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>, ...) | DEFAULT VALUES;
```

Заменим название рубрики с идентификатором 3:

```
sqlite> REPLACE INTO rubr VALUES (3, 'Игры');
sqlite> SELECT * FROM rubr;
1|Программирование
2|Музыка
3|Игры
```

17.4. 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 создании таблицы. После ключевого слова SET указываются названия полей и их новые значения после знака равенства. Чтобы ограничить набор изменяемых записей, применяется инструкция WHERE. Обратите внимание на то, что если не указано <Условие>, то в таблице будут обновлены все записи. 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 из таблицы будут удалены все записи. В противном случае удаляются только записи, соответствующие условию. Для 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 фрагментации таблицы. Чтобы освободить неиспользуемое пространство, можно воспользоваться SQL-командой VACUUM. Обратите внимание на то, что эта SQL-команда может изменить порядок нумерации в специальных полях ROWID, OID и _ROWID_.

17.5. Изменение структуры 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, а именно: переименование таблицы и добавление поля. 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, что 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
```

17.6. 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 в указанной таблице все записи, которые удовлетворяют условию в инструкции WHERE. Если инструкция WHERE не указана, то из таблицы будут возвращены все записи. 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 из таблицы rubr все записи:

```
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 и таблицам. Это особенно полезно при выборке из нескольких таблиц сразу. Для 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 перед названием поля через точку задать псевдоним, 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 говорит, что возвращаются все записи. Если указано слово DISTINCT, то в результат попадут лишь 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. Для 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. Рассмотрим 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 в виде целого числа. Выведем сумму значений коэффициентов релевантности:

```
sqlite> SELECT SUM(iq) FROM site;
4000
```

- TOTAL(<Поле>) — то же самое, что и SUM(), но результат возвращается в виде числа с плавающей точкой:

```
sqlite> SELECT TOTAL(iq) FROM site;
4000.0
```

- GROUP_CONCAT(<Поле>[, <Разделитель>]) — возвращает строку, которая содержит все значения из указанного поля, разделенные указанным разделителем. Если разделитель не указан, используется запятая:

```
sqlite> SELECT GROUP_CONCAT(name_rubr) FROM rubr;
Программирование,Музыка,Поисковые порталы
sqlite> SELECT GROUP_CONCAT(name_rubr, ' | ') FROM rubr;
Программирование | Музыка | Поисковые порталы
```

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 отсортирова-

ны 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. 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;
```

17.7. Выбор записей из нескольких таблиц

SQL-команда SELECT позволяет выбирать записи сразу из нескольких таблиц одновременно.

Проще всего это сделать, перечислив нужные таблицы через запятую в инструкции FROM и указав в инструкции WHERE через запятую пары полей, являющиеся для этих таблиц свя-

зующими. Причем в условии и перечислении полей вначале указывается название таблицы (или псевдоним), а затем через точку название поля. Для 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;
```

Связать таблицы позволяет также оператор `JOIN`**, который имеет два синонима:** `CROSS JOIN` **и** `INNER JOIN`**. Переделаем наш предыдущий пример с использованием оператора** `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`**, также в инструкции** `WHERE` **можно указать дополнительное условие. Для 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
```

17.8. Условия в инструкциях *WHERE* и *HAVING*

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 (обычное сравнение — значение по умолчанию), NOCASE (без учета регистра) или RTRIM (предварительное удаление лишних пробелов справа). 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 (не** NULL**);**
- IS NULL **или** ISNULL **— проверка на отсутствие значения (**NULL**);**
- 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 3.8.9
```

```
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
```

Да, здесь результат выполнения становится более логичным. Получается, что 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 cp866 и Windows-1251, поэтому результат сравнения строк в этих кодировках может быть некорректным. С кодировкой UTF-8 мы еще поработаем в следующей главе, а сейчас 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.

17.9. Индексы

Записи в таблицах расположены в том порядке, в котором они были добавлены. Чтобы найти какие-либо данные, необходимо каждый раз просматривать все записи. Для ускорения выполнения запросов применяются *индексы*, или *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 в качестве первичного ключа сразу несколько полей.

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. Для 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|0|SEARCH TABLE site USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|0|SCAN TABLE site (~100000 rows)
```

В первом случае фраза USING INTEGER 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|0|SCAN TABLE site (~100000 rows)
sqlite> CREATE INDEX index_rubr ON site (id_rubr);
sqlite> EXPLAIN QUERY PLAN SELECT * FROM site WHERE id_rubr=1;
0|0|0|SEARCH TABLE site USING INDEX index_rubr (id_rubr=?) (~10 rows)
```

Обратите внимание на то, что после создания индекса добавилась фраза USING 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 фрагментации файла с базой данных, в результате чего в нем появляется неиспользуемое свободное пространство. Чтобы удалить его, можно воспользоваться SQL-командой VACUUM.

Вся статистическая информация об индексах хранится в специальной таблице sqlite_stat1. Пока в ней 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
rubr||3
users||1
```

17.10. Вложенные запросы

Результаты выполнения инструкции SELECT можно использовать в других инструкциях, создавая *вложенные запросы*. Для создания таблицы с помощью вложенного запроса служит следующий формат:

```
CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS]
[<Название базы данных>.]<Название таблицы> AS <Запрос SELECT>;
```

Для 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, указанными 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
```

17.11. 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 таковая 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];
```

Чтобы нормально завершить транзакцию, следует выполнить инструкции COMMIT или END — любая из них сохраняет все изменения и завершает 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>];
```

Для 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 закроется или отключится компьютер.

Когда ошибка возникает в одной из инструкций внутри транзакции, то запускается 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 соответствующее сообщение. Все изменения, сделанные 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
```

В процессе выполнения транзакции база данных блокируется, после чего ни одна другая транзакция не сможет внести в нее изменения, пока не будет выполнена операция завершения (COMMIT или END) или отмены (ROLLBACK). Это делается во избежание конфликтов, когда разные транзакции пытаются внести изменения в одну и ту же запись таблицы.

Мы имеем возможность управлять режимом блокировки базы данных. Для этого при ее запуске после ключевого слова BEGIN следует указать обозначение нужного режима:

- DEFERRED — база данных блокируется при выполнении первой операции чтения или записи, что встретилась после оператора BEGIN. Сам же этот оператор не блокирует базу. Другие транзакции могут читать из заблокированной базы, но не могут в нее записывать. Этот режим используется по умолчанию;
- IMMEDIATE — база данных блокируется непосредственно оператором BEGIN. Другие транзакции могут читать из заблокированной базы, но не могут в нее записывать;
- EXCLUSIVE — база данных блокируется непосредственно оператором BEGIN. Другие транзакции не могут ни читать из заблокированной базы, ни записывать в нее.

В большинстве случаев используется режим блокировки DEFERRED. Остальные режимы применяются лишь в особых случаях.

Пример запуска транзакции, полностью блокирующей базу:

```
sqlite> BEGIN EXCLUSIVE TRANSACTION;
sqlite> -- База данных заблокирована
sqlite> -- Выполняем какие-либо операции с базой
sqlite> COMMIT TRANSACTION;
sqlite> -- Транзакция завершена, и база разблокирована
```

Вместо запуска транзакции с помощью инструкции BEGIN можно создать именованную метку, выполнив инструкцию 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
```

17.12. Удаление таблицы и базы данных

Удалить таблицу позволяет инструкция DROP TABLE. Удалить можно как обычную, так и временную таблицу. Все индексы, связанные с таблицей, также удаляются. Формат инструкции:

```
DROP TABLE [IF EXISTS] [<Название базы данных>.]<Название таблицы>;
```

Поскольку 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.

> ***Примечание***
>
> Для интерактивной работы с базами данных SQLite удобно пользоваться бесплатной программой Sqliteman. Ее можно загрузить по интернет-адресу **http://sourceforge.net/projects/sqliteman/**.

ГЛАВА 18

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 и в дополнительной установке не нуждается.

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**.

Разумеется, модуль `sqlite3` поддерживает эту спецификацию, а также предоставляет некоторые нестандартные возможности. Поэтому, изучив методы и атрибуты этого модуля, вы получите достаточно полное представление о стандарте 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 или четырех чисел. Пример:

```
>>> sqlite3.sqlite_version
'3.8.3.1'
>>> sqlite3.sqlite_version_info
(3, 8, 3, 1)
```

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.

18.1. Создание и открытие базы данных

Для создания и открытия базы данных служит функция `connect()`. Функция имеет следующий формат:

```
connect(database[, timeout][, isolation_level][, detect_types]
        [, factory][, check_same_thread][, cached_statements][, uri = False])
```

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 не являются обязательными и могут быть указаны в произвольном порядке путем присвоения значения названию параметра. Так, параметр `timeout` задает время ожидания снятия блокировки с открываемой базы данных (по умолчанию — пять секунд). Предназначение остальных параметров мы рассмотрим 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
```

Поддержка необязательного параметра `uri` появилась в Python 3.4. Если его значение равно `True`, путь к базе данных должен быть указан в виде интернет-адреса формата `file:///<Путь к файлу>`. В этом случае можно задать дополнительные параметры соединения с базой, перечислив их в конце интернет-адреса в виде пар `<Имя параметра>=<Значение параметра>` и отделив от собственно пути символом `?`, а друг от друга — символом `&`. Наиболее интересные для нас параметры таковы:

- `mode=<Режим доступа>` — задает режим доступа к базе. Поддерживаются значения `ro` (только чтение), `rw` (чтение и запись — при этом база уже должна существовать), `rwc` (чтение и запись — если база данных не существует, она будет создана) и `memory` (база данных располагается исключительно в оперативной памяти и удаляется после закрытия);

- `immutable=1` — указывает, что база полностью недоступна для записи (например, записана на компакт-диске, не поддерживающем запись). В результате отключается механизм транзакций и блокировок SQLite, что позволяет несколько повысить производительность.

Примеры доступа к базе данных по интернет-адресу:

```
>>> import sqlite3
>>> # Доступ к базе, хранящейся в файле c:\book\testdb.db
>>> con = sqlite3.connect(r"file:///c:/book/testdb.db", uri = True)
>>> con.close()
>>> # Доступ только для чтения
>>> con = sqlite3.connect(r"file:///c:/book/testdb.db?mode=ro", uri = True)
>>> con.close()
>>> # Доступ к неизменяемой базе данных
>>> con = sqlite3.connect(r"file:///f:/data.db?immutable=1", uri = True)
>>> con.close()
```

18.2. Выполнение запросов

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, метод возбуждает исключение. Для 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 as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
cur.close()                   # Закрываем объект-курсор
con.close()                   # Закрываем соединение
input()
```

Сохраняем код в файле, а затем запускаем его с помощью двойного щелчка на значке файла. Обратите внимание на то, что мы работаем с кодировкой UTF-8, которая используется в SQLite по умолчанию;

- `execute(<SQL-запрос>[, <Значения>])` — выполняет один SQL-запрос. Если в процессе выполнения запроса возникает ошибка, возбуждается 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 as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()            # Завершаем транзакцию
cur.close()                 # Закрываем объект-курсор
con.close()                 # Закрываем соединение
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`), нужно 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, нужно 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.

Для примера заполним таблицу с рубриками этими способами:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()                  # Создаем объект-курсор
t1 = ("Программирование",)
t2 = (2, "Музыка")
d  = {"id": 3, "name": """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 as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()                    # Завершаем транзакцию
cur.close()                         # Закрываем объект-курсор
con.close()                         # Закрываем соединение
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, "http://wwwadmin.ru", "Название", "", 100),
          (1, 1, "http://python.org", "Python", "", 1000),
          (1, 3, "http://google.ru", "Гугль", "", 3000)
      ]
sql = """\
INSERT INTO site (id_user, id_rubr, url, title, msg, iq)
VALUES (?, ?, ?, ?, ?, ?)
"""
try:
    cur.executemany(sql, arr)
except sqlite3.DatabaseError as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()                    # Завершаем транзакцию
cur.close()                         # Закрываем объект-курсор
con.close()                         # Закрываем соединение
input()
```

Объект соединения также поддерживает методы execute(), executemany() и executescript(), которые позволяют выполнить запрос без создания объекта-курсора. Эти методы не входят в спецификацию DB-API 2.0. Для примера изменим название рубрики с идентификатором 3 (листинг 18.1).

Листинг 18.1. 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 as err:
    print("Ошибка:", err)
else:
    con.commit()                    # Завершаем транзакцию
    print("Запрос успешно выполнен")
con.close()                         # Закрываем соединение
input()
```

Объект-курсор поддерживает несколько атрибутов:

- `lastrowid` — индекс последней записи, добавленной с помощью инструкции `INSERT` и метода `execute()`. Если индекс не определен, то атрибут будет содержать 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 as err:
    print("Ошибка:", err)
else:
    con.commit()                    # Завершаем транзакцию
    print("Запрос успешно выполнен")
    print("Индекс:", cur.lastrowid)
cur.close()                         # Закрываем объект-курсор
con.close()                         # Закрываем соединение
input()
```

- `rowcount` — количество записей, измененных или удаленных методом `executemany()`. Если количество не определено, то атрибут имеет значение `-1`.

  Помимо этого, объект соединения поддерживает атрибут `total_changes`, возвращающий количество записей, которые были созданы, изменены или удалены во всех таблицах базы после того, как соединение было установлено:

```
con = sqlite3.connect("catalog.db")
. . .
print(con.total_changes)
```

- `description` — содержит кортеж кортежей. Каждый внутренний кортеж состоит из семи элементов: первый 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))
```

18.3. 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 со значениями отдельных ее полей, а затем перемещает указатель текущей позиции на следующую запись. Если записей больше нет, возвращается `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, 'unicross@mail.ru', 'password1')
>>> print(cur.fetchone())
None
```

- `__next__()` — **при каждом вызове возвращает одну запись из результата запроса в виде кортежа со значениями отдельных ее полей, а затем перемещает указатель текущей позиции на следующую запись. 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, 'unicross@mail.ru', 'password1')
>>> cur.__next__()
... Фрагмент опущен ...
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("{0}|{1}".format(id_rubr, name))

1|Программирование
2|Музыка
3|Поисковые порталы
4|Кино
```

- `fetchmany([size=cursor.arraysize])` — **при каждом вызове возвращает указанное количество записей из результата запроса в виде списка, а затем перемещает указатель текущей позиции на запись, следующую за последней возвращенной. Каждый элемент возвращенного списка является кортежем, содержащим значения полей записи. Количество элементов, выбираемых за один раз, задается с помощью необязательного параметра. Если он не указан, используется значение 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, 'Программирование')]
>>> cur.fetchmany(2)
[(2, 'Музыка'), (3, 'Поисковые порталы')]
```

```
>>> cur.fetchmany(3)
[(4, 'Кино')]
>>> cur.fetchmany()
[]
```

- `fetchall()` — **возвращает список всех (или всех оставшихся) записей из результата запроса в виде списка. Каждый элемент этого списка является кортежем, хранящим значения отдельных полей записи. Если записей больше нет, метод возвращает пустой список. Пример:**

```
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x0150E3B0>
>>> cur.fetchall()
[(1, 'Программирование'), (2, 'Музыка'), (3, 'Поисковые порталы'),
 (4, 'Кино')]
>>> cur.fetchall()
[]
>>> con.close()
```

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>
```

Для примера выведем записи из таблицы `user` в виде словаря (листинг 18.2).

Листинг 18.2. Использование атрибута `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: 'unicross@mail.ru', 2: 'password1', 'id_user': 1,
'passw': 'password1', 'email': 'unicross@mail.ru'}]"""
print(arr[0][1])          # Доступ по индексу
print(arr[0]["email"])    # Доступ по названию поля
```

```
cur.close()                 # Закрываем объект-курсор
con.close()                 # Закрываем соединение
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. Объект `Row` поддерживает итерации, доступ по индексу и метод `keys()`, который возвращает список с названиями полей. Переделаем наш предыдущий пример и используем объект `Row` (листинг 18.3).

Листинг 18.3. Использование объекта 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]))         # <class '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
input()
```

Как видно из результатов предыдущих примеров, все данные, имеющие в SQLite тип `TEXT`, возвращаются в виде строк. В предыдущей главе мы создали базу данных testdb.db и сохранили 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")
... Фрагмент опущен ...
sqlite3.OperationalError: Could not decode to UTF-8 column 'name_rubr'
>>> con.close()
```

При осуществлении преобразования 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 последовательность байтов, следует указать ссылку на функцию `bytes()` (листинг 18.4).

Листинг 18.4. Использование атрибута text_factory

```
>>> con = sqlite3.connect("testdb.db")
>>> con.text_factory = bytes # Название функции без круглых скобок!
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x014FE380>
>>> cur.fetchone()
(1, b'\xcf\xf0\xee\xe3\xf0\xe0\xec\xec\xe8\xf0\xee\xe2\xe0\xed\xe8\xe5')
```

Если необходимо вернуть строку, то внутри функции обратного вызова следует вызвать функцию `str()` и явно указать кодировку данных. Функция обратного вызова должна принимать один параметр и возвращать преобразованную строку. Выведем текстовые данные в виде строки (листинг 18.5).

Листинг 18.5. Указание пользовательской функции преобразования

```
>>> con.text_factory = lambda s: str(s, "cp1251")
>>> cur.execute("SELECT * FROM rubr")
<sqlite3.Cursor object at 0x014FE380>
>>> cur.fetchone()
(1, 'Программирование')
>>> con.close()
```

18.4. 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, следует вызвать метод `rollback()` объекта соединения. Для примера добавим нового пользователя, а затем отменим транзакцию и выведем содержимое таблицы (листинг 18.6).

Листинг 18.6. 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, 'unicross@mail.ru', '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()`**, поскольку 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 18.7).**

Листинг 18.7. 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, 'unicross@mail.ru', 'password1'), (2, 'user@mail.ru', '')]
>>> con.close()
```

Атрибут `in_transaction` **класса соединения возвращает** `True`**, если в данный момент существует активная транзакция, и** `False` **— в противном случае. Попытаемся добавить в таблицу нового пользователя и посмотрим, какие значения будет хранить этот атрибут в разные моменты времени (листинг 18.7).**

Листинг 18.8. Получение состояния транзакции

```
>>> con = sqlite3.connect("catalog.db")
>>> cur = con.cursor()
>>> cur.execute("INSERT INTO user VALUES (NULL, 'user2@mail.ru', '')")
<sqlite3.Cursor object at 0x03C33460>    # Запущена транзакция
>>> con.in_transaction
True                                     # Есть активная транзакция
>>> con.commit()                         # Завершаем транзакцию
>>> con.in_transaction
False                                    # Нет активной транзакции
>>> cur.close()
>>> con.close()
```

18.5. 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 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 эта функция работать не будет.

Для примера создадим новую таблицу с одним полем, вставим три записи, а затем произведем сортировку стандартным методом и с помощью пользовательской функции (листинг 18.9).

Листинг 18.9. Сортировка записей

```
# -*- coding: utf-8 -*-
import sqlite3

def myfunc(s1, s2): # Пользовательская функция сортировки
    s1 = s1.lower()
    s2 = s2.lower()
    if s1 == s2:
        return 0
    elif s1 > s2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:", 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], end=" ") # Результат: Единица2 Единый единица1
print()
# Пользовательская сортировка
cur.execute("""SELECT * FROM words
               ORDER BY word COLLATE myfunc""")
for line in cur:
    print(line[0], end=" ") # Результат: единица1 Единица2 Единый
cur.close()
con.close()
input()
```

18.6. Поиск без учета регистра символов

Как уже говорилось в предыдущей главе, сравнение строк и поиск с помощью оператора `LIKE` для русских букв производится 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 обеих строк к верхнему или нижнему регистру. Но встроенные функции SQLite `UPPER()` и `LOWER()` с русскими буквами также 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 в виде строки указывается название функции, которое будет использоваться в SQL-командах. Количество параметров, принимаемых функцией, задается во втором параметре, причем параметры могут быть любого типа. Если функция принимает строку, то ее типом данных будет `str`. В третьем параметре указывается ссылка на пользовательскую функцию в программе. Для примера произведем поиск рубрики без учета регистра символов (листинг 18.10).

Листинг 18.10. 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 = "%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()
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 лишнее преобразование регистра будет производиться 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. При этом дату и время можно хранить разными способами — например, в числовом поле как количество секунд, прошедших с начала эпохи (см. об этом также в *разд. 18.9*). Для преобразования количества секунд в другой формат следует создать пользовательскую функцию форматирования (листинг 18.11).

Листинг 18.11. 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(1429100920)")
print(cur.fetchone()[0]) # Результат: 15.04.2015
cur.close()
con.close()
input()
```

18.7. Создание агрегатных функций

При изучении SQLite мы рассматривали встроенные агрегатные функции `COUNT()`, `MIN()`, `MAX()`, `AVG()`, `SUM()`, `TOTAL()` и `GROUP_CONCAT()`. Если возможностей этих функций окажется недостаточно, то можно определить пользовательскую агрегатную функцию. Связать на-

звание 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 создаваемой агрегатной функции в виде строки. В третьем параметре передается ссылка на класс (название класса без круглых скобок). Этот класс должен поддерживать два метода: `step()` и `finalize()`. Метод `step()` вызывается для каждой из обрабатываемых записей, и ему передаются параметры, количество которых задается во втором параметре метода `create_aggregate()`. Метод `finalize()` должен возвращать результат выполнения. Для примера выведем все названия рубрик в алфавитном порядке через разделитель (листинг 18.12).

Листинг 18.12. 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()
input()
```

18.8. Преобразование типов данных

SQLite поддерживает пять типов данных, для каждого из которых в модуле `sqlite3` определено соответствие с типом данных Python:

- `NULL` — значение `NULL`. Значение соответствует типу `None` в Python;
- `INTEGER` — целые числа. Соответствует типу `int`;
- `REAL` — вещественные числа. Соответствует типу `float`;

- TEXT — строки. По умолчанию преобразуется в тип str. Предполагается, что строка в базе данных хранится в кодировке UTF-8. Соответствие можно изменить с помощью атрибута text_factory;
- BLOB — бинарные данные. Соответствует типу bytes.

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. 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. Для примера создадим новую таблицу и сохраним в ней значения атрибутов класса (листинг 18.13).

Листинг 18.13. Сохранение в базе атрибутов класса

```
# -*- coding: utf-8 -*-
import sqlite3

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color

def my_adapter(car):
    return "{0}|{1}".format(car.model, car.color)

# Регистрируем функцию для преобразования типа
sqlite3.register_adapter(Car, my_adapter)
# Создаем экземпляр класса Car
car = Car("ВАЗ-2109", "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 as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()
cur.close()
con.close()
input()
```

Вместо регистрации функции преобразования типа можно внутри класса 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, который поддерживается SQLite. Создадим таблицу `cars2` и сохраним в ней значения атрибутов, используя метод `__conform__()` (листинг 18.14).

Листинг 18.14. Использование метода `__conform__()`

```
# -*- coding: utf-8 -*-
import sqlite3

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color
    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "{0}|{1}".format(car.model, car.color)

# Создаем экземпляр класса Car
car = Car("Москвич-412", "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 as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()
cur.close()
con.close()
input()
```

Чтобы восстановить объект Python из значения типа, поддерживаемого SQLite, следует зарегистрировать функцию обратного преобразования типов данных 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 — 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 оператор `|`):

- `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 18.15).**

Листинг 18.15. Использование значения sqlite3.PARSE_COLNAMES

```
# -*- coding: utf-8 -*-
import sqlite3, sys

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color
    def __repr__(self):
        s = "Модель: {0}, цвет: {1}".format(self.model, self.color)
        return s

def my_converter(value):
    value = str(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()
input()
```

Теперь выведем значение из таблицы cars2, **где мы указали тип данных прямо при создании поля (листинг 18.16).**

Листинг 18.16. Использование значения sqlite3.PARSE_DECLTYPES

```
# -*- coding: utf-8 -*-
import sqlite3, sys

class Car:
    def __init__(self, model, color):
        self.model, self.color = model, color
    def __repr__(self):
        s = "Модель: {0}, цвет: {1}".format(self.model, self.color)
        return s

def my_converter(value):
    value = str(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()
input()
```

18.9. 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 18.17).

Листинг 18.17. 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 числа 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]) # 2015-04-15 15:41:47
con.close()
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 18.18.

Листинг 18.18. 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]) # 2015-04-15
print(res[1]) # 2015-04-15 15:41:47.190000
con.close()
input()
```

18.10. Обработка исключений

Модуль sqlite3 поддерживает следующую иерархию исключений:

```
Exception
    Warning
    Error
        InterfaceError
        DatabaseError
            DataError
            OperationalError
            IntegrityError
            InternalError
            ProgrammingError
            NotSupportedError
```

Базовым классом самого верхнего уровня является класс Exception. 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 это 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.**

Для примера обработки исключений напишем программу, которая позволяет пользователям вводить название базы данных и SQL-команды в консоли (листинг 18.19).

Листинг 18.19. 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) as err:
        print("Не удалось подключиться к БД")
        input()
        sys.exit(0)
    return db

print("Введите название базы данных:", end=" ")
db_name = input()
con = db_connect(db_name)        # Подключаемся к базе
cur = con.cursor()
sql = ""
print("Чтобы закончить выполнение программы, введите <Q>+<Enter>")
while True:
    tmp = input()
    if tmp in ["q", "Q"]:
        break
    if tmp.strip() == "":
        continue
    sql = "{0} {1}".format(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) as err:
            print("Ошибка:", err)
        else:
            print("Запрос успешно выполнен")
        sql = ""
cur.close()
con.close()
```

Чтобы SQL-запрос можно было разместить на нескольких строках, мы выполняем 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
```

Язык Python также поддерживает протокол менеджеров контекста, который гарантирует выполнение завершающих действий вне зависимости 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 18.20).**

Листинг 18.20. Инструкция with...as

```
# -*- coding: utf-8 -*-
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 as err:
    print("Ошибка:", err)
else:
    print("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 as err:
    print("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
con.close()
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()`**.**

18.11. Трассировка выполняемых запросов

Иногда возникает необходимость выяснить, какой запрос обрабатывается в данный момент времени, и выполнить при этом какие-либо действия — т. е. произвести *трассировку*. Именно для таких случаев объект соединения, начиная с Python 3.3, поддерживает метод `set_trace_callback(<Функция>)`**. Он позволяет зарегистрировать функцию, которая будет выполнена после обработки каждой команды SQL. Эта функция должна принимать единственный параметр — строку с очередной обрабатываемой SQL-командой, и не должна возвращать результата. Давайте используем этот метод, чтобы выводить на экран каждую команду на доступ к базе данных, что будут выполняться в нашей программе (листинг 18.21).**

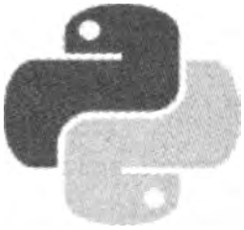
Листинг 18.21. Вывод выполняемых SQL-команд на экран

```
import sqlite3

# Объявляем функцию, которая станет выводить команды на экран
def tracer(command):
    print(command)

con = sqlite3.connect(r"C:\book\catalog.db")
con.set_trace_callback(tracer)          # Регистрируем функцию tracer()
con.execute("SELECT * FROM user;")
con.execute("SELECT * FROM rubr;")
con.close()
```

В результате выполнения этого кода каждый раз, когда вызывается метод `execute()`**, на экране будет появляться код SQL-запроса к базе, выполняемого этим методом.**

Чтобы отменить трассировку запросов, следует вызвать метод `set_trace_callback()`**, передав ему в качестве параметра** `None`**:**

```
con.set_trace_callback(None)
```



ГЛАВА 19

Доступ к базам данных MySQL

MySQL является наиболее популярной системой управления базами данных среди СУБД, не требующих платить 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 неудивительно: MySQL проста в освоении, имеет высокую скорость работы и предоставляет функциональность, доступную ранее только в коммерческих СУБД.

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 входит в состав Python. Кроме того, в состав стандартной библиотеки последнего 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/**. Кроме того, MySQL входит в состав пакетов хостинга, таких как OpenServer (**http://open-server.ru/**).

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 с ним работать. Если это не так, то сначала вам следует изучить специальную литературу по MySQL и лишь затем вернуться к изучению материала, описываемого в этой главе. Описание MySQL можно также найти в книгах «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера»[1] и «Разработка Web-сайтов с помощью Perl и MySQL»[2].

Для доступа к базе данных MySQL существует большое количество библиотек, написанных сторонними разработчиками. В этой главе мы рассмотрим функциональные возможности библиотек `MySQLClient` и `PyODBC`.

[1] Прохоренок Н. А., Дронов В. А. HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера. (+ Видеокурс). — 4-е изд. — СПб.: БХВ-Петербург, 2015.

[2] Прохоренок Н. А. Разработка Web-сайтов с помощью Perl и MySQL. — СПб.: БХВ-Петербург, 2009.

19.1. Библиотека *MySQLClient*

Библиотека `MySQLClient` является ответвлением от популярной во времена господства Python 2 библиотеки `MySQLdb` и полностью совместима с последней. Ключевые модули, реализующие поддержку MySQL, в обеих этих библиотеках носят одинаковое название — `MySQLdb`.

Загрузить дистрибутив `MySQLClient` в формате WHL (этот формат в последнее время активно используется для распространения дополнительных библиотек к Python) можно со страницы **http://www.lfd.uci.edu/~gohlke/pythonlibs/#mysqlclient**. Будьте внимательны — там хранится несколько редакций дистрибутива, предназначенных для различных версий Python. Для 32-разрядной редакции Python 3.4 следует загрузить файл mysqlclient-1.3.6-cp34-none-win32.whl, а для 64-разрядной редакции — файл mysqlclient-1.3.6-cp34-none-win_amd64.whl.

Как только нужный WHL-файл будет загружен, запустим командную строку Windows и наберем следующие команды:

```
cd <Полный путь к папке, где хранится загруженный WHL-файл>
c:\python34\scripts\pip install <Имя WHL-файла с дистрибутивом>
```

Через несколько секунд модуль будет установлен, о чем нас оповестят сообщения, появившиеся в окне командной строки.

Чтобы проверить работоспособность модуля, в окне **Python Shell** редактора IDLE набираем следующий код:

```
>>> import MySQLdb
>>> MySQLdb.__version__
'1.3.6'
```

Модуль `MySQLdb` является «оберткой» модуля `_mysql` и предоставляет интерфейс доступа, совместимый со спецификацией DB-API. Получить номер поддерживаемой версии спецификации можно с помощью атрибута `apilevel`:

```
>>> MySQLdb.apilevel
'2.0'
```

19.1.1. Подключение к базе данных

Для подключения к базе данных служит функция `connect()`, имеющая 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 пользователя. По умолчанию пароль пустой;
- `db` — название базы данных, которую необходимо выбрать для работы. По умолчанию никакая база данных не выбирается. Указать название базы данных также можно 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 сжатия нет;
- connect_timeout — время, в течение которого должно быть установлено соединение. Если соединение не удалось установить за указанное время, операция прерывается и возбуждается исключение. По умолчанию ограничения по времени нет;
- named_pipe — использовать именованный канал (применяется только в Windows). По умолчанию не используется;
- init_command — команда, передаваемая на сервер при подключении. По умолчанию не передается никаких команд;
- cursorclass — класс курсора. По умолчанию MySQLdb.cursors.Cursor;
- sql_mode — режим SQL;
- use_unicode — если параметр имеет значение True, то значения, хранящиеся в полях типов CHAR, VARCHAR и TEXT, будут возвращаться в виде Unicode-строк;
- read_default_file — местоположение конфигурационных файлов 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. В большинстве случаев сервер MySQL по умолчанию настроен на кодировку соединения latin1. Получить настройки кодировки позволяет метод get_character_set_info() (листинг 19.1).

Листинг 19.1. Получение настроек кодировки

```
>>> import MySQLdb # Подключаем модуль MySQLdb
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456")
>>> con.get_character_set_info()
{'mbmaxlen': 1, 'collation': 'latin1_swedish_ci', 'mbminlen': 1,
'name': 'latin1', 'comment': ''}
>>> con.close()
```

Поэтому при установке подключения с базой данных обязательно следует явно указать одну из русских кодировок (листинг 19.2).

Листинг 19.2. Указание кодировки соединения при подключении к базе

```
>>> import MySQLdb
>>> # Задаем кодировку 1251
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456", charset="cp1251")
```

```
>>> con.get_character_set_info()
{'mbmaxlen': 1, 'collation': 'cp1251_general_ci', 'mbminlen': 1,
'name': 'cp1251', 'comment': ''}
>>> con.close()

>>> # Задаем кодировку UTF-8
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456", charset="utf8")
>>> con.get_character_set_info()
{'mbmaxlen': 3, 'collation': 'utf8_general_ci', 'mbminlen': 1, 'name': 'utf8',
'comment': ''}
>>> con.close()
```

Обычно используют кодировку UTF-8 — как универсальную и наиболее часто применяемую в настоящее время.

Указать кодировку также позволяет метод `set_character_set(<Кодировка>)` **объекта соединения (листинг 19.3).**

Листинг 19.3. Указание кодировки соединения методом set_character_set()

```
>>> import MySQLdb
>>> con = MySQLdb.connect(host="localhost", user="root",
                          passwd="123456")
>>> con.set_character_set("utf8")
>>> con.get_character_set_info()
{'mbmaxlen': 3, 'collation': 'utf8_general_ci', 'mbminlen': 1, 'name': 'utf8',
'comment': ''}
>>> con.close()
```

В некоторых версиях `MySQLClient` **и Python поиск файлов с кодовыми таблицами по умолчанию производится в папке C:\mysql\share\charsets\. Поэтому попытка задать кодировку в параметре** `charset` **без указания значения в параметре** `read_default_file` **приводит 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\\)")
```

Чтобы избежать ошибки, необходимо в параметре `read_default_file` **указать путь к конфигурационному файлу MySQL. Причем в этом файле в директиве** `character-sets-dir`**, находящейся внутри секции** `[client]`**, должен быть задан путь к файлам с кодовыми таблицами:**

```
character-sets-dir="C:\\Program Files\\MySQL\\MySQL Server
5.5\\share\\charsets\\"
```

Если ранее производились попытки подключения к базе данных 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 таблица-

ми, все равно произойдет ошибка. Пример указания пути к конфигурационному файлу приведен в листинге 19.4.

Листинг 19.4. Указание пути к конфигурационному файлу

```
>>> import MySQLdb # Подключаем модуль MySQLdb
>>> ini = r"C:\Program Files\MySQL\MySQL Server 5.5\my.ini"
>>> con = MySQLdb.connect(host="localhost", user="root",
          passwd="123456", read_default_file=ini, charset="cp1251")
>>> con.get_character_set_info()
{'mbmaxlen': 1, 'collation': 'cp1251_general_ci', 'mbminlen': 1,
'name': 'cp1251', 'comment': '', 'dir':
'C:\\Program Files\\MySQL\\MySQL Server 5.5\\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.

19.1.2. Выполнение запросов

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:

```
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", charset="utf8")
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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
cur.close()                 # Закрываем объект-курсор
con.close()                 # Закрываем соединение
input()
```

Обратим внимание на обратные апострофы, присутствующие в представленном здесь коде. Ими в MySQL выделяются имена баз данных, таблиц и полей. Если этого не сделать, запрос не будет обработан, и возникнет ошибка.

Теперь подключимся к новой базе данных, создадим таблицу и добавим запись:

```
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", charset="utf8", 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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    con.commit()
cur.close()
con.close()
input()
```

В этом примере мы применили метод `commit()` объекта соединения. Метод `commit()` позволяет завершить транзакцию, которая запускается автоматически. При использовании транзакций в MySQL существуют нюансы. Так, таблица типа `MyISAM`, которую мы создали в этом примере, не поддерживает транзакции, поэтому вызов метода `commit()` можно опустить. Тем не менее, как видно из примера, указание метода не приводит в ошибке. Однако попытка отменить изменения с помощью метода `rollback()` не приведет к желаемому результату, а 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()`. При этом отменить изменения можно будет 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 получит 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.

Для примера заполним таблицу с городами этими способами:

```
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", charset="utf8", 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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
cur.close()
con.close()
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()`:

```
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", charset="utf8", 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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
cur.close()
con.close()
input()
```

Объект-курсор поддерживает несколько атрибутов:

- `lastrowid` — **индекс последней добавленной с помощью инструкции** `INSERT` **и метода** `execute()` **записи. Вместо атрибута** `lastrowid` **можно воспользоваться методом** `insert_id()` **объекта соединения. Для примера добавим новый город и выведем его индекс двумя способами:**

```
import MySQLdb
con = MySQLdb.connect(host="localhost", user="root",
                      passwd="123456", charset="utf8", 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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
    print("Индекс:", cur.lastrowid)
    print("Индекс:", con.insert_id())
cur.close()
con.close()
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))
```

19.1.3. 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 со значениями отдельных полей, а затем перемещает указатель текущей позиции на следующую запись. 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", charset="utf8", db="python")
>>> cur = con.cursor()
>>> cur.execute("SET NAMES utf8")
0
>>> sql = "SELECT `name_city` FROM `city` WHERE `id_city`<3"
>>> cur.execute(sql)
2
>>> cur.rowcount       # Количество записей
2
>>> con.field_count() # Количество полей
1
>>> cur.fetchone()
('Санкт-Петербург',)
>>> cur.fetchone()
('Москва',)
>>> print(cur.fetchone())
None
```

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 из результата запроса кортеж записей, а затем перемещает указатель текущей позиции на запись, следующую за последней возвращенной. Каждый элемент кортежа также является кортежем, хранящим значения отдельных полей. Количество элементов, выбираемых за один раз, задается с помощью необязательного параметра — если он не задан, используется значение 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`>2"
>>> cur.execute(sql)
4
>>> cur.arraysize
1
>>> cur.fetchmany()
(('Новгород',),)
>>> cur.fetchmany(2)
(('Новый \' " город',), ('Пермь',))
>>> cur.fetchmany(3)
(('Самара',),)
```

```
>>> cur.fetchmany()
()
```

- `fetchall()` — возвращает кортеж всех (или всех оставшихся) записей из результата запроса. Каждый элемент кортежа также является кортежем, хранящим значения отдельных полей. Если записей больше нет, возвращается пустой кортеж. Пример:

```
>>> sql = "SELECT `name_city` FROM `city` WHERE `id_city`>4"
>>> cur.execute(sql)
2
>>> cur.fetchall()
(('Пермь',), ('Самара',))
>>> cur.fetchall()
()
```

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`>4"
>>> cur.execute(sql)
2
>>> for row in cur: print(row[0])

Пермь
Самара
```

Все рассмотренные методы после возвращения результата перемещают указатель текущей позиции. Если необходимо вернуться в начало или переместить указатель к произвольной записи, 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, возбуждается исключение `IndexError`. Для примера переместим указатель в начало, выведем все записи, а затем вернемся на одну запись назад (листинг 19.5).

Листинг 19.5. Перемещение указателя текущей позиции

```
>>> cur.scroll(0, "absolute")
>>> res = cur.fetchall()
>>> for name in res: print(name[0])

Пермь
Самара
>>> cur.scroll(-1, "relative")
>>> res = cur.fetchall()
>>> for name in res: print(name[0])

Самара
```

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. Для примера выведем запись с идентификатором 5 в виде словаря (листинг 19.6).

Листинг 19.6. Получение записей в виде словаря

```
>>> con = MySQLdb.connect(host="localhost", user="root",
          passwd="123456", charset="utf8", db="python")
>>> cur = con.cursor(MySQLdb.cursors.DictCursor)
>>> sql = "SELECT * FROM `city` WHERE `id_city`=5"
>>> cur.execute(sql)
1
>>> cur.fetchone()
{'id_city': 5, 'name_city': 'Пермь'}
>>> con.close()
```

19.2. Библиотека *PyODBC*

Библиотека PyODBC позволяет работать с любыми источниками, поддерживаемыми ODBC, — в частности, с базами данных Access, SQL Server, MySQL и таблицами Excel. В этом разделе мы рассмотрим возможности этой библиотеки применительно к базе данных MySQL.

Загрузить дистрибутив PyODBC в формате WHL можно со страницы **http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyodbc**. Опять же, будьте внимательны — там хранится несколько редакций дистрибутива, предназначенных для различных версий Python. Для 32-разрядной редакции Python 3.4 следует загрузить файл pyodbc-3.0.7-cp34-none-win32.whl, а для 64-разрядной — файл pyodbc-3.0.7-cp34-none-win_amd64.whl. После чего установить библиотеку тем же способом, как ранее устанавливали MySQLClient.

Чтобы проверить работоспособность библиотеки в окне **Python Shell** редактора IDLE, набираем следующий код:

```
>>> import pyodbc
>>> pyodbc.version
'3.0.7'
```

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 дистрибутивный файл, наиболее подходящий для нашей системы: mysql-connector-odbc-5.3.4-win32.msi для 32-разрядной редакции Windows и mysql-connector-odbc-5.3.4-winx64.msi — для 64-разрядной ее редакции. Затем запускаем его с помощью двойного щелчка. После установки драйвера можно подключиться к MySQL.

19.2.1. Подключение к базе данных

Для подключения к базе данных служит функция `connect()`. Функция имеет следующий формат:

```
connect(<Строка подключения>[, autocommit=False]
                             [, unicode_results=False]
                             [, readonly=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 5.3 Unicode Driver}"` при использовании кодировки UTF-8 и `"{MySQL ODBC 5.3 ANSI Driver}"` при использовании однобайтовых кодировок — например, 1251;
- `SERVER` — имя хоста. По умолчанию используется локальный хост;
- `UID` — имя пользователя;
- `PWD` — пароль для авторизации пользователя. По умолчанию пароль пустой;
- `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/5.5/en/connector-odbc-info.html**.

Для примера подключимся к базе данных `python`, которую мы создали при изучении библиотеки `MySQLClient`:

```
>>> import pyodbc
>>> s = "DRIVER={MySQL ODBC 5.3 Unicode 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 этого 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 типов `CHAR`, `VARCHAR` и `TEXT`, будут возвращаться в виде Unicode-строк. По умолчанию параметр имеет значение `False`.

Если для параметра `readonly` задать значение `True`, база данных будет доступна лишь для чтения. По умолчанию этот параметр имеет значение `False`.

19.2.2. Выполнение запросов

После подключения к базе данных необходимо с помощью метода `cursor()` создать объект-курсор. 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, возбуждается исключение. Метод возвращает объект-курсор. Создадим три таблицы в базе данных `python`:

```
import pyodbc
s = "DRIVER={MySQL ODBC 5.3 Unicode 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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
```

```
cur.close()
con.close()
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 подставить в SQL-запрос необработанные данные, то пользователь получит 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 во втором параметре метода `execute()`. Их также можно передать как обычные параметры этого метода. Для примера заполним таблицу с рубриками и добавим нового пользователя:

```
import pyodbc
s = "DRIVER={MySQL ODBC 5.3 Unicode 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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
cur.close()
con.close()
input()
```

- `executemany(<SQL-запрос>, <Последовательность>)` — выполняет SQL-запрос несколько раз, при этом подставляя значения из последовательности. Если в процессе выполнения запроса возникает ошибка, возбуждается исключение. Заполним таблицу `site` с помощью метода `executemany()`:

```
import pyodbc
s = "DRIVER={MySQL ODBC 5.3 Unicode 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("Ошибка:", err)
else:
    print("Запрос успешно выполнен")
cur.close()
con.close()
input()
```

19.2.3. 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 на следующую запись. Если записей больше нет, метод возвращает значение `None`. Выведем записи из таблицы с рубриками:

```
>>> import pyodbc
>>> s = "DRIVER={MySQL ODBC 5.3 Unicode 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, 'Поисковые \' " порталы')
>>> 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 представляет собой атрибут этого объекта и поэтому указывается через точку. Если вывести полностью содержимое объекта, то возвращается кортеж со значениями. Так как при подключении в параметре `unicode_results` мы указали значение `True`, 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 на запись, следующую за последней возвращенной. Каждый элемент списка является объектом `Row`.

Количество элементов, выбираемых за один раз, задается с помощью необязательного параметра — если он не указан, используется значение 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, 'Поисковые \' " порталы')]
>>> 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, 'Программирование'), (2, 'Поисковые \' " порталы')]
>>> 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 параметрами полей, полученными 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 получит 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 19.7).**

Листинг 19.7. 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 этом 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 19.8).**

Листинг 19.8. 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 оказались экранированы.

ГЛАВА 20

Библиотека *Pillow*. Работа с изображениями

Для работы с изображениями в Python наиболее часто используется библиотека `Pillow`, полностью совместимая с аналогичной библиотекой `PIL`, которая активно применялась во времена Python 2. В этой главе мы рассмотрим базовые возможности этой библиотеки.

Дистрибутив библиотеки `Pillow` в формате WHL можно найти на странице **http://www.lfd.uci.edu/~gohlke/pythonlibs/#pillow**. Отыскиваем дистрибутив, подходящий для нашей версии Python, и загружаем его. Так, для 32-разрядной редакции Python 3.4 следует загрузить файл Pillow-2.8.1-cp34-none-win32.whl, а для 64-разрядной — файл Pillow-2.8.1-cp34-none-win_amd64.whl. После чего устанавливаем библиотеку способом, описанным в *главе 19*.

Ключевой модуль библиотеки носит имя `PIL`. Проверим его работоспособность, набрав в окне **Python Shell** редактора IDLE следующий код:

```
>>> from PIL import Image
>>> Image.VERSION
'1.1.7'
```

> ***Примечание***
>
> Документацию по библиотеке `Pillow` можно найти по интернет-адресу **http://pillow.readthedocs.org/**.

20.1. Загрузка готового изображения

Для открытия файла с готовым изображением служит 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. Необязательный второй параметр задает режим доступа к файлу — если он не указан, файл будет доступен лишь для чтения.

Откроем файл foto.gif, который расположен в текущем рабочем каталоге:

```
>>> 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)
>>> img.format                 # Выводим формат изображения
'GIF'
>>> f.close()                  # Закрываем файл
```

Как видно из примера, формат изображения определяется автоматически. Следует также заметить, что после открытия файла с помощью функции `open()` само изображение не загружается сразу из файла в память — загрузка производится при первой операции с изображением.

Загрузить изображение явным образом, если такая нужда возникнет, позволяет метод `load()` объекта изображения. Он возвращает объект, с помощью которого можно получить доступ к отдельным пикселам изображения. Указав внутри квадратных скобок два значения: горизонтальную и вертикальную координаты пиксела, можно получить или задать его цвет (листинг 20.1).

Листинг 20.1. 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 20.2.

Листинг 20.2. 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. (Так, в Windows 8, которой пользуется один из авторов, таковой является WinRT-приложение Photos.)

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 20.3).

Листинг 20.3. 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 GIF, а сохранили его в форматах JPEG и BMP. То есть, можно открывать изображения в одном формате и конвертировать их в другой формат. Если сохранить изображение не удалось, возбуждается исключение `IOError`. Когда параметр `<Формат>` не задан, формат изображения определяется по расширению файла, однако если методу `save()` в качестве первого параметра передан файловый поток, формат должен быть указан.

В параметре `<Опции>` можно передать 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 соответствующей документации.

20.2. Создание нового изображения

Библиотека Pillow позволяет работать не только с готовыми изображениями, но и создавать их. 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 битов, черно-белое;
- `P` — 8 битов, цветное (256 цветов);
- `RGB` — 24 бита, цветное;
- `RGBA` — 32 бита, цветное с альфа-каналом;
- `CMYK` — 32 бита, цветное;
- `YCbCr` — 24 бита, цветное, видеоформат;
- `LAB` — 24 бита, цветное, используется цветовое пространство Lab;

- `HSV` — 24 бита, цветное, используется цветовое пространство HSV;
- `I` — 32 бита, цветное, цвета кодируются целыми числами;
- `F` — 32 бита, цветное, цвета кодируются вещественными числами.

Во втором параметре необходимо передать размер создаваемого изображения (холста) в виде кортежа из двух элементов: (<Ширина>, <Высота>). В необязательном параметре <Цвет фона> задается цвет фона. Если параметр не указан, то фон 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 20.4.

Листинг 20.4. 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
```

20.3. Получение информации об изображении

Получить информацию об изображении позволяют следующие атрибуты объекта изображения:

- `size` — размер изображения в виде кортежа из двух элементов: (<Ширина>, <Высота>);
- `format` — формат изображения в виде строки (например: `'GIF'`, `'JPEG'` и т. д.);
- `mode` — режим в виде строки (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 20.5).

Листинг 20.5. 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',
>>> img.info
{'version': 'GIF89a', 'background': 254}
>>> img = Image.open("foto.bmp")
>>> img.size, img.format, img.mode
((800, 600),        '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)}
```

20.4. Манипулирование изображением

Произвести различные манипуляции с загруженным изображением позволяют следующие методы:

- `copy()` — **создает копию изображения:**

```
>>> from PIL import Image
>>> 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** — *миниатюру*. **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** `LANCZOS`. **Если параметр не указан, используется значение** `BICUBIC`. **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.LANCZOS)
>>> img.size # Изменяется само изображение
(400, 300)
>>> img = Image.open("foto.jpg")
>>> img.thumbnail((400, 100), Image.LANCZOS)
>>> 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. Изменение размера производится не пропорционально, и если пропорции не соблюдены, то изображение будет искажено. В параметре** `<Фильтр>` **могут быть указаны фильтры** `NEAREST`, `BILINEAR`, `BICUBIC` **или** `LANCZOS`. **Если параметр не указан, используется значение** `NEAREST`. **Пример:**

```
>>> img = Image.open("foto.jpg")
>>> img.size # Исходные размеры изображения
(800, 600)
>>> img2 = img.resize((400, 300), Image.LANCZOS)
>>> img2.size # Пропорциональное уменьшение
(400, 300)
>>> img3 = img.resize((400, 100), Image.LANCZOS)
>>> img3.size # Изображение будет искажено
(400, 100)
```

- `rotate(<Угол>[, <Фильтр>][, expand=0])` — **поворачивает изображение на указанный угол, отмеряемый в градусах против часовой стрелки. Метод возвращает новое изображение. В параметре** `<Фильтр>` **могут быть указаны фильтры** `NEAREST`, `BILINEAR` **или** `BICUBIC`. **Если параметр не указан, используется значение** `NEAREST`. **Если параметр** `expand` **имеет значение** `True`, **то размер изображения будет увеличен таким образом, чтобы оно полностью поместилось: по умолчанию размер изображения сохраняется, а 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`, `ROTATE_270` **или** `TRANSPOSE`. **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° против часовой стрелки
>>> img5 = img.transpose(Image.ROTATE_180)
>>> img5.show() # Поворот на 180°
```

```
>>> img6 = img.transpose(Image.ROTATE_270)
>>> img6.show() # Поворот на 270°
>>> img7 = img.transpose(Image.TRANSPOSE)
>>> img7.show() # Поворот на 90° по часовой стрелке
```

- `crop((<X1>, <Y1>, <X2>, <Y2>))` — **вырезает прямоугольный фрагмент из исходного изображения. В качестве параметра указывается кортеж из четырех элементов: первые два элемента задают координату левого верхнего угла вырезаемого фрагмента, а вторые два элемента задают координату его 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 фрагмент
>>> img2.load() # Считываем фрагмент, создавая 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 верхнего угла закрашиваемой области, а вторые два элемента — координату ее 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, он задает начальную точку этой области. Для примера загрузим изображение, создадим его 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. Для 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 метод** `merge(<Режим>, <Каналы>)`**. Для 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>[, <Режим смешивания цветов>[, <Палитра>[, <Количество цветов>]]]])` — **преобразует изображение в указанный режим. Метод возвращает новое изображение. Третий параметр указывает способ получения сложных цветов из более простых путем смешивания и имеет смысл при преобразовании изображений формата** `RGB` **или** `L` **в формат** `P` **или** `1` **— доступны значения** `None` **(смешивание не выполняется) и** `Image.FLOYDSTEINBERG` **(значение по умолчанию). Четвертый параметр задает тип палитры при преобразовании из** `RGB` **в** `P`**:** `Image.WEB` **(Web-совместимая палитра — значение по умолчанию) или** `Image.ADAPTIVE` **(адаптивная палитра). Пятый параметр задает количество цветов в палитре, по умолчанию — 256. Преобразуем изображение из формата** `RGB` **в режим** `RGBA`**:**

```
>>> img = Image.open("foto.jpg")
>>> img.mode
'RGB'
>>> img2 = img.convert("RGBA")
>>> img2.mode
'RGBA'
>>> img2.show()
```

Преобразуем изображение `RGB` **в формат** `P`**, указав смешивание цветов и адаптивную палитру в 128 цветов:**

```
>>> img = Image.open("foto.jpg")
>>> img.mode
'RGB'
>>> img2 = img.convert("P", None, Image.FLOYDSTEINBERG, Image.ADAPTIVE, 128)
>>> img2.mode
'P'
```

- `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:**

```
>>> from PIL import ImageFilter
>>> img = Image.open("foto.jpg")
>>> img2 = img.filter(ImageFilter.EMBOSS)
>>> img2.show()
```

20.5. 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
>>> 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 range(5, 31):
        draw.point( (n, 5), fill=(255, 0, 0) )
>>> img.show()
```

- `line(<Координаты>, fill=<Цвет>[, width=<Ширина>])` — **проводит линию между двумя точками. 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 и правого нижнего углов рисуемого прямоугольника. Нарисуем три прямоугольника: первый — 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> **указывается кортеж с координатами трех и более точек: из каждой пары элементов этого списка первая задает горизонтальную координату, вторая — вертикальную. 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> **указывается кортеж с координатами верхнего левого и правого нижнего углов прямоугольника, в который необходимо вписать эллипс. Из каждой пары элементов этого кортежа первый задает горизонтальную координату, второй — вертикальную. 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 начальный и конечный угол, между которыми будет отображена дуга. Угол, равный 0, расположен в крайней правой точке. Углы откладываются 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()
```

20.6. Библиотека *Wand*

Если приглядеться к контурам фигур, созданных с помощью класса `ImageDraw` из библиотеки `Pillow`, можно заметить, что граница отображается в виде ступенек. Сделать контуры более гладкими позволяет библиотека `Wand`, являющаяся программной оберткой популярного программного пакета обработки графики ImageMagick. Оба этих пакета не входят в состав Python и должны устанавливаться отдельно.

Сначала необходимо установить сам пакет ImageMagick. Для этого переходим на страницу **http://www.imagemagick.org/download/binaries/** и загружаем дистрибутивный файл ImageMagick-6.9.1-2-Q16-x86-dll.exe для 32-разрядной версии Windows или файл ImageMagick-6.9.1-2-Q16-x64-dll.exe — для 64-разрядной. После чего щелкаем мышью на полученном файле и следуем появляющимся на экране инструкциям. Установка ImageMagick не вызывает проблем и может быть выполнена с параметрами по умолчанию.

Теперь можно заняться библиотекой `Wand`. Одна из примечательных возможностей Python 3.4 — наличие встроенных инструментов для установки библиотек из *репозитория*[1] PyPI: **https://pypi.python.org/pypi**, поддерживаемого сообществом разработчиков этого языка. Ими-то мы и воспользуемся. Проверим, подключен ли наш компьютер к Интернету, откроем командную строку и наберем в ней следующую команду:

```
c:\python34\scripts\pip install Wand
```

[1] Репозиторий — это файловый интернет-архив, служащий централизованным хранилищем программ и дополнительных библиотек.

Через некоторое время дистрибутив `Wand` будет загружен и установлен, о чем нас уведомит соответствующее сообщение. Проверим, все ли прошло нормально, набрав в **Python Shell** строку:

```
>>> import wand
```

Если интерпретатор не выдал сообщения об ошибке, значит, `Wand` установлена и работает.

Чтобы увидеть разницу между `Pillow` и `Wand`, нарисуем два круга — сначала средствами первой библиотеки, потом средствами второй (листинг 20.6).

Листинг 20.6. Сравнение класса `ImageDraw` и модуля `wand`

```
>>> # Рисуем эллипс средствами Pillow
>>> 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="white", outline="red")
>>> img.show()
>>> input()
>>> # Рисуем эллипс средствами ImageMagick и Wand
>>> # Импортируем класс Image из модуля wand.image под именем
>>> # WandImage, чтобы избежать конфликта имен с одноименным классом
>>> # из модуля PIL
>>> from wand.image import Image as WandImage
>>> from wand.color import Color
>>> from wand.drawing import Drawing
>>> from wand.display import display
>>> img = WandImage(width = 300, height = 300, background = Color("white"))
>>> draw = Drawing()
>>> draw.stroke_color = Color("red")
>>> draw.fill_color = Color("white")
>>> draw.ellipse((150, 150), (150, 150))
>>> draw.draw(img)
>>> display(img)
```

Методика создания первого круга (с помощью `Pillow`) должна быть уже нам знакома — в отличие от методики рисования второго круга, когда мы задействовали средства `Wand`.

Итак, сначала мы создаем объект класса `Image`, определенный в модуле `wand.image` и представляющий рисуемое изображение. Конструктор этого класса в нашем случае имеет следующий формат вызова:

```
Image(width = <Ширина>, height = <Высота>[, background = <Цвет фона>])
```

Если цвет фона не указан, изображение будет иметь прозрачный фон. Пример:

```
>>> from wand.image import Image as WandImage
>>> from wand.color import Color
>>> img = WandImage(width = 400, height = 300, background = Color("black"))
>>> img
<wand.image.Image: e1038a4 '' (400x300)>
```

Цвет в `Wand` **задается в виде объекта класса** `Color`**, определенного в модуле** `wand.color`**. Конструктор этого класса в качестве параметра принимает строку с описанием цвета, которое может быть задано любым из знакомых нам способов. Примеры:**

```
>>> from wand.color import Color
>>> Color("white")                                  # Белый цвет
wand.color.Color('srgb(255,255,255)')
>>> Color("#FF0000")                                # Красный цвет
wand.color.Color('srgb(255,0,0)')
>>> Color("rgb(0, 255, 0)")                         # Зеленый цвет
wand.color.Color('srgb(0,255,0)')
>>> Color("rgba(0, 255, 0, 0.5)")                   # Полупрозрачный зеленый цвет
wand.color.Color('srgba(0,255,0,0.499992)')
```

Рисованием на изображении заведует определенный в модуле `wand.drawing` **класс** `Drawing`**. Создадим его, вызвав конструктор без параметров:**

```
>>> from wand.drawing import Drawing
>>> draw = wand.drawing.Drawing()
>>> draw
<wand.drawing.Drawing object at 0x028C9570>
```

Класс `Drawing` **поддерживает ряд свойств, позволяющих задать параметры рисуемых фигур. Вот они:**

- `stroke_color` — **цвет линий:**

  ```
  >>> draw.stroke_color = Color("black")
  ```

- `stroke_opacity` — **степень полупрозрачности линий в виде числа с плавающей точкой от 0 до 1:**

  ```
  >>> draw.stroke_opacity = 0.5
  ```

- `stroke_width` — **толщина линий в виде числа с плавающей точкой:**

  ```
  >>> draw.stroke_width = 2
  ```

- `fill_color` — **цвет заливки:**

  ```
  >>> draw.fill_color = Color("blue")
  ```

- `fill_opacity` — **степень полупрозрачности заливки в виде числа с плавающей точкой от 0 до 1:**

  ```
  >>> draw.fill_opacity = 0.2
  ```

Для собственно рисования класс `Drawing` **предоставляет следующие методы:**

- `point(<X>, <Y>)` — **рисует точку с заданными координатами:**

  ```
  >>> from wand.image import Image as WandImage
  >>> from wand.drawing import Drawing
  >>> from wand.color import Color
  >>> from wand.display import display
  >>> img = WandImage(width = 400, height = 300, background = Color("white"))
  >>> draw = Drawing()
  >>> draw.stroke_color = Color("black")
  ```

```
>>> draw.point(100, 200)
>>> draw.draw(img)
>>> display(img)
```

- `line(<Начальная точка>, <Конечная точка>)` — **рисует линию между точками с заданными координатами:**

```
>>> draw.stroke_color = Color("blue")
>>> draw.line((0, 0), (400, 300))
>>> draw.draw(img)
>>> display(img)
```

- `rectangle()` — **рисует прямоугольник, возможно, со скругленными углами, и выполняет его заливку. Формат метода:**

```
rectangle(left=<X1>, top=<Y1>, right=<X2>, bottom=<Y2> | width=<Ширина>,
height=<Высота>[, radius=<Радиус скругления> |
xradius=<Радиус скругления по горизонтали>,
yradius=<радиус скругления по вертикали>])
```

Параметры `left` и `top` задают горизонтальную и вертикальную координаты верхнего левого угла. Размеры прямоугольника можно задать либо координатами нижнего правого угла (параметры `right` и `bottom`), либо в виде ширины и высоты (параметры `width` и `height`). Можно задать либо радиус скругления углов и по горизонтали, и по вертикали (параметр `radius`), либо отдельно радиусы скругления по горизонтали и вертикали (параметры `xradius` и `yradius`). Если радиус скругления не задан, прямоугольник будет иметь острые углы. Пример:

```
>>> draw.stroke_color = Color("rgba(67, 82, 11, 0.7)")
>>> draw.fill_color = draw.stroke_color
>>> draw.rectangle(left = 100, top = 0, right = 150, bottom = 50)
>>> draw.rectangle(left = 200, top = 0, width = 50, height = 50, ↳
radius = 5)
>>> draw.rectangle(left = 300, top = 0, width = 50, height = 100, ↳
xradius = 5, yradius = 15)
>>> draw.draw(img)
>>> display(img)
```

- `polygon(<Список координат точек>)` — **рисует многоугольник. Единственный параметр представляет собой список, каждый элемент которого задает координаты одной точки и должен представлять собой кортеж из двух значений: горизонтальной и вертикальной координат. Указанные точки соединяются линиями, кроме того, проводится прямая линия между первой и последней точками, и полученный контур закрашивается. Пример:**

```
>>> draw.stroke_color = Color("rgb(0, 127, 127)")
>>> draw.fill_color = Color("rgb(127, 127, 0)")
>>> draw.polygon([(50, 50), (350, 50), (350, 250), (50, 250)])
>>> draw.draw(img)
>>> display(img)
```

- `polyline(<Список координат точек>)` — **то же самое, что `polygon()`, но прямая линия между первой и последней точкой не проводится, хотя контур все равно закрашивается;**

- `circle(<Центр>, <Периметр>)` — **рисует круг с заливкой. Первым параметром указывается кортеж с координатами центра окружности, вторым — кортеж с координатами любой точки, которая должна находиться на окружности и, таким образом, задает ее размеры. Пример:**

```
>>> draw.stroke_color = Color("black")
>>> draw.fill_color = Color("white")
>>> # Рисуем окружность радиусом 100 пикселов
>>> draw.circle((200, 150), (100, 150))
>>> draw.stroke_color = Color("white")
>>> draw.fill_color = Color("black")
>>> # Рисуем окружность радиусом 200 пикселов
>>> draw.circle((200, 150), (0, 150))
>>> draw.draw(img)
>>> display(img)
```

- `ellipse(<Центр>, <Радиус>[, rotation=<Начальный и конечный углы>])` — **рисует либо эллипс, либо его сектор с заливкой. Первым параметром указывается кортеж с координатами центра эллипса, вторым — кортеж с величинами радиусов по горизонтали и вертикали. Параметр** `rotation` **указывает начальный и конечный углы в градусах — если он задан, будет нарисован сегмент эллипса. Углы отсчитываются от горизонтальной координатной линии по часовой стрелке. Пример:**

```
>>> draw.stroke_color = Color("black")
>>> draw.fill_color = Color("white")
>>> draw.ellipse((200, 150), (100, 150))
>>> draw.stroke_color = Color("white")
>>> draw.fill_color = Color("black")
>>> draw.ellipse((200, 150), (200, 50), rotation = (20, 110))
>>> draw.draw(img)
>>> display(img)
```

- `arc(<Начальная точка>, <Конечная точка>, <Начальный и конечный углы>)` — **рисует дугу с заливкой. Первые два параметра должны представлять собой кортежи с двумя значениями — горизонтальной и вертикальной координат, третий — также кортеж со значениями начального и конечного углов. Углы отсчитываются от горизонтальной координатной линии по часовой стрелке. Пример:**

```
>>> draw.stroke_color = Color("green")
>>> draw.fill_color = Color("red")
>>> draw.arc((10, 10), (290, 290), (20, 110))
>>> draw.draw(img)
>>> display(img)
```

- `bezier(<Точки>)` — **рисует кривую Безье. Параметр должен представлять собой список, каждый элемент которого является кортежем с координатами одной из точек кривой. Точек должно быть, по меньшей мере, четыре: первая и последняя станут, соответственно, начальной и конечной, а промежуточные — контрольными точками. Пример:**

```
>>> draw.stroke_color = Color("red")
>>> draw.fill_color = Color("green")
```

```
>>> draw.bezier([(70, 167), (220, 109), (53, 390), (122, 14)])
>>> draw.draw(img)
>>> display(img)
```

Вы уже, наверно, заметили, что каждый набор выражений, рисующих какую-либо фигуру, завершен вызовом метода `draw()` класса `Drawing`. Дело в том, что описанные здесь методы, выполняющие рисование различных фигур, лишь говорят библиотеке `Wand`, что нужно нарисовать, но реально ничего не делают. Чтобы дать команду собственно выполнить рисование, следует вызвать метод `draw(<Изображение, на котором выполняется рисование>)`:

```
>>> draw.draw(img)
```

Для вывода на экран изображения, созданного средствами библиотеки `Wand`, предназначена функция `display(<Выводимое изображение>)`, определенная в модуле `wand.display` (она вам также должна быть знакома). Пример:

```
>>> from wand.display import display
>>> display(img)
```

Вывод выполняется в служебной утилите IMDisplay, входящей в состав поставки программного пакета ImageMagick (рис. 20.1).

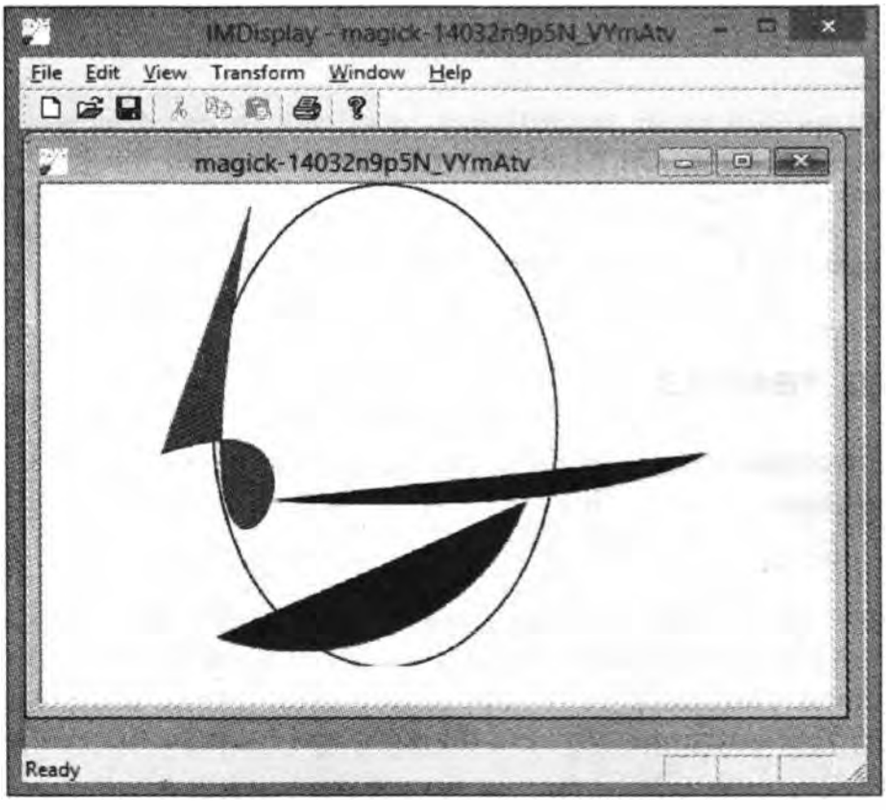


Рис. 20.1. Утилита IMDisplay

Для сохранения изображения в файле следует использовать метод `save()` класса `Image`. Его формат очень прост:

```
save(filename=<Имя файла>)
```

Давайте для примера создадим средствами `Wand` изображение, нарисуем на нем круг, сохраним в файл, после чего откроем и нарисуем рядом с ним второй круг, уже средствами `Pillow` (листинг 20.7).

Листинг 20.7. Совместное использование библиотек Wand и Pillow

```
from wand.image import Image as WandImage
from wand.color import Color
from wand.drawing import Drawing
from PIL import Image, ImageDraw
img = WandImage(width = 400, height = 300, background = Color("white"))
draw = Drawing()
draw.stroke_color = Color("red")
draw.fill_color = Color("white")
draw.circle((100, 100), (100, 0))
draw.draw(img)
img.save(filename = "tmp.bmp")
img = Image.open("tmp.bmp")
draw = ImageDraw.Draw(img)
draw.ellipse((200, 0, 400, 200), fill = "white", outline = "red")
img.show()
```

И, просмотрев картинку, получившуюся в результате выполнения приведенного кода, убедимся еще раз, что библиотека `Wand` рисует линии гораздо качественнее, чем `Pillow`.

> ***Примечание***
>
> **Вообще, программный пакет ImageMagick, оберткой которого является `Wand`, — исключительно мощное решение. Он позволяет рисовать сложные фигуры, накладывать на них всевозможные эффекты, обрабатывать растровые изображения и многое другое Полное описание `Wand` можно найти по интернет-адресу http://docs.wand-py.org/en/0.4.0/, а полное описание ImageMagick — на сайте http://www.imagemagick.org/.**

20.7. Вывод текста

Вывести текст на изображение позволяет метод `text()` из модуля `ImageDraw` библиотеки `Pillow`. 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 будет 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 — возникнет ошибка. 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`. **Файл со шрифтами с расширением pil можно загрузить с интернет-адреса http://effbot.org/media/downloads/pilfonts.zip. Пример:**

```
>>> font = ImageFont.load("pilfonts/helvO12.pil")
>>> draw.text((10, 40), "Hello", font=font, fill="blue")
>>> img.show()
```

Однако вывести русские буквы таким способом тоже нельзя;

- `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>[, size=<Размер>][, index=<Номер шрифта>])` — **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. Если размер не указан, загружается шрифт с размером 10 пунктов. Если не указан номер шрифта, хранящегося в шрифтовом файле, загружается шрифт с номером 0, т. е. первый попавшийся. Пример вывода надписи на русском языке:**

```
>>> txt = "Привет, мир!"
>>> font_file = r"C:\WINDOWS\Fonts\arial.ttf"
>>> font = ImageFont.truetype(font_file, size=24)
>>> draw.text((10, 80), txt, font=font, fill=(0, 0, 0))
>>> img.show()
```

Получить размеры прямоугольника, в который вписывается надпись, позволяет метод `textsize()` **класса** `ImageDraw`. **Формат метода:**

```
textsize(<Строка>, font=<Объект шрифта>)
```

Метод возвращает кортеж из двух элементов: `(<Ширина>, <Высота>)`. **Кроме того, можно воспользоваться методом** `getsize(<Строка>)` **объекта шрифта, возвращающим аналогичный результат. Пример:**

```
>>> txt = "Привет, мир!"
>>> font_file = r"C:\WINDOWS\Fonts\arial.ttf"
>>> font = ImageFont.truetype(font_file, size=24)
>>> draw.textsize(txt, font=font)
(143, 27)
>>> font.getsize(txt)
(143, 27)
```

Что касается библиотеки `Wand`, **то вывести текст на изображение позволяет также метод** `text()` **класса** `Drawing`. **Формат метода:**

```
text(<X>, <Y>, <Выводимый текст>)
```

Первые два параметра задают горизонтальную и вертикальную координаты точки, в которой начнется вывод текста.

Для указания параметров выводимого текста применяются следующие атрибуты класса `Drawing`:

- `font` — путь к файлу шрифта. Поддерживаются шрифты в форматах TrueType и OTF (OpenType Font);
- `font_size` — размер шрифта;
- `font_weight` — «жирность» шрифта в виде числа от 100 до 900. Обычный шрифт обозначается числом 400, полужирный — числом 700;
- `font_style` — стиль шрифта. Указывается в виде одного из элементов кортежа `STYLE_TYPES`, объявленного в модуле `wand.drawing`. Далее приведены индексы его элементов и стили шрифта, которые они задают:
  - `1` — обычный шрифт;
  - `2` — курсив;
  - `3` — наклонное начертание шрифта.

  Пример:

```
from wand.drawing import Drawing, STYLE_TYPES
draw = Drawing()
draw.font = r"c:\Windows\Fonts\arial.ttf"
draw.font_size = 24
draw.font_weight = 700
draw.font_style = STYLE_TYPES[2]
```

- `text_alignment` — выравнивание текста. Указывается в виде одного из элементов кортежа `TEXT_ALIGN_TYPES`, объявленного в модуле `wand.drawing`. Далее приведены индексы элементов кортежа и режимы выравнивания текста, которые они задают:
  - `1` — выравнивание по левому краю относительно точки, где будет выведен текст;
  - `2` — выравнивание по центру;
  - `3` — выравнивание по правому краю;
- `text_decoration` — дополнительное оформление текста. Должно представлять собой один из элементов кортежа `TEXT_DECORATION_TYPES`, объявленного в модуле `wand.drawing`. Далее приведены индексы его элементов и задаваемые ими режимы дополнительного оформления текста:
  - `1` — дополнительное оформление отсутствует;
  - `2` — подчеркивание;
  - `3` — надчеркивание;
  - `4` — зачеркивание.

  Пример:

```
from wand.drawing import TEXT_ALIGN_TYPES, TEXT_DECORATION_TYPES
draw.text_alignment = TEXT_ALIGN_TYPES[3]
draw.text_decoration = TEXT_DECORATION_TYPES[2]
```

Выведем текст на русском языке с помощью библиотеки `Wand` (листинг 20.8).

Листинг 20.8. Вывод текста на русском языке с помощью библиотеки Wand

```
from wand.image import Image as WandImage
from wand.color import Color
from wand.drawing import Drawing, STYLE_TYPES, TEXT_ALIGN_TYPES, ↳
TEXT_DECORATION_TYPES
from wand.display import display
img = WandImage(width = 400, height = 300, background = Color("white"))
draw = Drawing()
draw.stroke_color = Color("blue")
draw.fill_color = Color("yellow")
draw.font = r"c:\Windows\Fonts\verdana.ttf"
draw.font_size = 32
draw.font_weight = 700
draw.font_style = STYLE_TYPES[2]
draw.text_alignment = TEXT_ALIGN_TYPES[2]
draw.text_decoration = TEXT_DECORATION_TYPES[2]
draw.text(200, 150, "Привет, мир!")
draw.draw(img)
display(img)
```

Получить размеры прямоугольника, в который будет вписана надпись, в `Wand` **позволяет метод** `get_font_metrics()` **класса** `Drawing`. **Формат метода:**

```
textsize(<Изображение>, <Строка>)
```

Изображение должно представляться объектом класса `Image`. **Метод возвращает в качестве результата объект класса** `FontMetrics`, **объявленного в модуле** `wand.drawing`. **Из атрибутов этого класса нас интересуют следующие:**

- `text_width` — **ширина строки;**
- `text_height` — **высота строки;**
- `ascender` — **расстояние от базовой линии текста до верхней точки самого высокого символа строки. Всегда является положительным числом;**
- `descender` — **расстояние от базовой линии текста до нижней точки самого выступающего снизу символа. Всегда является отрицательным числом;**
- `maximum_horizontal_advance` — **максимальное расстояние между левой границей текущего и левой границей следующего символов;**
- `character_width` и `character_height` — **максимальные ширина и высота символов соответственно.**

Пример:

```
>>> fm = draw.get_font_metrics(img, "Привет, мир!")
>>> print(fm.text_width, fm.text_height)
216.0 39.0
```

20.8. Создание скриншотов

Библиотека `Pillow` в операционной системе Windows позволяет сделать *снимок экрана* (скриншот). Можно получить как полную копию экрана, так и копию определенной прямоугольной области. Для получения копии экрана предназначена функция `grab()` из модуля `ImageGrab`. Формат функции:

```
grab([<Координаты прямоугольной области>])
```

Координаты указываются в виде кортежа из четырех элементов — координат левого верхнего и правого нижнего углов прямоугольника. Если параметр не указан, возвращается полная копия экрана в виде объекта изображения в режиме `RGB`. Пример создания скриншотов приведен в листинге 20.9.

Листинг 20.9. 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 21

Взаимодействие с Интернетом

Интернет прочно вошел в нашу жизнь. Очень часто нам необходимо передать информацию на Web-сервер или, наоборот, получить с него какие-либо данные, например, 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 `http.client` и `urllib.request`.

21.1. Разбор URL-адреса

С помощью модуля `urllib.parse` можно манипулировать URL-адресом — например, разобрать его на составляющие или получить абсолютный URL-адрес, указав базовый и относительный адреса. URL-адрес (его также называют *интернет-адресом*) 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, хранящееся в атрибуте `netloc`, будет содержать номер порта. Кроме того, не ко всем атрибутам объекта можно получить доступ с помощью индексов. Пример кода, разбирающего URL-адрес, приведен в листинге 21.1.

Листинг 21.1. Разбор URL-адреса с помощью функции urlparse()

```
>>> from urllib.parse 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 таковой отсутствует в составе URL-адреса. По умолчанию это пустая строка. Примеры:**

```
>>> 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 (обычно хранящего предыдущую часть адреса), а не fragment. По умолчанию параметр имеет значение True. Примеры:

```
>>> 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')
```

Метод geturl() возвращает изначальный URL-адрес. Пример:

```
>>> url.geturl()
'http://wwwadmin.ru:80/test.php;st?var=5#metka'
```

Выполнить обратную операцию (собрать URL-адрес из отдельных значений) позволяет функция urlunparse(<Последовательность>) (листинг 21.2).

Листинг 21.2. Использование функции urlunparse()

```
>>> from urllib.parse import urlunparse
>>> t = ('http', 'wwwadmin.ru:80', '/test.php', '', 'var=5', 'metka')
>>> urlunparse(t)
'http://wwwadmin.ru:80/test.php?var=5#metka'
>>> l = ['http', 'wwwadmin.ru:80', '/test.php', '', 'var=5', 'metka']
>>> urlunparse(l)
'http://wwwadmin.ru:80/test.php?var=5#metka'
```

Вместо функции urlparse() можно воспользоваться функцией urlsplit(<URL-адрес>[, <Схема>[, <Разбор якоря>]]). Ее отличие от urlparse() проявляется в том, что она не выделяет из интернет-адреса параметры. 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 по названию атрибутов. Пример использования функции urlsplit() приведен в листинге 21.3.

Листинг 21.3. Разбор URL-адреса с помощью функции urlsplit()

```
>>> from urllib.parse 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 21.4).**

Листинг 21.4. Использование функции `urlunsplit()`

```
>>> from urllib.parse 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'
```

21.2. 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 пунктов в списке с множественным выбором:

```
str=%D1%F2%F0%EE%EA%E0&v=10&v=20
```

Разобрать строку запроса на составляющие и декодировать данные позволяют следующие функции из модуля `urllib.parse`:

- `parse_qs()` — **разбирает строку запроса и возвращает словарь с ключами, представляющими собой названия параметров, и значениями, которыми станут значения этих параметров. Формат функции:**

```
parse_qs(<Строка запроса>[, keep_blank_values=False][,
   strict_parsing=False][, encoding='utf-8'][, errors='replace'])
```

Если в параметре keep_blank_values указано значение True, то параметры, не имеющие значений внутри строки запроса, также будут добавлены в результат. По умолчанию пустые параметры игнорируются. Если в параметре strict_parsing указано значение True, то при наличии ошибки возбуждается исключение ValueError. По умолчанию ошибки игнорируются. Параметр encoding позволяет указать кодировку данных, а параметр errors — уровень обработки ошибок. Пример разбора строки запроса:

```
>>> from urllib.parse import parse_qs
>>> s = "str=%D1%F2%F0%EE%EA%E0&v=10&v=20&t="
>>> parse_qs(s, encoding="cp1251")
{'str': ['Строка'], 'v': ['10', '20']}
>>> parse_qs(s, keep_blank_values=True, encoding="cp1251")
{'str': ['Строка'], 't': [''], 'v': ['10', '20']}
```

- parse_qsl() — функция аналогична parse_qs(), только возвращает не словарь, а список кортежей из двух элементов: первый элемент «внутреннего» кортежа содержит название параметра, а второй элемент — его значение. Если строка запроса содержит несколько параметров с одинаковыми значениями, то они будут расположены в разных кортежах. Формат функции:

```
parse_qsl(<Строка запроса>[, keep_blank_values=False][,
   strict_parsing=False][, encoding='utf-8'][, errors='replace'])
```

Пример разбора строки запроса:

```
>>> from urllib.parse import parse_qsl
>>> s = "str=%D1%F2%F0%EE%EA%E0&v=10&v=20&t="
>>> parse_qsl(s, encoding="cp1251")
[('str', 'Строка'), ('v', '10'), ('v', '20')]
>>> parse_qsl(s, keep_blank_values=True, encoding="cp1251")
[('str', 'Строка'), ('v', '10'), ('v', '20'), ('t', '')]
```

Выполнить обратную операцию — преобразовать отдельные составляющие в строку запроса — позволяет функция urlencode(). Формат функции:

```
urlencode(<Объект>[, doseq=False][, safe=''][, encoding=None]
          [, errors=None])
```

В качестве первого параметра можно указать словарь с данными или последовательность, каждый элемент которой содержит кортеж из двух элементов: первый элемент такого кортежа станет параметром, а второй элемент — его значением. Параметры и значения автоматически обрабатываются с помощью функции quote_plus() из модуля urllib.parse. 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 21.5.

Листинг 21.5. Использование функции urlencode()

```
>>> from urllib.parse import urlencode
>>> params = {"str": "Строка 2", "var": 20}       # Словарь
>>> urlencode(params, encoding="cp1251")
'str=%D1%F2%F0%EE%EA%E0+2&var=20'
>>> params = [ ("str", "Строка 2"), ("var", 20) ] # Список
>>> urlencode(params, encoding="cp1251")
'str=%D1%F2%F0%EE%EA%E0+2&var=20'
```

Если необязательный параметр doseq в функции urlencode() имеет значение True, то во втором параметре кортежа можно указать последовательность из нескольких значений. В этом случае в строку запроса добавляются несколько параметров со значениями из этой последовательности. Значение параметра doseq по умолчанию — False. В качестве примера укажем список из двух элементов (листинг 21.6).

Листинг 21.6. Составление строки запроса из элементов последовательности

```
>>> params = [ ("var", [10, 20]) ]
>>> urlencode(params, doseq=False, encoding="cp1251")
'var=%5B10%2C+20%5D'
>>> urlencode(params, doseq=True, encoding="cp1251")
'var=10&var=20'
```

Последовательность также можно указать в качестве значения в словаре:

```
>>> params = { "var": [10, 20] }
>>> urlencode(params, doseq=True, encoding="cp1251")
'var=10&var=20'
```

Выполнить кодирование и декодирование отдельных элементов строки запроса позволяют следующие функции из модуля urllib.parse:

- quote() — заменяет все специальные символы последовательностями %nn. Цифры, английские буквы и символы подчеркивания (_), точки (.) и дефиса (-) не кодируются. Пробелы преобразуются в последовательность %20. Формат функции:

  ```
  quote(<Строка>[, safe='/'][, encoding=None][, errors=None])
  ```

  В параметре safe можно указать символы, которые преобразовывать нельзя, — по умолчанию параметр имеет значение /. Параметр encoding позволяет указать кодировку данных, а параметр errors — уровень обработки ошибок. Пример:

  ```
  >>> from urllib.parse import quote
  >>> quote("Строка", encoding="cp1251")  # Кодировка Windows-1251
  '%D1%F2%F0%EE%EA%E0'
  >>> quote("Строка", encoding="utf-8")   # Кодировка UTF-8
  '%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0'
  >>> quote("/~nik/"), quote("/~nik/", safe="")
  ('/%7Enik/', '%2F%7Enik%2F')
  >>> quote("/~nik/", safe="/~")
  '/~nik/'
  ```

- quote_plus() — функция аналогична quote(), но пробелы заменяются символом +, а не преобразуются в последовательность %20. Кроме того, по умолчанию символ / преобразуется в последовательность %2F. Формат функции:

  ```
  quote_plus(<Строка>[, safe=''][, encoding=None][, errors=None])
  ```

  Примеры:

  ```
  >>> from urllib.parse import quote, quote_plus
  >>> quote("Строка 2", encoding="cp1251")
  '%D1%F2%F0%EE%EA%E0%202'
  >>> quote_plus("Строка 2", encoding="cp1251")
  '%D1%F2%F0%EE%EA%E0+2'
  ```

```
>>> quote_plus("/~nik/")
'%2F%7Enik%2F'
>>> quote_plus("/~nik/", safe="/~")
'/~nik/'
```

- quote_from_bytes() — **функция аналогична** quote(), **но в качестве первого параметра принимает последовательность байтов, а не строку. Формат функции:**

```
quote_from_bytes(<Последовательность байтов>[, safe='/'])
```

Пример:

```
>>> from urllib.parse import quote_from_bytes
>>> quote_from_bytes(bytes("Строка 2", encoding="cp1251"))
'%D1%F2%F0%EE%EA%E0%202'
```

- unquote() — **заменяет последовательности** %nn **соответствующими символами. Символ** + **пробелом не заменяется. Формат функции:**

```
unquote(<Строка>[, encoding='utf-8'][, errors='replace'])
```

Примеры:

```
>>> from urllib.parse import unquote
>>> unquote("%D1%F2%F0%EE%EA%E0", encoding="cp1251")
'Строка'
>>> s = "%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0"
>>> unquote(s, encoding="utf-8")
'Строка'
>>> unquote('%D1%F2%F0%EE%EA%E0+2', encoding="cp1251")
'Строка+2'
```

- unquote_plus() — **функция аналогична** unquote(), **но дополнительно заменяет символ** + **пробелом. Формат функции:**

```
unquote_plus(<Строка>[, encoding='utf-8'][, errors='replace'])
```

Примеры:

```
>>> from urllib.parse import unquote_plus
>>> unquote_plus("%D1%F2%F0%EE%EA%E0+2", encoding="cp1251")
'Строка 2'
>>> unquote_plus("%D1%F2%F0%EE%EA%E0%202", encoding="cp1251")
'Строка 2'
```

- unquote_to_bytes() — **функция аналогична** unquote(), **но в качестве первого параметра принимает строку или последовательность байтов и возвращает последовательность байтов. Формат функции:**

```
unquote_to_bytes(<Строка или последовательность байтов>)
```

Примеры:

```
>>> from urllib.parse import unquote_to_bytes
>>> unquote_to_bytes("%D1%F2%F0%EE%EA%E0%202")
b'\xd1\xf2\xf0\xee\xea\xe0 2'
>>> unquote_to_bytes(b"%D1%F2%F0%EE%EA%E0%202")
b'\xd1\xf2\xf0\xee\xea\xe0 2'
```

```
>>> unquote_to_bytes("%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0")
b'\xd0\xa1\xd1\x82\xd1\x80\xd0\xbe\xd0\xba\xd0\xb0'
>>> str(_, "utf-8")
'Строка'
```

21.3. Преобразование относительного URL-адреса в абсолютный

Очень часто в коде Web-страниц указываются не абсолютные URL-адреса, а относительные. При относительном URL-адресе путь определяется с учетом местоположения страницы, на которой находится ссылка, или значения параметра `href` тега `<base>`. Преобразовать относительную ссылку в абсолютный URL-адрес позволяет функция `urljoin()` из модуля `urllib.parse`. Формат функции:

```
urljoin(<Базовый URL-адрес>, <Относительный или абсолютный URL-адрес>
        [, <Разбор якоря>])
```

Для примера рассмотрим преобразование различных относительных интернет-адресов (листинг 21.7).

Листинг 21.7. Варианты преобразования относительных интернет-адресов

```
>>> from urllib.parse 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 таком случае формируется некорректный интернет-адрес.

21.4. Разбор HTML-эквивалентов

В языке HTML некоторые символы являются специальными — например, знаки «меньше» (<) и «больше» (>), кавычки и др. Для отображения специальных символов служат так называемые *HTML-эквиваленты*. При этом знак «меньше» заменяется последовательностью `&lt;`, а знак «больше» — `&gt;`. 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. Примеры:**

```
>>> 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. Примеры:**

```
>>> 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. Примеры:**

```
>>> from xml.sax.saxutils import unescape
>>> s = '&amp;&lt;&gt;&quot; '
>>> unescape(s)
'&<>&quot; '
>>> unescape(s, { "&quot;": '"', " ": " " } )
'&<>" '
```

Для замены символов <, > и & HTML-эквивалентами также можно воспользоваться функцией `escape(<Строка>[, <Флаг>])` **из модуля** `html`**. Если во втором параметре указано значение** `False`**, двойные кавычки и апострофы не будут заменяться HTML-эквивалентами. А функция** `unescape(<Строка>)`**, объявленная в том же модуле и поддерживаемая, начиная с Python 3.4, выполняет обратную операцию — замену HTML-эквивалентов соответствующими им символами (листинг 21.8).**

Листинг 21.8. Замена спецсимволов HTML-эквивалентами

```
>>> import html
>>> html.escape("""&<>"' """)
'&amp;&lt;&gt;&quot;&#x27; '
>>> html.escape("""&<>"' """, False)
'&amp;&lt;&gt;"\' '
```

```
>>> html.unescape('&amp;&lt;&gt;&quot;&#x27; ')
'&<>"\' '
>>> html.unescape('&amp;&lt;&gt;"\' ')
'&<>"\' '
```

21.5. Обмен данными по протоколу HTTP

Модуль `http.client` 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`. Его конструктор имеет следующий формат:

```
HTTPConnection(<Домен>[, <Порт>[, timeout[, source_address]]])
```

В первом параметре указывается название домена без протокола. Во втором параметре задается номер порта — если параметр не указан, используется порт 80. Номер порта можно также задать после названия домена через двоеточие. Пример создания объекта соединения:

```
>>> from http.client import HTTPConnection
>>> con = HTTPConnection("test1.ru")
>>> con2 = HTTPConnection("test1.ru", 80)
>>> con3 = HTTPConnection("test1.ru:80")
```

После создания объекта соединения необходимо отправить запрос, возможно, с параметрами, вызвав метод `request()` класса `HTTPConnection`. Формат метода:

```
request(<Метод>, <Путь>[, body=None][, headers=<Заголовки>])
```

В первом параметре указывается метод передачи данных (`GET`, `POST` или `HEAD`). Второй параметр задает путь к запрашиваемому файлу или вызываемой программе, отсчитанный 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, файловый объект или последовательность. Четвертый параметр задает в виде словаря HTTP-заголовки, отправляемые на сервер.

Получить объект результата запроса позволяет метод `getresponse()`. Он возвращает результат выполненного запроса, представленный в виде объекта класса `HTTPResponse`. Из него мы сможем получить ответ сервера.

Прочитать ответ сервера (без заголовков) можно с помощью метода `read([<Количество байт>])` класса `HTTPResponse`. Если параметр не указан, метод `read()` возвращает все данные, а при наличии параметра — 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. Метод `read()` возвращает последовательность байтов, а не строку. Закрыть объект соединения позволяет метод `close()` класса `HTTPConnection`. Для примера отправим запрос методом `GET` и прочитаем результат (листинг 21.9).

Листинг 21.9. Отправка данных методом GET

```
>>> from http.client import HTTPConnection
>>> from urllib.parse import urlencode
>>> data = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
```

```
>>> 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 = HTTPConnection("test1.ru")
>>> con.request("GET", "/testrobots.php?%s" % data, headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print(result.read().decode("cp1251")) # Читаем данные
... 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 21.10.**

Листинг 21.10. Отправка данных методом POST

```
>>> from http.client import HTTPConnection
>>> from urllib.parse import urlencode
>>> data = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
>>> 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 = HTTPConnection("test1.ru")
>>> con.request("POST", "/testrobots.php", data, headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print(result.read().decode("cp1251"))
... Фрагмент опущен ...
>>> con.close()
```

Обратите внимание на заголовок `Content-Type`**. Если в нем указано значение** `application/x-www-form-urlencoded`**, это означает, что отправлены данные формы. При наличии этого заголовка некоторые программные платформы автоматически производят разбор строки запроса. Например, в PHP переданные данные будут доступны через глобальный массив** `$_POST`**. Если же заголовок не указать, то данные через массив** `$_POST` **доступны не будут.**

Класс `HTTPResponse`**, представляющий результат запроса, 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()
[('Date', 'Mon, 27 Apr 2015 13:33:21 GMT'), ('Server', 'Apache/2.2.4
 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'), ('X-Powered-By',
 'PHP/5.2.4'), ('Content-Length', '422'), ('Content-Type',
 'text/plain; charset=windows-1251')]
```

С помощью функции `dict()` **такой список можно преобразовать в словарь:**

```
>>> dict(result.getheaders())
{'Date': 'Mon, 27 Apr 2015 13:33:21 GMT', 'Content-Length': '422',
 'X-Powered-By': 'PHP/5.2.4', 'Content-Type': 'text/plain;
 charset=windows-1251', 'Server': 'Apache/2.2.4 (Win32)
 mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'}
```

- `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:**

```
>>> 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** `http.client.HTTPMessage`. **С его помощью можно получить дополнительную информацию о заголовках ответа сервера. Если объект передать функции** `print()`, **то мы получим все заголовки ответа сервера. Пример:**

```
>>> print(result.msg)
Date: Mon, 27 Apr 2015 13:33:21 GMT
Server: Apache/2.2.4 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4
X-Powered-By: PHP/5.2.4
Content-Length: 422
Content-Type: text/plain; charset=windows-1251
```

Рассмотрим основные методы и атрибуты объекта `http.client.HTTPMessage`:

- `as_string([unixfrom=False][, maxheaderlen=0])` — **возвращает все заголовки ответа сервера в виде строки:**

```
>>> result.msg.as_string()
'Date: Mon, 27 Apr 2015 13:33:21 GMT\nServer: Apache/2.2.4 (Win32)
 mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4\nX-Powered-By:
 PHP/5.2.4\nContent-Length: 422\nContent-Type: text/plain;
 charset=windows-1251\n\n'
```

- `items()` — **список всех заголовков ответа сервера:**

```
>>> result.msg.items()
[('Date', 'Mon, 27 Apr 2015 13:33:21 GMT'), ('Server', 'Apache/2.2.4
 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'), ('X-Powered-By',
 'PHP/5.2.4'), ('Content-Length', '422'), ('Content-Type',
 'text/plain; charset=windows-1251')]
```

- `keys()` — **список ключей в заголовках ответа сервера:**

```
>>> result.msg.keys()
['Date', 'Server', 'X-Powered-By', 'Content-Length', 'Content-Type']
```

- `values()` — **список значений в заголовках ответа сервера:**

```
>>> result.msg.values()
['Mon, 27 Apr 2015 13:33:21 GMT', 'Apache/2.2.4 (Win32) mod_ssl/2.2.4
 OpenSSL/0.9.8d PHP/5.2.4', 'PHP/5.2.4', '422', 'text/plain;
 charset=windows-1251']
```

- `get(<Заголовок>[, failobj=None])` — **возвращает значение указанного заголовка в виде строки. Если заголовок не найден, возвращается** `None` **или значение из второго параметра. Примеры:**

```
>>> result.msg.get("X-Powered-By")
'PHP/5.2.4'
>>> print(result.msg.get("X-Powered-By2"))
None
>>> result.msg.get("X-Powered-By2", failobj=10)
10
```

- `get_all(<Заголовок>[, failobj=None])` — **возвращает список всех значений указанного заголовка. Если заголовок не найден, возвращается** `None` **или значение из второго параметра. Пример:**

```
>>> result.msg.get_all("X-Powered-By")
['PHP/5.2.4']
```

- `get_content_type()` — **возвращает MIME-тип документа из заголовка** `Content-Type`:

```
>>> result.msg.get_content_type()
'text/plain'
```

- `get_content_maintype()` — **возвращает первую составляющую MIME-типа:**

```
>>> result.msg.get_content_maintype()
'text'
```

- `get_content_subtype()` — **возвращает вторую составляющую MIME-типа:**

```
>>> result.msg.get_content_subtype()
'plain'
```

- `get_content_charset([failobj=None])` — **позволяет получить кодировку из заголовка** `Content-Type`. **Если кодировка не найдена, возвращается** `None` **или значение из параметра** `failobj`. **Получим кодировку документа:**

```
>>> result.msg.get_content_charset()
'windows-1251'
```

Для примера отправим запрос методом `HEAD` **и выведем заголовки ответа сервера (листинг 21.11).**

Листинг 21.11. Отправка запроса методом HEAD

```
>>> from http.client import HTTPConnection
>>> 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 = HTTPConnection("test1.ru")
>>> con.request("HEAD", "/testrobots.php", headers=headers)
>>> result = con.getresponse()  # Создаем объект результата
>>> print(result.msg)
Date: Mon, 27 Apr 2015 13:39:54 GMT
Server: Apache/2.2.4 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4
X-Powered-By: PHP/5.2.4
Content-Type: text/plain; charset=windows-1251

>>> result.read() # Данные не передаются, только заголовки!
b''
>>> con.close()
```

Рассмотрим 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 Web-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.

21.6. Обмен данными с помощью модуля *urllib.request*

Модуль `urllib.request` 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>]
        [, cafile=None][, capath=None])
```

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. Метод возвращает объект класса `HTTPRequest`.

Этот класс поддерживает следующие методы и атрибуты:

- `read([<Количество байтов>])` — считывает данные. Если параметр не указан, 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:

```
>>> from urllib.request import urlopen
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> print(res.read(34).decode("cp1251"))
Название робота: Python-urllib/3.4
>>> print(res.read().decode("cp1251"))
... Фрагмент опущен ...
>>> res.read()
b''
```

- `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, будет считана одна строка, а не указанное количество байтов. Если количество байтов в строке больше, возвращается указанное количество байтов. Пример:

```
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> print(res.readline().decode("cp1251"))
Название робота: Python-urllib/3.4
```

- `readlines([<Количество байтов>])` — считывает весь результат в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Если параметр задан, считывается указанное количество байтов плюс фрагмент до конца строки. При достижении конца возвращается пустой список. Пример:

```
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> res.readlines(3)
[b'\xcd\xe0\xe7\xe2\xe0\xed\xe8\xe5 \xf0\xee\xe1\xee\xf2\xe0:
 Python-urllib/3.4\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 = urlopen("http://test1.ru/testrobots.php")
>>> for line in res: print(line)
```

- `close()` — закрывает объект результата;

- `geturl()` — возвращает интернет-адрес полученного документа. Так как все перенаправления автоматически обрабатываются, интернет-адрес полученного документа может не совпадать с 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 `http.client` (см. значение атрибута `msg` объекта результата). Пример:

```
>>> res = urlopen("http://test1.ru/testrobots.php")
>>> info = res.info()
>>> info.items()
[('Date', 'Mon, 27 Apr 2015 13:55:25 GMT'), ('Server', 'Apache/2.2.4
 (Win32) mod_ssl/2.2.4 OpenSSL/0.9.8d PHP/5.2.4'), ('X-Powered-By',
 'PHP/5.2.4'), ('Content-Length', '288'), ('Connection', 'close'),
 ('Content-Type', 'text/plain; charset=windows-1251')]
```

```
>>> info.get("Content-Type")
'text/plain; charset=windows-1251'
>>> info.get_content_type(), info.get_content_charset()
('text/plain', 'windows-1251')
>>> info.get_content_maintype(), info.get_content_subtype()
('text', 'plain')
```

- code — **содержит код возврата в виде числа;**
- msg — **содержит текстовый статус возврата:**

```
>>> res.code, res.msg
(200, 'OK')
```

Для примера выполним запросы методами GET и POST (листинг 21.12).

Листинг 21.12. Отправка данных методами GET и POST

```
>>> from urllib.request import urlopen
>>> from urllib.parse import urlencode
>>> data = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
>>> # Отправка данных методом GET
>>> url = "http://test1.ru/testrobots.php?" + data
>>> res = urlopen(url)
>>> print(res.read(34).decode("cp1251"))
Название робота: Python-urllib/3.4
>>> res.close()
>>> # Отправка данных методом POST
>>> url = " http://test1.ru/testrobots.php"
>>> res = urlopen(url, data.encode("cp1251"))
>>> print(res.read().decode("cp1251"))
... 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 интернет-адреса. Конструктор класса** Request **имеет следующий формат:**

```
Request(<URL-адрес>[, data=None][, headers={}][, origin_req_host=None]
                   [, unverifiable=False][, method=None])
```

В первом параметре указывается URL-адрес. Запрос выполняется методом GET**, если данные не указаны во втором параметре, или 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. Шестой параметр, поддержка которого появилась в Python 3.3, указывает метод передачи данных в виде строки — например:** "GET" **или** "HEAD". **За дополнительной информацией по этим параметрам обращайтесь к документации. В качестве примера выполним запросы методами** GET **и** POST **(листинг 21.13).**

Листинг 21.13. Использование класса Request

```
>>> from urllib.request import urlopen, Request
>>> from urllib.parse import urlencode
>>> 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 = urlencode({"color": "Красный", "var": 15}, encoding="cp1251")
>>> # Отправка данных методом GET
>>> url = "http://test1.ru/testrobots.php?" + data
>>> request = Request(url, headers=headers)
>>> res = urlopen(request)
>>> print(res.read(29).decode("cp1251"))
Название робота: MySpider/1.0
>>> res.close()
>>> # Отправка данных методом POST
>>> url = "http://test1.ru/testrobots.php"
>>> request = Request(url, data.encode("cp1251"), headers=headers)
>>> res = urlopen(request)
>>> print(res.read().decode("cp1251"))
... Фрагмент опущен ...
>>> res.close()
```

Как видно из результата, название нашего робота теперь `MySpider/1.0`.

21.7. Определение кодировки

Документы в Интернете могут быть представлены в различных кодировках. Чтобы документ был правильно обработан, необходимо знать его кодировку. Определить кодировку можно по заголовку `Content-Type` **в заголовках ответа Web-сервера:**

```
Content-Type: text/html; charset=utf-8
```

Кодировку Web-страницы можно также определить по значению параметра `content` **тега** `<meta>`**, расположенного в разделе** `HEAD`**:**

```
<meta http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
```

Однако часто встречается ситуация, когда кодировка в ответе сервера не совпадает с кодировкой, указанной в теге `<meta>`**, или же таковая вообще не указана. Определить кодировку документа в этом случае позволяет библиотека** `chardet`**. Установить ее можно из репозитория PyPI способом, описанным в** *главе 20*. **Открываем командную строку и набираем в ней команду:**

```
c:\python34\scripts\pip install chardet
```

Через некоторое время библиотека будет установлена.

Для проверки установки запускаем редактор IDLE и в окне Python Shell выполняем следующий код:

```
>>> import chardet
>>> chardet.__version__
'2.3.0'
```

Определить кодировку строки позволяет функция `detect(<Последовательность байтов>)` из модуля `chardet`. В качестве результата она возвращает словарь с двумя элементами. Ключ `encoding` содержит название кодировки, а ключ `confidence` — коэффициент точности определения в виде вещественного числа от `0` до `1`. Пример определения кодировки приведен в листинге 21.14.

Листинг 21.14. Пример определения кодировки

```
>>> import chardet
>>> chardet.detect(bytes("Строка", "cp1251"))
{'confidence': 0.99, 'encoding': 'windows-1251'}
>>> chardet.detect(bytes("Строка", "koi8-r"))
{'confidence': 0.99, 'encoding': 'KOI8-R'}
>>> chardet.detect(bytes("Строка", "utf-8"))
{'confidence': 0.99, 'encoding': 'utf-8'}
```

Если файл имеет большой размер, то вместо считывания его целиком 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 21.15.

Листинг 21.15. Пример использования класса `UniversalDetector`

```
from chardet.universaldetector import UniversalDetector
ud = UniversalDetector()        # Создаем объект
for line in open("file.txt", "rb"):
    ud.feed(line)               # Передаем текущую строку
    if ud.done: break           # Прерываем цикл, если done == True
ud.close()                      # Закрываем объект
print(ud.result)                # Выводим результат
input()
```

Как показали тесты одного из авторов, при использовании кодировки Windows-1251 файл просматривается полностью, и 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.

> ***Примечание***
>
> Полное описание библиотеки `chardet` можно найти по интернет-адресу **http://chardet.readthedocs.org/en/latest/**.

ГЛАВА 22

Сжатие данных

С архивными файлами любой пользователь сталкивается постоянно. В них распространяются дистрибутивы программ, документы, изображения, всевозможные служебные данные, в том числе и дополнительные библиотеки Python.

Неудивительно, что рассматриваемый нами язык программирования включает в состав стандартной библиотеки развитые средства для упаковки и распаковки как целых файлов, так и произвольных данных. Ими-то мы и займемся в этой, последней в книге главе.

22.1. Сжатие и распаковка по алгоритму GZIP

Для сжатия и распаковки файлов и данных по популярному алгоритму GZIP используются средства, определенные в модуле `gzip`.

Прежде всего, это функция `open()`. Формат ее вызова таков:

```
open(<Файл>[, mode='rb'][, compresslevel=9][, encoding=None] ↳
[, errors=None][, newline=None])
```

Первым параметром указывается либо путь к открываемому архивному файлу, либо представляющий его файловый объект. Второй параметр задает режим открытия файла (об этом подробно рассказано в *главе 16*): файл может быть открыт как в текстовом, так и в двоичном режиме. Третий параметр указывает степень сжатия архива в виде целого числа от 1 (минимальная степень, но максимальная скорость сжатия) до 9 (максимальная степень и минимальная скорость сжатия), также может быть задано значение 0 (отсутствие сжатия). Остальные параметры имеют смысл лишь при открытии файла в текстовом режиме и рассмотрены в *главе 16*.

Отметим, что по умолчанию файл открывается в двоичном режиме. Поэтому, если мы собираемся записывать в него строковые данные, нам следует явно указать текстовый режим открытия.

Функция `open()` возвращает объект класса `GzipFile`, представляющий открытый архивный файл. Этот класс поддерживает все атрибуты и методы, описанные в *главе 16*, за исключением метода `truncate()`. При этом все записываемые в такой файл данные будут автоматически архивироваться, а считываемые из него — распаковываться.

Для примера давайте создадим архивный GZIP-файл, сохраним в него строку, после чего откроем тот же файл и прочитаем его содержимое (листинг 22.1).

Листинг 22.1. Сохранение в архивном файле GZIP произвольных данных

```
>>> import gzip
>>> fn = "test.gz"
>>> s = "Это очень, очень, очень, очень большая строка"
>>> f = gzip.open(fn, mode = "wt", encoding = "utf-8")
>>> f.write(s)
>>> f.close()
>>> f = gzip.open(fn, mode = "rt", encoding = "utf-8")
>>> print(f.read())
Это очень, очень, очень, очень большая строка
>>> f.close()
```

Вместо функции `open()` можно непосредственно создать объект класса `GzipFile`. Его конструктор имеет следующий формат вызова:

```
GzipFile([filename=None][, mode='rb'][, compresslevel=9] ↳
[, fileobj=None][, mtime=None])
```

Можно задать либо путь к файлу в параметре `filename`, либо файловый объект в параметре `fileobj`. Если задано и то, и другое, имя файла будет включено в заголовок создаваемого GZIP-файла.

Параметр `mtime` указывает значение времени, которое будет добавлено в заголовок создаваемого архивного файла, как того требует формат GZIP. Это значение может быть получено вызовом функции `time()` из модуля `time` или сформировано иным образом. Если параметр не указан, будет использовано текущее время.

Отметим, что архивный файл в этом случае всегда открывается в двоичном режиме, и открыть его в текстовом режиме нельзя, даже если указать текстовый режим открытия.

Давайте сохраним в архиве графическое изображение, после чего распакуем его. Для создания архива используем функцию `open()`, а для распаковки — класс `GzipFile` (листинг 22.2).

Листинг 22.2. Сжатие и распаковка двоичного файла по алгоритму GZIP

```
>>> import gzip
>>> fn = "image.gz"
>>> f1 = open("image.jpg", "rb")
>>> f2 = gzip.open(fn, "wb")
>>> f2.write(f1.read())
>>> f2.close()
>>> f1.close()
>>> f1 = open("image_new.jpg", "wb")
>>> f2 = gzip.GzipFile(filename = fn)
>>> f1.write(f2.read())
>>> f1.close()
>>> f2.close()
```

В модуле `gzip` присутствуют также две функции, позволяющие сжимать и распаковывать произвольные двоичные данные без сохранения их в файл. Функция `compress(<Значение>[, compresslevel=9])` выполняет сжатие значения и возвращает получившийся в результате массив байтов типа `bytes`:

```
>>> import gzip
>>> s = b"This is a very, very, very, very big string"
>>> gzip.compress(s)
b'\x1f\x8b\x08\x00\x0f4>U\x02\xff\x0b\xc9\xc8,V\x00\xa2D\x85\xb2\xd4↳
\xa2J\x1d\x0cR!)3]\xa1\xb8\xa4(3/\x1d\x00\xbaZ)I+\x00\x00\x00'
```

А функция `decompress(<Значение>)` **выполняет распаковку сжатых данных и возвращает получившийся результат:**

```
>>> b = b'\x1f\x8b\x08\x00\x0f4>U\x02\xff\x0b\xc9\xc8,V\x00\xa2D\x85↳
\xb2\xd4\xa2J\x1d\x0cR!)3]\xa1\xb8\xa4(3/\x1d\x00\xbaZ)I+\x00\x00\x00'
>>> gzip.decompress(b)
b'This is a very, very, very, very big string'
```

22.2. Сжатие и распаковка по алгоритму BZIP2

Для сжатия и распаковки данных по алгоритму BZIP2 Python предусматривает модуль `bz2`.

Опять же, здесь присутствует функция `open()`**, позволяющая создать, записать или прочитать архивный файл:**

```
open(<Файл>[, mode='rb'][, compresslevel=9][, encoding=None] ↳
[, errors=None][, newline=None])
```

Она принимает те же параметры, что и одноименная функция из модуля `gzip`**. Есть лишь два исключения: для параметра** `compresslevel` **доступны значения от 1 до 9, а сама функция возвращает объект класса** `BZ2File`.

Попробуем заархивировать строку в формат BZIP2 и распаковать ее в первоначальный вид (листинг 22.3).

Листинг 22.3. Сохранение в архивном файле BZIP2 произвольных данных

```
>>> import bz2
>>> fn = "test.bz2"
>>> s = "Это очень, очень, очень, очень большая строка"
>>> f = bz2.open(fn, mode = "wt", encoding = "utf-8")
>>> f.write(s)
>>> f.close()
>>> f = bz2.open(fn, mode = "rt", encoding = "utf-8")
>>> print(f.read())
Это очень, очень, очень, очень большая строка
>>> f.close()
```

Также мы можем непосредственно создать объект класса `BZ2File` **и использовать его. Формат конструктора этого класса:**

```
BZ2File(<Файл>[, mode='rb'][, compresslevel=9])
```

Давайте поэкспериментируем с архивированием целых файлов и возьмем для примера файл документа Microsoft Word — это позволит нам оценить степень сжатия, обеспечиваемую алгоритмом BZIP2 (листинг 22.4).

Листинг 22.4. Сжатие и распаковка двоичного файла по алгоритму BZIP2

```
>>> import bz2
>>> fn = "doc.bz2"
>>> f1 = open("doc.doc", "rb")
>>> f2 = bz2.open(fn, "wb")
>>> f2.write(f1.read())
>>> f2.close()
>>> f1.close()
>>> f1 = open("doc_new.doc", "wb")
>>> f2 = bz2.BZ2File(filename = fn)
>>> f1.write(f2.read())
>>> f1.close()
>>> f2.close()
```

Однако здесь мы можем столкнуться с проблемой. При использовании подхода, представленного в листинге 22.4, в оперативную память загружается все содержимое сжимаемого или распаковываемого файла, и, если файл достаточно велик, потребление памяти может оказаться чрезмерным.

Выходом может оказаться сжатие или распаковка файла по частям. Для этого модуль `bz2` предлагает классы `BZ2Compressor` и `BZ2Decompressor`.

Итак, класс `BZ2Compressor` обеспечивает сжатие данных по частям. Его конструктор имеет формат `BZ2Compressor([compresslevel=9])`. Что касается его методов, то их всего два:

- `compress(<Данные>)` — сжимает переданные в качестве параметра данные и возвращает результат сжатия как массив байтов типа `bytes`;
- `flush()` — завершает процесс сжатия переданных ранее данных и возвращает результат их сжатия, оставшийся во внутренних буферах. Должен вызываться в любом случае после окончания сжатия.

Класс `BZ2Decompressor` позволяет распаковать сжатые ранее данные. Его конструктор вызывается без параметров, а его методы и атрибуты рассмотрены далее:

- `decompress(<Данные>)` — распаковывает переданные в качестве параметра сжатые данные и возвращает результат распаковки;
- `eof` — возвращает `True`, если в переданных сжатых данных присутствует сигнатура конца архива, т. е., если данные закончились;
- `unused_data` — возвращает «лишние» данные, присутствующие после сигнатуры конца архива.

Для практики запакуем и распакуем документ Microsoft Word, применив только что рассмотренные классы (листинг 22.5).

Листинг 22.5. Сжатие и распаковка двоичного файла по алгоритму BZIP2 по частям

```
import bz2
fn = "doc.bz2"
f1 = open("doc.doc", "rb")
f2 = open(fn, "wb")
comp = bz2.BZ2Compressor()
data = f1.read(1024)
```

```
while data:
    f2.write(comp.compress(data))
    data = f1.read(1024)
f2.write(comp.flush())
f2.close()
f1.close()
f1 = open("doc_new.doc", "wb")
f2 = open(fn, "rb")
decomp = bz2.BZ2Decompressor()
data = f2.read(1024)
while data:
    f1.write(decomp.decompress(data))
    data = f2.read(1024)
f1.close()
f2.close()
```

Как и в случае алгоритма GZIP, мы можем использовать аналогичные функции `compress(<Значение>[, compresslevel=9])` **и** `decompress(<Значение>)` **для сжатия и распаковки произвольных данных.**

22.3. Сжатие и распаковка по алгоритму LZMA

Поддержка формата LZMA появилась в Python 3.3. За нее отвечает модуль `lzma`.

Функция `open()`**, открывающая архивный файл для записи или чтения, имеет здесь такой формат:**

```
open(<Файл>[, mode='rb'][, format=None][, check=-1][, preset=None] ↳
[, filters=None][, encoding=None][, errors=None][, newline=None])
```

Параметр `format` **задает формат создаваемого или открываемого архивного файла. Здесь доступны следующие значения:**

- `lzma.FORMAT_AUTO` **— формат выбирается автоматически. Может быть задан только при открытии существующего файла, и в этом случае является значением по умолчанию;**
- `lzma.FORMAT_XZ` **— новая разновидность формата (расширение файлов — xz). Значение по умолчанию при создании архивного файла;**
- `lzma.FORMAT_ALONE` **— старая разновидность формата (расширение файлов — lzma);**
- `lzma.FORMAT_RAW` **— никакой формат не используется, и в файл записывается «сырой» набор данных.**

Параметр `check` **определяет тип проверки целостности архива — его имеет смысл задавать лишь при создании архивного файла. Доступные значения:**

- `lzma.CHECK_NONE` **— проверка на целостность не проводится. Значение по умолчанию и единственное доступное значение для форматов** `lzma.FORMAT_ALONE` **и** `lzma.FORMAT_RAW`**;**
- `lzma.CHECK_CRC32` **— 32-разрядная циклическая контрольная сумма;**
- `lzma.CHECK_CRC64` **— 64-разрядная циклическая контрольная сумма. Значение по умолчанию для формата** `lzma.FORMAT_XZ`**;**
- `lzma.CHECK_SHA256` **— хеширование по алгоритму SHA256.**

Параметр `preset` указывает используемый при сжатии или распаковке набор параметров архиватора, фактически — степень сжатия. Доступны числовые значения от 0 (минимальное сжатие и высокая производительность) до 9 (максимальное сжатие и низкая производительность). Этот параметр имеет смысл задавать лишь при создании архивного файла. Значение по умолчанию — `lzma.PRESET_DEFAULT`, соответствующее числу 6.

Параметр `filters` задает набор дополнительных фильтров, используемых при архивировании и распаковке. Отметим, что для формата `lzma.FORMAT_RAW` фильтр требуется указать обязательно. За описанием процесса создания фильтров обращайтесь к документации по Python.

Функция `open()` возвращает объект класса `LZMAFile`, представляющий созданный или открытый архивный файл.

Если же мы захотим непосредственно создать объект этого класса, то вызовем его конструктор согласно следующему формату:

```
LZMAFile(filename=<Файл>[, mode='rb'][, format=None][, check=-1] ↳
[, preset=None][, filters=None])
```

Представленный в листинге 22.6 код архивирует строку, сохраняет ее в архив, а потом распаковывает.

Листинг 22.6. Сохранение строки в архиве LZMA

```
>>> import lzma
>>> fn = "test.xz"
>>> s = "Это очень, очень, очень, очень большая строка"
>>> f = lzma.open(fn, mode = "wt", encoding = "utf-8")
>>> f.write(s)
>>> f.close()
>>> f = lzma.LZMAFile(filename = fn)
>>> str(f.read(), encoding = "utf-8")
'Это очень, очень, очень, очень большая строка'
>>> f.close()
```

Для сжатия и распаковки данных по частям мы применим классы `LZMACompressor` и `LZMADecompressor`. Формат вызова конструктора первого класса таков:

```
LZMACompressor([format=lzma.FORMAT_XZ][, check=-1][, preset=None] ↳
[, filters=None])
```

Поддерживаются методы `compress(<Данные>)` и `flush()`, знакомые нам по классу `BZ2Compressor`.

Конструктор класса `LZMADecompressor` имеет следующий формат вызова:

```
LZMADecompressor([format=lzma.FORMAT_AUTO][, memlimit=None] ↳
[, filters=None])
```

Параметр `memlimit` устанавливает максимальный размер памяти в байтах, который может быть использован архиватором. По умолчанию этот размер не ограничен. Отметим, что в случае задания параметра `memlimit`, если архиватор не сможет уложиться в отведенный ему

объем памяти, будет возбуждено исключение `LZMAError`, класс которого объявлен в модуле `lzma`.

Этим классом поддерживаются знакомые нам метод `decompress(<Данные>)` и атрибуты `eof` и `unused_data`.

В качестве примера заархивируем и тут же распакуем документ Microsoft Word (листинг 22.7). Заодно поэкспериментируем с заданием формата архива и степени сжатия.

Листинг 22.7. Сжатие и распаковка двоичного файла по алгоритму LZMA по частям

```
import lzma
fn = "doc.lzma"
f1 = open("doc.doc", "rb")
f2 = open(fn, "wb")
comp = lzma.LZMACompressor(format = lzma.FORMAT_ALONE, preset = 9)
data = f1.read(1024)
while data:
    f2.write(comp.compress(data))
    data = f1.read(1024)
f2.write(comp.flush())
f2.close()
f1.close()
f1 = open("doc_new.doc", "wb")
f2 = open(fn, "rb")
decomp = lzma.LZMADecompressor()
data = f2.read(1024)
while data:
    f1.write(decomp.decompress(data))
    data = f2.read(1024)
f1.close()
f2.close()
```

В модуле `lzma` определены функции `compress()` и `decompress()` для сжатия и распаковки произвольных данных. Только форматы их вызова несколько другие:

```
compress(<Данные>[, format=lzma.FORMAT_XZ][, check=-1][, preset=None] ↳
[, filters=None])
```

и

```
decompress(<Данные>[, format=lzma.FORMAT_AUTO][, memlimit=None] ↳
[, filters=None])
```

Если в процессе обработки архива LZMA возникнет ошибка, будет возбуждено исключение класса `LZMAError`. Этот класс объявлен в модуле `lzma`. Пример обработки таких исключений:

```
import lzma
try:
    f = lzma.open("test.xz")
except lzma.LZMAError:
    print("Что-то пошло не так...")
```

22.4. Работа с архивами ZIP

Рассмотренные нами ранее форматы архивов GZIP, BZIP2 и LZMA позволяют хранить лишь один файл. В отличие от них, популярнейший формат ZIP может хранить сколько угодно файлов, однако произвольные данные мы сохранить в нем не сможем.

Поддержка формата ZIP реализована в модуле `zipfile`. В первую очередь, нам понадобится класс `ZipFile`, представляющий архив и выполняющий все манипуляции с ним. Конструктор этого класса вызывается следующим образом:

```
ZipFile(<Файл>[, mode='r'][, compression=ZIP_STORED][, allowZip64=True])
```

Первый параметр задает путь к архивному файлу. Вместо него можно задать файловый объект.

Параметр `mode` определяет режим открытия файла. Мы можем указать строковые значения:

- `r` — открыть существующий файл для чтения. Если файл не существует, будет возбуждено исключение;
- `w` — открыть существующий файл для записи. Если файл не существует, будет возбуждено исключение. Если файл существует, он будет перезаписан;
- `a` — открыть существующий файл для записи. Если файл не существует, он будет создан. Если файл существует, новое содержимое будет добавлено в его конец, а старое содержимое сохранится.

Параметр `compression` указывает алгоритм сжатия, который будет применен для архивирования содержимого файла. Доступны значения:

- `zipfile.ZIP_STORED` — сжатие как таковое отсутствует. Значение по умолчанию;
- `zipfile.ZIP_DEFLATED` — алгоритм сжатия Deflate, стандартно применяемый в архивах ZIP;
- `zipfile.ZIP_BZIP2` — алгоритм, применяемый в архивах BZIP2;
- `zipfile.ZIP_LZMA` — алгоритм, применяемый в архивах LZMA.

Если присвоить параметру `allowZip64` значение `False`, будет невозможно создать архив размером более 2 Гбайт. Этот параметр предусмотрен для совместимости со старыми версиями архиваторов ZIP.

Архивный файл всегда открывается в двоичном режиме. Пример:

```
>>> import zipfile
>>> f = zipfile.ZipFile("test.zip", mode = "a", ↳
compression = zipfile.ZIP_DEFLATED)
```

Теперь рассмотрим методы и атрибуты класса `ZipFile`.

- `write(<Имя файла>[, arcname=<Имя, которое он будет иметь в архиве>][, compress_type=None])` — добавляет в архив файл с указанным именем. Параметр `arcname` задает имя, которое файл примет, будучи помещенным в архив, — если он не указан, файл сохранит свое оригинальное имя. Параметр `compress_type` задает алгоритм сжатия — если он не указан, будет использован алгоритм, заданный при открытии самого архива. Примеры:

  ```
  >>> # Добавляем в архив файл doc.doc
  >>> f.write("doc.doc")
  ```

```
>>> # Добавляем в архив файл doc2.doc под именем newdoc.doc
>>> f.write("doc2.doc", arcname = "newdoc.doc")
```

- `writestr(<Имя файла>, <Данные>[, compress_type=None])` — **добавляет в архив произвольные данные в виде файла с указанным именем**:

```
>>> # Считываем содержимое файла text.txt
>>> f2 = open("text.txt", mode = "r")
>>> s = f2.read()
>>> # Добавляем прочитанные данные в архив под именем textual.txt
>>> f.writestr("textual.txt", s)
>>> f2.close()
```

- `close()` — **закрывает архивный файл**:

```
>>> f.close()
```

- `getinfo(<Имя файла>)` — **возвращает сведения о хранящемся в архиве файле с указанным именем. Эти сведения представляются в виде объекта класса** `ZipInfo`, **объявленного в модуле** `zipfile` **и поддерживающего следующие полезные нам атрибуты**:
  - `filename` — **имя файла**;
  - `file_size` — **размер изначального (несжатого) файла**;
  - `date_time` — **дата и время последнего изменения файла. Представляется в виде кортежа из шести элементов: года, номера месяца (от 1 до 12), числа (от 1 до 31), часов (от 0 до 23), минут (от 0 до 59) и секунд (от 0 до 59)**;
  - `compress_size` — **размер файла в сжатом виде**;
  - `compress_type` — **алгоритм сжатия**;
  - `CRC` — **32-разрядная контрольная сумма**;
  - `comment` — **комментарий к файлу**;
  - `create_system` — **операционная система, в которой был создан архив**;
  - `create_version` — **версия архиватора, в которой был создан архив**;
  - `extract_version` — **версия архиватора, необходимая для распаковки архива.**

  Если файл с заданным именем отсутствует в архиве, возбуждается исключение `KeyError`. **Пример**:

```
>>> f = zipfile.ZipFile("test.zip", mode = "r", ↳
compression = zipfile.ZIP_DEFLATED)
>>> gf = f.getinfo("doc.doc")
>>> gf.filename, gf.file_size, gf.compress_size
('doc.doc', 242688, 63242)
>>> gf.date_time
(2015, 4, 27, 14, 51, 4)
```

- `infolist()` — **возвращает сведения обо всех содержащихся в архиве файлах в виде списка объектов класса** `ZipInfo`:

```
>>> for i in f.infolist(): print(i.filename, end = " ")
doc.doc newdoc.doc textual.txt
```

- `namelist()` — **возвращает список с именами хранящихся в архиве файлов:**

```
>>> f.namelist()
['doc.doc', 'newdoc.doc', 'textual.txt']
```

- `extract(<Файл>[, path=None][, pwd=None])` — **распаковывает из архива указанный файл, который может быть задан в виде имени или объекта класса** `ZipInfo`. **Параметр** `path` **сообщает архиватору путь, по которому должен быть распакован файл, — если он не указан, файл будет сохранен там же, где находится сам архив. Параметр** `pwd` **задает пароль для распаковки файла, если таковой требуется. В качестве результата возвращается полный путь к распакованному файлу. Примеры:**

```
>>> # Распаковываем файл doc.doc, сведения о котором хранятся
>>> # в переменной gf
>>> f.extract(gf)
'C:\\Python34\\doc.doc'
>>> # Распаковываем файл newdoc.doc в папку c:\work
>>> f.extract("newdoc.doc", path = r'c:\work')
'c:\\work\\newdoc.doc'
```

- `extractall([path=None][, members=None][, pwd=None])` — **распаковывает сразу несколько или даже все файлы из архива. Параметр** `members` **задает список имен файлов, которые должны быть распакованы, — если он не указан, будут распакованы все файлы. Назначение параметров** `path` **и** `pwd` **рассмотрено в описании метода** `extract()`. **Примеры:**

```
>>> # Распаковываем все файлы
>>> f.extractall()
>>> # Распаковываем лишь файлы doc.doc и newdoc.doc в папку c:\work
>>> f.extractall(path = r'c:\work', ↳
members = ['doc.doc', 'newdoc.doc'])
```

- `open(<Файл>[, pwd=None])` — **открывает хранящийся в архиве файл для чтения. Файл может быть задан либо в виде имени, либо как объект класса** `ZipInfo`. **Результатом, возвращенным методом, станет объект класса** `ZipExtFile`, **поддерживающий методы** `read()`, `readline()`, `readlines()`, **знакомые нам по** *главе 16*, **а также итерационный протокол.**

  Вот пример открытия файла textual.txt, хранящегося в архиве, и записи его содержимого в файл newtext.txt:

```
>>> d = f.open("textual.txt")
>>> f2 = open("newtext.txt", mode = "wb")
>>> f2.write(d.read())
>>> f2.close()
```

- `read(<Имя файла>, pwd=None)` — **возвращает содержимое хранящегося в архиве файла с указанным именем. Для примера перепишем приведенный ранее код:**

```
>>> d = f.read("textual.txt")
>>> f2 = open("newtext.txt", mode = "wb")
>>> f2.write(d)
>>> f2.close()
```

- `setpassword(<Пароль>)` — **задает пароль по умолчанию для распаковки файлов;**
- `testzip()` — **выполняет проверку целостности архива. Возвращает** `None`, **если архив не поврежден, или имя первого встретившегося ему сбойного файла;**

- comment — позволяет получить или задать комментарий к архиву. В качестве комментария может выступать строка длиной не более 65535 байтов. Более длинные строки автоматически сокращаются при закрытии архива.

В модуле zipfile также определена функция is_zipfile(<Имя файла>). Она возвращает True, если файл с переданным ей именем является архивом ZIP, и False — в противном случае. Примеры:

```
>>> zipfile.is_zipfile("test.zip")
True
>>> zipfile.is_zipfile("doc.doc")
False
```

При обработке файлов ZIP могут возникнуть исключения следующих классов (все они объявлены в модуле zipfile):

- BadZipFile — либо архив поврежден, либо это вообще не ZIP-архив;
- LargeZipFile — слишком большой архив ZIP. Обычно возникает, когда архив создается вызовом конструктора класса ZipFile с параметром allowZip64, имеющим значение False, и размер получившегося архива в процессе работы становится больше 2 Гбайт.

22.5. Работа с архивами TAR

В отличие от файлов ZIP, архивы формата TAR не используют сжатие. Обычно такие архивы дополнительно сжимаются другим архиватором: GZIP, BZIP2 или LZMA. Однако — и в этом состоит их другое отличие от ZIP-файлов — они могут включать в свой состав не только файлы, но и папки, хранящие как файлы, так и другие папки.

Python поддерживает работу с архивами TAR — как несжатыми, так и сжатыми. Для этого предназначается модуль tarfile.

Проще всего открыть архив вызовом функции open(). Она принимает очень много параметров:

```
open([name=None][, mode='r'][, fileobj=None][, compresslevel=9] ↪
[, format=DEFAULT_FORMAT] [, dereference=False] [, tarinfo=TarInfo] ↪
[, ignore_zeros=False][, encoding=ENCODING][, errors='surrogateescape'] ↪
[, pax_headers=None][, debug=0][, errorlevel=0])
```

Открываемый файл может быть задан либо в виде имени (параметр name), либо в виде файлового объекта (параметр fileobj).

Параметр mode задает режим открытия и алгоритм сжатия файла. Его значение указывается в виде строки в формате:

```
<Режим открытия>[[:]<Алгоритм сжатия>]
```

Доступны три режима открытия файла:

- r — открыть существующий файл для чтения. Если файл не существует, будет возбуждено исключение;
- w — открыть существующий файл для записи. Если файл не существует, будет возбуждено исключение. Если файл существует, он будет перезаписан;

- `a` — открыть существующий файл для записи. Если файл не существует, он будет создан. Если файл существует, новое содержимое будет добавлено в его конец, а старое содержимое сохранится. В этом режиме можно открывать лишь несжатые файлы.

Алгоритмов сжатия можно указать также три: `gz` (GZIP), `bz2` (BZIP2) и `xz` (LZMA). Если же алгоритм не указан, то при открытии файла на чтение он будет определен автоматически, а при открытии на запись или добавление сжатие использовано не будет. Примеры указания режимов открытия и сжатия файлов: `r` (чтение, алгоритм сжатия определяется автоматически), `r:bz2` (чтение, сжатие BZIP2), `w:xz` (запись, сжатие LZMA).

Параметр `compresslevel` задает степень сжатия и доступен лишь при указании какого-либо алгоритма сжатия.

Параметр `format` указывает формат архива TAR. Для него доступны четыре значения:

- `tarfile.USTAR_FORMAT` — формат POSIX.1-1988 (ustar);
- `tarfile.GNU_FORMAT` — формат GNU;
- `tarfile.PAX_FORMAT` — формат POSIX.1-2001 (pax);
- `tarfile.DEFAULT_FORMAT` — формат по умолчанию, на данный момент — GNU (`tarfile.GNU_FORMAT`).

Если параметр `dereference` имеет значение `True`, при добавлении в архив символической или жесткой ссылки на самом деле будет добавлен файл или папка, на которую указывает эта ссылка. Если же задать для него значение `False` (это, кстати, значение по умолчанию), то в архив будет добавлен сам файл ссылки.

Остальные параметры используются в особых случаях и подробно описаны в документации по модулю `tarfile`, поставляемой в составе Python.

Функция `open()` возвращает в качестве результата объект класса `TarFile`, представляющий созданный или открытый архивный файл.

Открыть или создать файл мы можем, непосредственно создав объект этого класса. Его конструктор вызывается так же, как и функция `open()`:

```
TarFile([name=None][, mode='r'][, fileobj=None][, compresslevel=9] ↳
[, format=DEFAULT_FORMAT] [, dereference=False] [, tarinfo=TarInfo] ↳
[, ignore_zeros=False][, encoding=ENCODING][, errors='surrogateescape'] ↳
[, pax_headers=None][, debug=0][, errorlevel=0])
```

Однако следует иметь в виду, что в этом случае параметр `mode` может указывать лишь режим открытия файла: `r`, `w` или `a`. Задать алгоритм сжатия в нем нельзя. Пример:

```
>>> import tarfile
>>> # Поскольку мы не можем создать сжатый файл TAR,
>>> # сначала создадим несжатый...
>>> f = tarfile.TarFile(name = "test.tar.gz", mode = "a")
>>> # ...сразу же закроем его...
>>> f.close()
>>> # ...а потом откроем снова, указав алгоритм сжатия GZIP
>>> f = tarfile.open(name = "test.tar.gz", mode = "w:gz")
```

Методы и атрибуты, поддерживаемые классом `TarFile` и предназначенные для работы с содержимым архива, рассмотрены далее:

- add(<Имя элемента>[, arcname=<Имя, которое он будет иметь в архиве>][, recursive=True][, exclude=None]) — **добавляет в архив элемент (файл, папку, символическую или жесткую ссылку) с указанным именем. Параметр** arcname **задает имя, которое элемент примет, будучи помещенным в архив. По умолчанию это изначальное имя элемента. Если параметру** recursive **присвоить значение** False**, папки будут добавляться в архив без содержащихся в них папок и файлов. По умолчанию они добавляются вместе с содержимым.**

  Параметру exclude **можно присвоить функцию, которая будет принимать один параметр — имя очередного добавляемого в архив элемента — и возвращать логическую величину. Если она равна** True**, элемент не будет добавлен в архив, если** False **— то будет. Причем этот элемент может как добавляться непосредственно в вызове метода** add()**, так и находиться в добавляемой папке. Примеры:**

```
>>> # Добавляем в архив файл doc.doc
>>> f.add("doc.doc")
>>> # Добавляем в архив файл doc2.doc под именем newdoc.doc
>>> f.add("doc2.doc", arcname = "newdoc.doc")
>>> # Добавляем в архив папку test с содержимым
>>> f.add("test")
>>> # Добавляем в архив папку test2 без содержимого
>>> f.add("test2", recursive = False)
>>> # Добавляем в архив папку test3, исключив все временные файлы,
>>> # что могут в ней находиться
>>> def except_tmp(filename):
        return filename.find(".tmp") != -1
>>> f.add("test3", exclude = except_tmp)
```

- close() — **закрывает архивный файл:**

```
>>> f.close()
```

- getmember(<Имя элемента>) — **возвращает объект класса** TarInfo**, представляющий хранящийся в архиве элемент с указанным именем. Класс** TarInfo **поддерживает следующие полезные нам атрибуты и методы:**
  - name — **имя элемента (файла, папки, жесткой или символической ссылки);**
  - size — **размер элемента в байтах;**
  - mtime — **время последнего изменения элемента;**
  - mode — **права доступа к элементу;**
  - linkname — **путь, на который указывает жесткая или символическая ссылка. Доступно только для элементов-ссылок;**
  - isfile() **и** isreg() — **возвращают** True**, если элемент является файлом;**
  - isdir() — **возвращает** True**, если элемент является папкой;**
  - issym() — **возвращает** True**, если элемент является символической ссылкой;**
  - islnk() — **возвращает** True**, если элемент является жесткой ссылкой.**

  Если элемент с заданным именем отсутствует в архиве, возбуждается исключение KeyError.

Примеры:

```
>>> f = tarfile.open(name = "test.tar.gz")
>>> # Получаем сведения о файле doc.doc
>>> ti = f.getmember("doc.doc")
>>> ti.name, ti.size, ti.mtime, ti.isfile(), ti.isdir()
('doc.doc', 242688, 1430135464, True, False)
>>> # Получаем сведения о папке test
>>> ti = f.getmember("test")
>>> ti.name, ti.size, ti.mtime, ti.isfile(), ti.isdir()
('test', 0, 1430223812, False, True)
```

- `getmembers()` — **возвращает сведения обо всех содержащихся в архиве элементах в виде списка объектов класса** `TarInfo`:

```
>>> for i in f.getmembers(): print(i.name, end = " ")
doc.doc newdoc.doc test test/test2 test/test2/text.txt test/text.txt
```

Отметим, что возвращаются, в том числе, все файлы и папки, хранящиеся в присутствующих в архиве папках;

- `getnames()` — **возвращает список с именами хранящихся в архиве элементов:**

```
>>> f.getnames()
['doc.doc', 'newdoc.doc', 'test', 'test/test2', 'test/test2/text.txt',
'test/text.txt']
```

- `next()` — **возвращает следующий элемент из находящихся в архиве. Если элементов больше нет, возвращается** `None`;

- `extract(<Элемент>[, path=""][, set_attrs=True])` — **распаковывает указанный элемент, который может быть задан в виде имени или объекта класса** `TarInfo`. **Параметр** `path` **сообщает архиватору путь, по которому должен быть распакован элемент, — если он не указан, элемент будет сохранен там же, где находится сам архив. Если задать для параметра** `set_attrs` **значение** `False`, **время последнего изменения элемента и права доступа для распаковываемого элемента задаст сама операционная система, если же его значение —** `True` **(как по умолчанию), эти сведения будут взяты из архива. Примеры:**

```
>>> # Распаковываем папку test, сведения о которой хранятся
>>> # в переменной ti
>>> f.extract(ti)
>>> # Распаковываем файл doc.doc в папку c:\work
>>> f.extract("doc.doc", path = r'c:\work')
```

- `extractall([path="."][, members=None])` — **распаковывает сразу несколько или даже все элементы из архива. Параметр** `members` **задает список элементов, представленных объектами класса** `TarFile`, **которые должны быть распакованы, — если он не указан, будут распакованы все элементы. Назначение параметра** `path` **рассмотрено в описании метода** `extract()`. **Примеры:**

```
>>> # Распаковываем все файлы
>>> f.extractall()
>>> # Распаковываем лишь файлы doc.doc и newdoc.doc в папку c:\work
>>> l = [f.getmember("doc.doc"), f.getmember("newdoc.doc")]
>>> f.extractall(path = r'c:\work', members = l)
```

- `extractfile(<Элемент>)` — открывает для чтения хранящийся в архиве элемент-файл, заданный именем или объектом класса `TarFile`. В качестве результата возвращается объект класса `ExFileObject`, поддерживающий методы `read()`, `readline()`, `readlines()`, знакомые нам по *главе 16*, и итерационный протокол.

  Пример открытия файла doc.doc, хранящегося в архиве, и записи его содержимого в файл doc2.doc:

```
>>> d = f.extractfile("doc.doc")
>>> f2 = open("doc2.doc", mode = "wb")
>>> f2.write(d.read())
>>> f2.close()
```

В модуле `tarfile` присутствует функция `is_tarfile(<Имя файла>)`, возвращающая `True`, если файл с переданным ей именем является архивом TAR. Примеры:

```
>>> tarfile.is_tarfile("test.tar.gz")
True
>>> tarfile.is_tarfile("doc2.doc")
False
```

При обработке TAR-архивов могут возбуждаться следующие исключения (все они объявлены в модуле `tarfile`):

- `TarError` — базовый класс для всех последующих классов исключений;
- `ReadError` — либо архив поврежден, либо это вообще не архив TAR;
- `CompressionError` — заданный алгоритм сжатия не поддерживается, или данные по какой-то причине не могут быть сжаты;
- `StreamError` — ошибка обмена данными с файлом архива;
- `ExtractError` — при распаковке данных возникла некритическая ошибка.

> ***Примечание***
>
> Python также поддерживает сжатие и распаковку файлов в формате ZLIB, похожем на формат GZIP. Инструменты, используемые для этого, описаны в документации.

Заключение

Вот и закончилось наше путешествие в мир Python. Материал книги описывает лишь базовые возможности этого универсального языка программирования. А мы сейчас рассмотрим, где найти дополнительную информацию и продолжить изучение языка Python.

Первым и самым важным источником информации является сайт **https://www.python.org/**. Там вы найдете последнюю версию интерпретатора, новости и ссылки на другие тематические интернет-ресурсы.

На сайте **https://docs.python.org/** опубликована актуальная документация по Python. Язык постоянно совершенствуется, появляются новые функции, изменяются параметры, добавляются модули и т. д. Регулярно посещайте этот сайт — и вы получите самую последнюю информацию. Кроме того, документация в формате CHM поставляется в составе дистрибутива Python и присутствует в исходном коде его модулей, — как отобразить ее, мы рассматривали в *разд. 1.8*.

В состав стандартной библиотеки Python входит большое количество модулей, позволяющих решить наиболее часто встречающиеся задачи. Однако этим возможности Python не исчерпываются. Мир Python включает множество самых разнообразных модулей и целых библиотек, созданных сторонними разработчиками и доступных для свободного скачивания. На сайтах **https://pypi.python.org/pypi** и **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. Отметим лишь, что при выборе модуля необходимо учитывать версию Python, которая обычно указывается в составе имени файла с дистрибутивом.

Для языка Python существует и полноценный компилятор, который порождает обычные исполняемые EXE-файлы, не требующие для работы обязательной установки интерпретатора. Он реализован в виде отдельной библиотеки, носит название cx_Freeze и может быть найден по адресу **http://cx-freeze.sourceforge.net/**. Там же находится и документация по этой программе.

Если в процессе изучения языка возникнут вопросы, следует наведаться в поисках ответов на тематические форумы, — в частности, авторы советуют регулярно посещать

http://python.su/forum/. А вообще, большой список русскоязычных ресурсов, посвященных Python, можно отыскать по адресу **https://wiki.python.org/moin/RussianLanguage**.

К тому же, сам Интернет предоставляет множество ответов на самые разнообразные вопросы — достаточно набрать свой вопрос в строке запроса поискового портала (например, **http://www.bing.com/** или **http://www.google.com/**). Наверняка уже кто-то сталкивался с подобным вопросом и описал решение на каком-либо сайте.

Засим авторы прощаются с вами, уважаемые читатели, и желают успехов в нелегком, но таком увлекательном деле, как программирование!

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив к книге выложен на FTP-сервер издательства по адресу: **ftp://ftp.bhv.ru/9785977536318.zip**. Ссылка доступна и со страницы книги на сайте **www.bhv.ru**.

Содержимое архива описано в табл. П.1.

Таблица П.1. *Содержимое электронного архива*

Файл	Описание
Listings.doc	Все листинги из книги
Readme.txt	Описание электронного архива

Предметный указатель

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ч

Я