O'REILLY®

5-е издание
Рассмотрены версии 3.4 и 2.7

Python
Карманный справочник

PYTHON В ВАШЕМ КАРМАНЕ

ВИЛЬЯМС
www.williamspublishing.com

Марк Лутц

FIFTH EDITION

Python

Pocket Reference

Mark Lutz

Beijing · Cambridge · Farnham - Köln · Sebastopol · Tokyo

O'REILLY®

ПЯТОЕ ИЗДАНИЕ

Python

Карманный справочник

Марк Лутц

Москва · Санкт-Петербург · Киев
2015

ББК 32.973.26-018.2.75
Л86
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь
в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Лутц, Марк.
Л86 Python. Карманный справочник, 5-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 320 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1965-6 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly & Associates.

Authorized Russian translation of the English edition of *Python Pocket Reference* (ISBN 978-1-449-35701-6) © 2014 Mark Lutz.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание
Марк Лутц
Python. Карманный справочник
5-е издание

Литературный редактор	*И.А. Попова*
Верстка	*Л.В. Чернокозинская*
Художественный редактор	*В.Г. Павлютин*
Корректор	*Л.А. Гордиенко*

Подписано в печать 22.01.2015. Формат 84x108/32.
Гарнитура Times. Усл. печ. л. 16,9. Уч.-изд. л.12,32.
Тираж 1000 экз. Заказ № 377.

Отпечатано способом ролевой струйной печати
в ОАО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, т/ф. 8(496)726-54-10

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1965-6 (рус.)
ISBN 978-1-449-35701-6 (англ.)

© 2015, Издательский дом "Вильямс"
© 2014, Mark Lutz

Содержание

Карманный справочник по Python

Введение

Python — это язык общего назначения с открытым исходным кодом, применением множества парадигм, поддержкой структур объектно-ориентированного, функционального и процедурного программирования. Как правило, он применяется для написания как автономных программ, так и сценариев в самых разных областях и обычно считается одним из самых широко употребляемых языков программирования во всем мире.

К характерным особенностям Python относятся акцент на удобочитаемости исходного кода и функциональных возможностях библиотек, а также конструкция, оптимизирующая производительность труда разработчика, качество программного обеспечения, переносимость программ и интеграцию их компонентов. Программы на Python выполняются на большинстве общеупотребительных платформ, включая Unix и Linux, Windows, Mac OS и .NET, Android, iOS и пр.

В этом карманном справочнике вкратце рассматриваются типы данных и операторы языка Python, имена специальных методов, встроенные функции и исключения, общеупотребительные стандартные библиотечные модули и прочие примечательные языковые средства Python. Этот справочник предназначен в качестве краткого руководства для разработчиков в дополнение к другой литературе с упражнениями, примерами исходного кода и прочим учебным материалом.

В настоящем, *пятом* издании рассматриваются обе версии — Python2.X и 3.X. И хотя основное внимание в нем уделяется версии 3.X, по ходу изложения материала отмечаются отличия, имеющиеся в версии 2.X. В частности, настоящее издание обновлено по версиям Python *2.7* и *3.3*, а также наиболее примечательным усовершенствованиям в версии *3.4*, хотя большая часть содержания

этого справочника распространяется как на прежние, так и на последующие выпуски в рамках версий 2.X и 3.X.

Настоящее издание охватывает все основные реализации Python, в том числе CPython, PyPy, Jython, IronPython и Stackless. Оно обновлено и дополнено с учетом самых последних изменений в данном языке, его библиотеках и практике программирования на нем. К этим изменениям относятся порядок разрешения методов (MRO) и функция `super()`, формальные алгоритмы наследования, импорт, диспетчеры контекста, расположение кодовых блоков с отступами, наиболее употребительные библиотечные модули и инструментальные средства, в том числе `json`, `timeit`, `random`, `subprocess`, `enum`, а также новое средство запуска программ в Windows.

Принятые условные обозначения

В этой книге приняты следующие условные обозначения:

`[]`

В форматах синтаксиса элементы в квадратных скобках являются дополнительными, но *не* обязательными. Квадратные скобки используются буквально в некоторых частях синтаксиса Python, например, для обозначения списков, как отмечается там, где это уместно.

`*`

В форматах синтаксиса элементы, следующие после знака звездочки, могут повторяться многократно или вообще не повторяться. Знак звездочки используется буквально в некоторых частях синтаксиса Python, например, для обозначения арифметической операции умножения.

`a | b`

В форматах синтаксиса элементы, разделяемые знаком прямой черты, являются альтернативными. Знак прямой черты используется буквально в некоторых частях синтаксиса Python, например, для обозначения операции логического сложения.

Курсив

Служит для обозначения компонентов Python, а также для выделения новых и важных терминов.

`Моноширинный шрифт прямой`

Служит для обозначения исходного кода, имен файлов, модулей, операторов, функций, атрибутов, переменных и методов.

`Моноширинный шрифт наклонный`

Служит для обозначения имен заменяемых параметров в синтаксисе команд, вводимых из командной строки, выражений, функций и методов.

`Моноширинный шрифт полужирный`

Служит для обозначения параметров и аргументов командной строки, а также команд Python, операторов, знаков и цифр там, где это уместно.

`Function()`

Обозначает вызываемые функции и методы с завершающими круглыми скобками, чтобы отличать их от других типов атрибутов, кроме тех случаев, где на это указывается специально.

См. раздел *"Заголовок раздела"*.

Обозначает ссылку на другие разделы этой книги с указанием их заголовков в двойных кавычках.

НА ЗАМЕТКУ

Принятые в этой книге обозначения *2.X* и *3.X* указывают на то, что рассматриваемая тема относится ко всем общеупотребительным выпускам Python в рамках версий 2.X и 3.X. А более конкретные номера версий указываются в темах с более ограниченными рамками (например, обозначение *2.7* указывает только на версию 2.7). Будущие изменения в Python могут сделать непригодными некоторые рассматриваемые здесь языковые средства в последующих

версиях, поэтому в дальнейшем рекомендуется обращаться к разделу "What's New in Python" (Что нового в Python) документации на Python по адресу `http://docs.python.org/3/whatsnew/index.html`.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д.43, стр. 1
в Украине: 03150, Киев, а/я 152

Запуск программ на Python из командной строки

Командные строки служат для запуска программ на Python из командной оболочки системы в следующем формате:

```
python [параметр*]
  [ файл_сценария | -c команда | -m модуль | - ] [arg*]
```

В этом формате *python* обозначает интерпретатор языка Python, исполняемый как по полностью указанному пути к каталогу, так и по слову *python*, зарезервированному в командной оболочке системы (например, в переменной окружения PATH). Параметры командной строки (*параметр*), определяющие режим работы интерпретатора Python, обычно указываются перед именем исполняемой программы, тогда как аргументы (*arg*), требующиеся для выполнения программы, — после ее имени.

Параметры командной строки в Python

Элементы *параметр* командной строки служат для обозначения режима работы самого интерпретатора Python. В версии Python 3.X допускаются приведенные ниже элементы *параметр* командной строки, а их отличия в версии 2.X см. далее, в разделе "Параметры командной строки в версии Python 2.X".

-b

Выдать предупреждения об ошибках при вызове функции `str()` с объектом типа `bytes` или `bytearray`, но без аргумента, обозначающего способ кодирования символов, а также при сравнении данных типа `bytes` или `bytearray` с данными типа `str`. А при указании параметра **-bb** выдаются непосредственно ошибки.

-B

Не записывать байт-код в файлы с расширением **.pyc** или **.pyo** при импорте.

-d

Активизировать вывод результатов отладки синтаксического анализатора (служит для разработчиков ядра интерпретатора Python).

-E

Игнорировать описываемые далее переменные окружения Python (например, переменную `PYTHONPATH`).

-h

Вывести вспомогательное сообщение и выйти из программы.

-i

Войти в диалоговый режим работы после выполнения сценария. *Совет:* этот параметр удобен для отладки программ после отказов. См. также описание функции `pdb.pm()` в руководстве по библиотекам Python.

-O

Оптимизировать генерируемый байт-код, создавая и используя файлы с расширением **`.pyc`** для хранения байт-кода. В настоящее время этот параметр не дает заметных преимуществ в производительности.

-OO

Этот параметр действует подобно параметру **-O**, но при его указании удаляются также строки документации из байт-кода.

-q

Не выводить сообщение о версии и авторском праве при запуске программ в диалоговом режиме, начиная с версии Python 3.2.

-s

Не указывать местный каталог пользователя в пути поиска модулей в переменной `sys.path`.

-S

Не подразумевать импорт модулей из местного каталога пользователя при инициализации.

-u

Сделать принудительно небуферизованными и двоичными стандартные потоки вывода данных (*stdout*) и ошибок (*stderr*).

-v

Выводить сообщение всякий раз, когда инициализируется модуль, показывая место, из которого он загружен. Для более подробного вывода этот параметр можно повторить.

-V

Вывести номер версии Python и выйти из программы (этот параметр может быть также указан в виде **--version**).

-W *arg*

Этот параметр управляет выдачей предупреждений, где аргумент ***arg*** принимает вид *действие*:*сообщение*:*категория*:*модуль*:*номер_строки*. См. далее разделы "Каркас предупреждений" и "Исключения категории предупреждений", а также документацию на модуль `warnings` в справочном руководстве по библиотеке Python (Python Library Reference), доступном по адресу `http://www.python.org/doc/`.

-x

Пропустить первую строку исходного кода, разрешая использовать формы директив #! *cmd*, отличающиеся от принятых в Unix.

-X *параметр*

Установить параметр в зависимости от реализации, начиная с версии Python 3.2. Поддерживаемые значения, которые принимает *параметр*, приведены в документации на конкретную реализацию.

Указание программ в командной строке

Исполняемый код программы на Python и передаваемые ей аргументы могут быть указаны в командной строке следующими способами:

файл_сценария

Обозначает имя файла сценария на языке Python, который должен выполняться как основной файл программы, находящийся на самом верхнем уровне ее иерархии (например, по команде **`python main.py`** выполняется код из файла `main.py`). Имя файла сценария может быть указано как по абсолютному, так и по относительному пути (`.`) и доступно в элементе `sys.argv[0]` списка аргументов. В командных строках на некоторых платформах элемент *`python`* может быть опущен, если они начинаются с имени файла сценария и не содержат параметры, определяющие режим работы самого интерпретатора Python.

`-c` ***команда***

Обозначает (в виде символьной строки) исполняемый код Python (например, по команде **`python -c "print('spam' * 8)"`** в Python выполняется операция вывода на печать). Значение `'-c'` устанавливается в элементе `sys.argv[0]` списка аргументов.

`-m` ***модуль***

Выполняет модуль в виде сценария. Поиск *`модуля`* осуществляется по пути в переменной `sys.path`, а его выполнение — в виде файла, находящего на самом верхнем уровне иерархии (например, по команде **`python -m pdb s.py`** модуль `pdb` отладки программ на Python, находящийся в каталоге стандартной библиотеки, выполняется с аргументом `s.py`). Полный путь к модулю указывается в элементе `sys.argv[0]` списка аргументов.

`-`

Вводит команды Python из стандартного потока ввода (*`stdin`* — по умолчанию). Входит в диалоговый режим работы, если команды вводятся из стандартного потока ввода `tty` (интерактивного устройства). Значение `'-'` указывается в элементе `sys.argv[0]` списка аргументов.

arg*

Обозначает, что все остальное в командной строке передается файлу сценария или команде и доступно в элементе `sys.argv[1:]` списка аргументов.

Если ни один из параметров *файл_сценария*, *команда* или *модуль* не указан в командной строке, интерпретатор Python переходит в диалоговый режим работы, вводя команды из стандартного потока ввода *stdin* и используя для этой цели библиотеку *readline* из проекта GNU, при условии, что она установлена, а также задавая значение `'-'` (пустую строку) в элементе `sys.argv[0]` списка аргументов, если только интерпретатор не вызывается с упомянутым выше параметром **-**.

Помимо традиционных командных строк, вводимых по приглашению системной командной оболочки, программы на Python можно запускать щелчком кнопкой мыши на именах их файлов в графическом пользовательском интерфейсе (ГПИ) проводника по файлам, вызывая функции из стандартной библиотеки Python (например, функцию `os.popen()`), а также выбирая соответствующие команды запуска из меню интегрированных сред разработки (ИСР) вроде IDLE, Komodo, Eclipse или NetBeans.

Параметры командной строки в версии Python 2.X

В версии Python 2.X поддерживается тот же самый формат командной строки, что и в версии 3.X. Но в ней отсутствует поддержка параметра **-b**, что связанно с изменениями в строковом типе данных, а также параметров **-q** и **-X**, введенных в версии 3.X. В то же время поддерживаются приведенные ниже дополнительные параметры, доступные в версиях 2.6 и 2.7, а возможно, и в прежних версиях Python.

-t и **-tt**

Выдают предупреждения о несогласованном совместном употреблении символов табуляции и пробелов при отступах. А при указании параметра **-tt** выдаются непосредственно ошибки. В версии 3.X такое совместное

употребление символов табуляции и пробелов всегда интерпретируется как синтаксические ошибки (дополнительно см. далее раздел "Правила синтаксиса").

`-Q`

Это параметры разделения **`-Qold`** (по умолчанию), **`-Qwarn`**, **`-Qwarnall`** и **`-Qnew`**. Эти параметры относятся к новому режиму подлинного разделения в версии Python 3.X (см. далее раздел "Примечания к применению операторов").

`-3`

Выдает предупреждения о любых признаках несовместимости с версией Python 3.X в исходном коде, которую не в состоянии заведомо устранить инструментальное средство *2to3* из стандартной установки Python.

`-R`

Активизирует псевдослучайную затравку, чтобы сделать непредсказуемыми хеш-значения различных типов в промежутках между последовательными вызовами интерпретатора и тем самым предотвратить атаки типа отказа в обслуживании. Этот параметр появился в версии Python 2.6.8 и остался ради совместимости в версии 3.X, начиная с выпуска 3.2.3, хотя такого рода рандомизация активизируется по умолчанию, начиная с выпуска 3.3.

Переменные окружения Python

Переменные окружения (или так называемые переменные *командной оболочки*) устанавливаются на уровне системы. Они доступны в программах и применяются для их глобальной настройки.

Операционные переменные

Ниже перечислены основные, настраиваемые пользователем переменные окружения, связанные с режимом работы сценариев.

PYTHONPATH

Расширяет исходный путь поиска импортируемых файлов модулей. Формат значения этой переменной такой же, как и у значения, устанавливаемого в переменной окружения `PATH` командной оболочки, а именно: имена путей к каталогам разделяются двоеточиями (или точками с запятой в Windows). Если переменная `PYTHONPATH` установлена, то поиск импортируемых файлов или каталогов модулей осуществляется в каждом каталоге, перечисленном в этой переменной слева направо. Значение этой переменной включается в переменную окружения `sys.path`, предназначенную для хранения полного пути поиска импортируемых модулей в крайних слева составляющих абсолютного пути, после каталога со сценарием и перед каталогами стандартных библиотек. См. далее описание переменной окружения `sys.path` в разделах "Модуль `sys`" и "Оператор `import`".

PYTHONSTARTUP

Если в этой переменной задано имя читаемого файла, то команды Python из этого файла выполняются до появления первого приглашения в диалоговом режиме работы, что удобно для определения часто используемых инструментальных средств.

PYTHONHOME

Если эта переменная установлена, то ее значение используется в качестве каталога библиотечных модулей с чередующимся префиксом (или `sys.prefix`, `sys.exec_prefix`). По умолчанию поиск модулей осуществляется по пути `sys.prefix/lib`.

PYTHONCASEOK

Если эта переменная установлена, то в операторах импорта регистр в именах файлов не учитывается (в настоящее время поддерживается только в Windows и Mac OS X).

PYTHONIOENCODING

Этой переменной присваивается символьная строка формата *наименование_кодировки*`[:`*обработчик_ошибок*`]`

для замены кодировки в уникоде (и дополнительно — обработчика ошибок) при вводе текста из стандартного потока *stdin* и его выводе в стандартные потоки *stdout* и *stderr*. Такая настройка может потребоваться в некоторых командных оболочках для обработки текста, не представленного в коде ASCII. Так, если вывод текста окажется неудачным, можно попробовать задать в этой переменной кодировку UTF-8 (т.е. значение "utf8").

PYTHONHASHSEED

Если в этой переменной задано случайное начальное значение, оно используется для хеширования объектов типа str, bytes и datetime. А для получения хеш-значений с предсказуемым начальным значением в этой переменной можно установить целое значение в пределах от **0** до **4294967295**. Такая возможность поддерживается в версиях Python 2.6.8 и 3.2.3.

PYTHONFAULTHANDLER

Если эта переменная установлена, то обработчики регистрируются при запуске программы на Python, чтобы вывести содержимое оперативной памяти и отследить снизу вверх причины фатальных ошибок. Начиная с версии Python 3.3, это равнозначно указанию параметра **-X** *обработчик_отказов* в командной строке.

Переменные, аналоги параметров командной строки в Python

Приведенные ниже переменные окружения являются аналогами некоторых параметров командной строки в Python (см. раздел "Параметры командной строки в Python").

PYTHONDEBUG

Если эта переменная не пустая, то она действует аналогично параметру **-d**.

PYTHONDONTWRITEBYTECODE

Если эта переменная не пустая, то она действует аналогично параметру **-B**.

PYTHONINSPECT

Если эта переменная не пустая, то она действует аналогично параметру **-i**.

PYTHONNOUSERSITE

Если эта переменная не пустая, то она действует аналогично параметру **-s**.

PYTHONOPTIMIZE

Если эта переменная не пустая, то она действует аналогично параметру **-O**.

PYTHONUNBUFFERED

Если эта переменная не пустая, то она действует аналогично параметру **-u**.

PYTHONVERBOSE

Если эта переменная не пустая, то она действует аналогично параметру **-v**.

PYTHONWARNINGS

Если эта переменная не пустая, то она действует аналогично параметру **-W** с тем же самым значением. Эта переменная принимает также разделенную запятыми символьную строку, равнозначную указанию нескольких параметров **-W**. Такая возможность имеется в версиях Python 2.7 и 3.2.

Запуск программ на Python в Windows

Начиная с версии Python 3.3 в Windows (и только в этой операционной системе) устанавливается средство запуска сценариев, которое было доступно отдельно в прежних версиях. Это средство состоит из исполняемых файлов `py.exe` (консольный вариант) и `pyw.exe` (бесконсольный вариант), которые можно вызывать без настроек переменной окружения `PATH`. Эти файлы

регистрируются для выполнения файлов Python через сопоставления типов файлов и позволяют выбирать версии Python с помощью:

- Unix-подобных директив #!, указываемых в самом начале сценариев;
- аргументов командной строки;
- параметров, настраиваемых по умолчанию.

Директивы запуска файлов

Средство запуска распознает строки кода с директивами #! в самом начале файлов сценариев. В этих директивах указываются версии Python в одной из приведенных ниже форм, где * обозначает использование версии *по умолчанию* (в настоящее время — версии **2**, если она установлена, что равнозначно пропуску директивы #!); номера *основной* версии (например, **3**) для запуска самой последней установленной версии или же *полной* спецификации в форме *основная_версия.упрощенная_версия* с дополнительным суффиксом **-32**, обозначающим предпочтительный вариант установки 32-разрядной версии (например, 3.1-32).

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

Любые аргументы команд Python (python.exe) могут быть заданы в конце строки, а в Python 3.4 и более поздней версии можно обратиться к переменной окружения PATH для получения строк с директивами #!, предоставляющими только обозначение python без явного указания номера версии.

Командные строки для запуска

Средство запуска может быть также вызвано из командных строк по приглашению системной командной оболочки в следующей форме:

py [*pyarg*] [*pythonarg] script.py [*scriptarg**]**

В более общем случае все, что может появиться в команде python после составляющей python, может также появиться после дополнительного аргумента *pyarg* в команде py и быть дословно передано средству запуска программ на Python. К этому относятся параметры **-m**, **-c** и **-** из форм спецификации программ (см. раздел "Запуск программ на Python из командной строки").

Средство запуска принимает приведенные ниже формы своего дополнительного, но не обязательного аргумента *pyarg*, отражающие составляющую * из строки с директивой #! в конце файла сценария.

```
-2          Запустить последнюю установленную версию 2.X
-3          Запустить последнюю установленную версию 3.X
-X.Y        Запустить указанную версию, где X равно 2 или 3
-X.Y-32     Запустить указанную 32-разрядную версию
```

Если присутствует и то и другое, то предпочтение отдается аргументам командной строки над значениями в строках с директивой #!. При соответствующей установке строки с директивой #! могут применяться и в более широком контексте (например, при выборе пиктограмм щелчком мышью).

Переменные окружения для запуска

В средстве запуска распознаются также настройки дополнительных переменных окружения, которые могут быть использованы для специальной настройки выбора версии в стандартных или частных случаях (например, пропуск версии, ее указание только в основной директиве #! или в аргументе командной строки py), как показано ниже. Все эти настройки используются только в исполняемых файлах средства запуска, но не при непосредственном вызове команды *python*.

```
PY_PYTHON   Версия для стандартных случаев (иначе версия 2)
PY_PYTHON3  Версия для 3 частных случаев (например, 3.2)
PY_PYTHON2  Версия для 2 частных случаев (например, 2.6)
```

Встроенные типы и операторы

Операторы и их предшествование

В табл. 1 перечислены операторы выражений Python. Операторы в нижних ячейках этой таблицы имеют более высокое предшествование (т.е. более тесную привязку), когда они применяются в выражениях с разнотипными операторами без круглых скобок.

Таблица 1. Операторы выражений Python 3 и их предшествование

Оператор	Описание
`yield X`	Возвращает результат выполнения функции-генератора (значение функции `send()`)
`lambda args1: X`	Создает анонимную функцию (возвращает при вызове значение *X*)
`X if Y else Z`	Тернарный оператор выбора (значение *X* вычисляется только в том случае, если *Y* истинно)
`X or Y`	Логическая операция ИЛИ: значение *Y* вычисляется лишь в том случае, если *X* ложно
`X and Y`	Логическая операция И: значение *Y* вычисляется лишь в том случае, если *X* истинно
`not X`	Логическое отрицание
`X in Y, X not in Y`	Членство: итераторы, множества
`X is Y, X is not Y`	Проверки объектов на идентичность
`X < Y, X <= Y, X > Y, X >= Y`	Сравнение величин, подмножеств и надмножеств отдельных множеств
`X == Y, X != Y`	Операторы равенства
`X | Y`	Логическая операция поразрядное ИЛИ, объединение множеств
`X ^ Y`	Логическая операция поразрядное исключающее ИЛИ, разноименность множеств
`X & Y`	Логическая операция поразрядное И, пересечение множеств
`X << Y, X >> Y`	Сдвиг значения *X* влево или вправо на количество битов, равное *Y*
`X + Y, X - Y`	Сложение/сцепление, вычитание/разность множеств
`X * Y, X % Y, X / Y, X // Y`	Умножение/повторение, получение остатка от деления/форматирование, деление, целочисленное деление
`-X, +X`	Унарное отрицание, идентичность

Оператор	Описание
`~X`	Логическая операция поразрядное НЕ в дополнительном коде (инверсия)
`X** Y`	Возведение в степень
`X[i]`	Индексирование (последовательностей, отображений и прочего)
`X[i:j:k]`	Нарезка (все три границы необязательны)
`X(args2)`	Вызов (функции, метода, класса и другого вызываемого объекта)
`X.attr`	Ссылка на атрибут
`(....)`	Кортеж, выражение, выражение-генератор
`[....]`	Список, генератор списков
`{....}`	Словарь, множество, генератор словарей и множеств

Атомарные члены и динамическая типизация

Заменяемые элементы выражений `X`, `Y`, `Z`, `i`, `j` и `k` в табл. 1 могут быть следующими.

- *Имена переменных*, заменяемые самым последним присвоенным им значением.
- *Литеральные выражения*, определяемые далее в разделе "Конкретные встроенные типы".
- *Вложенные выражения*, выбираемые из любой строки в табл. 1 (возможно, в круглых скобках).

Переменные в Python следуют *модели динамической типизации*, т.е. они не объявляются, но создаются, когда им присваиваются значения; хранят ссылки на объекты в виде значений; могут ссылаться на любой тип объекта, а значения должны присваиваться им до их употребления в выражениях, поскольку они не имеют значений по умолчанию. В именах переменных непременно учитывается регистр (подробно об этом — ниже, в разделе "Правила именования"). Объекты, на которые ссылаются переменные, автоматически создаются и освобождаются из оперативной памяти системой "сборки мусора" Python, когда они больше не нужны. Для этой цели в реализации CPython используется подсчет ссылок.

Кроме того, заменяемый элемент ***attr*** в табл. 1 должен быть буквальным именем атрибута (без кавычек); элемент ***args1*** — списком формальных аргументов, определяемым далее в разделе "Оператор `def`"; элемент ***args2*** — списком входных аргументов, определяемым далее в разделе "Оператор выражения"; а литерал `...` описывается как атомарное выражение (только в версии 3.X). Синтаксис генераторов литералов и структур данных (кортежей, списков и множеств), указываемых в абстрактной форме в трех последних строках табл. 1, определяется далее в разделе "Конкретные встроенные типы".

Примечания к применению операторов

- Неравенство значений может быть записано как *X* `!=` *Y* или *X* `<>` *Y*, но только в версии Python 2.X. А в версии Python 3.X последний из этих двух вариантов записи исключен как избыточный.
- Выражение в обратных кавычках `'`*X*`'` действует аналогично выражению `repr(`*X*`)`, преобразуя объекты в отображаемые символьные строки, но только в версии Python 2.X. В версии Python 3.X для этой цели используются более понятные и удобочитаемые встроенные функции `str()` и `repr()`.
- В обеих версиях, Python 2.X и 3.X, выражение *целочисленного деления* *X* `//` *Y* всегда приводит к отбрасыванию дробных остатков и возврату целочисленного результата для целых исходных значений.
- Выражение *X* `/` *Y* выполняет *настоящее деление* в версии 3.X, всегда сохраняя в результате остаток в формате с плавающей точкой, а в версии 2.X — *классическое деление*, отбрасывая остаток от деления целых значений, кроме тех случаев, когда в версии 2.X активизирован режим настоящего деления из версии 3.X с помощью оператора `from __future__ import division` или параметра **`-Qnew`** командной строки в Python.
- Синтаксис `[....]` служит для обозначения списочных литералов и выражений-генераторов. В последнем случае

выполняется подразумеваемый цикл, а результаты вычисления выражения накапливаются в новом списке.

- Синтаксис `(....)` служит для обозначения кортежей и выражений, а также выражений-генераторов — формы генераторов списков, выдающей результаты по требованию вместо построения списка результатов. Круглые скобки могут иногда опускаться во всех трех языковых конструкциях.
- Синтаксис `{....}` служит для обозначения словарных литералов. В версиях Python 2.7 и 3.X используются также литералы множеств, генераторы словарей и множеств. В Python 2.6 и более ранних версиях для этой цели следует использовать функцию `set()` и операторы циклов.
- Оператор выбора `yield` и условные операторы `if/else` доступны, начиная с версии Python 2.5. Так, оператор `yield` возвращает аргументы функции `send()` в генераторах, а условные операторы `if/else` служат краткой формой многострочного условного оператора `if`. Оператор `yield` требуется указывать в круглых скобках, если он не является единственным в правой части оператора присваивания.
- Операторы сравнения можно объединять в цепочки. Так, выражение $X < Y < Z$ дает такой же результат, как и выражения $X < Y$ и $Y < Z$, но при объединении в цепочку значение Y вычисляется только один раз.
- Выражение нарезки `X[i:j:k]` равнозначно индексации с помощью объекта нарезки: `X[slice(i, j, k)]`.
- В версии Python 2.X допускается сравнение разнотипных величин с приведением числовых значений к общему типу и упорядочением других типов данных в соответствии с наименованием типа. В версии Python 3.X сравнение разнотипных нечисловых величин, в том числе и при сортировке по объекту-заместителю, не допускается, но приводит к возникновению исключений.
- Сравнение величин в словарях также не поддерживается, начиная с версии Python 3.X, кроме проверки на равенство. Поэтому одним из возможных вариантов замены

сравнения в версии 3.X может служить вызов функции `sorted(adict.items())`.

- В выражениях с вызовом функций допускается указывать позиционные и именованные аргументы, а также произвольные крупные числа в качестве аргументов. Синтаксис вызовов функций приведен далее, в разделах "Оператор выражения" и "Оператор `def`".
- В версии Python 3.X допускается указывать многоточие (буквально — `...` или с помощью встроенного имени `Ellipsis`) для обозначения атомарных выражений в исходном коде. Такое обозначение может служить альтернативой оператору `pass` или объекту `None` в некоторых видах контекста (например, в теле фиктивных функций или при инициализации переменных независимо от их типа).
- В последующих версиях, начиная с Python 3.5, *может* быть обобщен, хотя полной уверенности в этом пока нет, синтаксис выражений `*X` и `**X` со знаком звездочки, появляющихся в литералах структур данных и генераторах, где подобным образом коллекции распаковываются в отдельные элементы, как это в настоящее время делается при вызове соответствующих функций. Подробнее об этом — ниже, в разделе "Оператор присваивания".

Категории операций

В именах методов типа `__X__`, упоминаемых в этом разделе, завершающие круглые скобки ради краткости опускаются. В общем, все встроенные типы данных поддерживают операции *сравнения* и *логические* операции, перечисленные в табл. 2. Хотя в версии 3.X не поддерживаются операции сравнения величин в словарях или разнотипных числовых типов.

Таблица 2. Операции сравнения и логические операции

Оператор	Описание
`X < Y`	Строго меньше[1]
`X <= Y`	Меньше или равно

Оператор	Описание
`X > Y`	Строго больше
`X >= Y`	Больше или равно
`X == Y`	Равно (одно и то же значение)
`X != Y`	Не равно (то же, что и операция `X <> Y`, только в версии Python 2.X)[2]
`X is Y`	Один и тот же объект
`X is not Y`	Отрицание идентичности объектов
`X < Y < Z`	Сравнение по цепочке
`not X`	Если `X` ложно, то результат операции истинен (`True`), в противном случае — ложен (`False`)
`X or Y`	Если `X` ложно, то результат операции равен значению `Y`, в противном случае — значению `X`
`X and Y`	Если `X` ложно, то результат операции равен значению `X`, в противном случае — значению `Y`

1. Для реализации выражений сравнения см. методы из классов для расширенного сравнения в версиях 2.X и 3.X (например, метод `__lt__`*, заменяющий оператор* `<`*), а также общий метод* `__cmp__`*, описываемый далее в разделе "Методы перегрузки операторов".*

2. Оба оператора, `!=` *и* `<>`*, обозначают неравенство значений в версии 2.X, но в этой версии предпочтение все же отдается оператору* `!=`*, тогда как в версии 3.X поддерживается только он. Оператор* `is` *выполняет проверку на идентичность, а оператор* `==` *— сравнение значений, и поэтому в целом он намного более полезный.*

Истинное логическое значение означает любое ненулевое число или любой непустой объект из коллекции (списка, словаря и т.д.), причем все объекты имеют логическое значение. Для обозначения истинного логического значения служит встроенное имя `True`, а для обозначения ложного логического значения — встроенное имя `False`, что равнозначно целочисленным значениям `1` и `0` соответственно в специальных форматах отображения. Специальный объект `None` имеет ложное логическое значение и часто присутствует в различных контекстах Python.

Операции сравнения возвращают логическое значение `True` или `False`. Они автоматически применяются рекурсивно в составных объектах, когда требуется определить конечный результат.

Логические операторы **and** и **or** выполняются укороченно, прекращая свое действие, как только результат становится известным. Они возвращают объекты одного из двух операндов, т.е. значение слева или справа от оператора, логическое значение которого дает результат.

В табл. 3–6 перечислены общие для всех типов операции, разделяемые на три категории: *последовательность* (упорядочение по позициям), *отображение* (доступ по ключу) и *число* (для всех числовых типов данных), а также операции, доступные для *изменяемых* типов данных в Python. Большинство типов данных экспортируют также дополнительные, характерные для этих типов операции (например, методы), как поясняется в разделе "Конкретные встроенные типы".

Таблица 3. Операции над последовательностями (символьными строками, списками, кортежами, байтами и массивами байтов)

Операция	Описание	Метод из класса
`X in S`	Проверка на членство	`__contains__`
`X not in S`		`__iter__`
		`__getitem__`[1]
`S1 + S2`	Сцепление	`__add__`
`S * N, N * S`	Повторение	`__mul__`
`S[i]`	Индексирование по смещению	`__getitem__`
`S[i:j], S[i:j:k]`	Нарезка: элементы из последовательности `S` от `i`-й до `j-1`-й позиции, дополнительно с шагом `k`	`__getitem__`[2]
`len(S)`	Длина	`__len__`
`min(S)`	Минимальный и	`__iter__`
`max(S)`	максимальный элементы	`__getitem__`
`iter(S)`	Протокол итерации	`__iter__`
`for X in S:`	Итерация (во всех контекстах)	`__iter__`
`[expr for X in S]`		`__getitem__`
`map(func, S)` и т.д.		

Дополнительные сведения об этих методах и их взаимозаменяемости приведены ниже, в подразделе "Протокол итерации". Если определен метод `__contains__`*, то предпочтение следует отдавать именно ему, но не методу* `__iter__`*, а последнему — над методом* `__getitem__`*.*

В версии Python 2.X можно также определить методы `__getslice__`, `__setslice__` *и* `__delslice__`, *чтобы выполнять операции нарезки. В версии 3.X эти методы исключены в пользу передачи объектов нарезки их аналогам с поэлементной индексацией. Объекты нарезки могут использоваться явным образом в выражениях индексации вместо указания границ нарезки* `i:j:k`.

Таблица 4. Операции над изменяемыми последовательностями (списками, массивами байтов)

Операция	Описание	Метод из класса
`S[i] = X`	Присваивание по индексу: заменяет значение элемента по существующему индексу `i` ссылкой на объект `X`	`__setitem__`
`S[i:j] = I`	Присваивание по нарезке: элементы из последовательности `S` от `i`-й и до `j-1`-й позиции, дополнительно с шагом `k` (возможно, пустая последовательность)	`__setitem__`
`S[i:j:k] = I`	Заменяются всеми элементами из итерируемого объекта `I`	
`del S[i]`	Удаление по индексу	`__delitem__`
`del S[i:j]`	Удаление по нарезке	`__delitem__`
`del S[i:j:k]`		

Таблица 5. Операции отображения (над словарями)

Операция	Описание	Метод из класса
`D[k]`	Индексирование по ключу	`__getitem__`
`D[k]=X`	Присваивание ключа: изменяет или создает запись по ключу `k` для ссылки на объект `X`	`__setitem__`
`del D[k]`	Удаление по ключу	`__delitem__`
`len(D)`	Длина (количество ключей)	`__len__`
`k in D`	Проверка ключа на членство[1]	То же, что и в табл. 3
`k not in D`	Операция, противоположная операции `k in D`	То же, что и в табл. 3
`iter(D)`	Объект итератора для ключей из словаря `D`	То же, что и в табл. 3
`for k in D:` и т.д.	Итерация ключей в словаре `D` (во всех контекстах итерации)	То же, что и в табл. 3

1. В версии Python 2.X членство ключей можно также запрограммировать, вызвав метод `D.has_key(k)`. *Этот метод исключен в версии Python 3.X в пользу выражения* `in`, *которому отдается предпочтение и в версии 2.X (см. далее раздел "Словари").*

Таблица 6. Числовые операции (над всеми типами данных)

Операция	Описание	Метод из класса
`X + Y, X - Y`	Сложение и вычитание	`__add__, __sub__`
`X * Y, X / Y, X // Y, X % Y`	Умножение, деление, целочисленное деление, деление с остатком	`__mul__, __truediv__`[1], `__floordiv__, __mod__`
`-X, +X`	Отрицание, идентичность	`__neg__, __pos__`
`X | Y, X & Y, X ^ Y`	Поразрядное ИЛИ, И, исключающее ИЛИ (целых значений)	`__or__, __and__, __xor__`
`X << N, X >> N`	Поразрядный сдвиг влево и вправо (целых значений)	`__lshift__, __rshift__`
`~X`	Поразрядное инвертирование (целых значений)	`__invert__`
`X ** Y`	Возведение `X` в степень `Y`	`__pow__`
`abs(X)`	Получение абсолютного значения	`__abs__`
`int(X)`	Преобразование в целочисленное значение[2]	`__int__`
`float(X)`	Преобразование в числовое значение с плавающей точкой	`__float__`
`complex(X)`	Получение комплексного числового значения	`__complex__`
`complex(re, im)`	—"—	
`divmod(X, Y)`	Кортеж: `(X / Y, X % Y)`	`__divmod__`
`pow(X, Y[, Z])`	Возведение в степень	`__pow__`

Оператор / вызывает метод `__truediv__` *в версии Python 3.X, а в версии Python 2.X — метод* `__div__`*, если настоящее деление не активизировано. О семантике деления см. ранее в разделе "Примечания к применению операторов".*

В версии Python 2.X из встроенной функции `long()` *вызывается метод* `__long__` *из класса. А в версии Python 3.X тип* `int` *относится к категории типа* `long`*, который исключен.*

Примечания к операциям над последовательностями

Ниже приведены примечания и примеры избранных операций над последовательностями из табл. 3 и 4.

Индексирование: `S[i]`

- Элементы извлекаются по индексу (первый элемент находится по нулевому индексу, т.е. имеет нулевое смещение).
- Отрицательные индексы отсчитываются от конца последовательности (последний элемент находится по индексу `-1`, т.е. имеет смещение `-1`).
- Операция `S[0]` означает извлечение первого элемента, операция `S[1]` — извлечение второго элемента, а операция `S[-2]` — предпоследнего элемента, что равнозначно операции `S[len(S) - 2]`.

Нарезка: `S[i:j]`

- Извлечение смежных частей последовательности от `i`-го до `j-1`-го элемента.
- По умолчанию границы нарезки `i` и `j` равны нулю, а длина последовательности — `len(S)`.
- Операция `S[1:3]` означает извлечение части последовательности от `1`-го до `3`-го элемента, но не включая последний.
- Операция `S[1:]` означает извлечение части последовательности от `1`-го элемента и до самого ее конца, что равнозначно операции `len(S)-1`.
- Операция `S[:-1]` означает извлечение части последовательности от нулевого элемента и до конца последовательности, но не включая последний ее элемент.
- Операция `S[:]` означает создание неполной (верхнего уровня) копии объекта последовательности `S`.

Расширенная нарезка: `S[i:j:k]`

- Третий элемент `k` обозначает шаг, который по умолчанию равен `1` и добавляется к смещению каждого элемента, извлекаемого из последовательности.
- Операция `S[::2]` означает извлечение элементов из всей последовательности `S` через одного.
- Операция `S[::-1]` означает обращение последовательности `S`.

- Операция `S[4:1:-1]` означает извлечение части последовательности в обратном порядке от `4`-го до `1`-го элемента, но не включая последний.

Присваивание по нарезке: `S[i:j:k] = I`

- Присваивание по нарезке аналогично удалению и последующей вставке новых элементов на месте удаленных.
- Итерируемые объекты, присваиваемые стандартным нарезкам `S[i:j]`, *не* должны совпадать по размеру.
- Итерируемые объекты, присваиваемые расширенным нарезкам `S[i:j:k]`, должны совпадать по размеру.

Прочее

- В результате сцепления, повторения и нарезки возвращаются новые объекты, хотя и не всегда кортежи.

Конкретные встроенные типы

В этом разделе рассматриваются числа, символьные строки, списки, словари, кортежи, файлы, множества и прочие базовые встроенные типы данных. В его подразделах подробно упоминается все общее, что имеется у обеих версий, Python2.X и 3.X. А в целом здесь рассматриваются все составные типы данных (например, списки, словари и кортежи), которые могут быть произвольно вложены друг в друга и настолько глубоко, насколько это требуется. Множества могут также участвовать во вложении, но содержать только неизменяемые объекты.

Числа

Числа являются *неизменяемыми* значениями, поддерживающими числовые операции. В этом разделе рассматриваются основные числовые типы данных (целочисленные и с плавающей точкой), а также более развитые типы данных (комплексные, десятичные и дробные числа).

Литералы и их создание

Числа записываются в самых разных числовых литеральных формах и создаются с помощью некоторых встроенных операций, как показано в приведенных ниже примерах.

`1234, -24, +42, 0`

Целые числа (неограниченной точности).[1]

`1.23, 3.14e-10, 4E210, 4.0e+210, 1., .1`

Числа с плавающей точкой (как правило, реализуются как числа с плавающей точкой двойной точности, реализованные в CPython таким же образом, как и в языке C).

`0o177, 0x9ff, 0b1111`

Восьмеричные, шестнадцатеричные или двоичные литералы для целых чисел.[2]

`3+4j, 3.0+4.0j, 3J`

Комплексные числа.

`decimal.Decimal('1.33'), fractions.Fraction(4, 3)`

Модульные типы: десятичные, дробные числа.

`int(9.1), int('-9'), int('1111', 2),`
`int('0b1111', 0), float(9), float('1e2'), float('-.1'),`
`complex(3, 4.0)`

В этих примерах числа образуются из других объектов или из символьных строк с возможной базовой точностью.

---

[1] В версии Python 2.X имеется отдельный тип `long` для целых чисел неограниченной точности. А тип `int` служит для обычных чисел с точностью, которая обычно ограничивается 32 разрядами. Объекты типа `long` обозначаются вместе с суффиксом `L` (например, **`99999L`**), хотя целые числа автоматически продвигаются к длинным числам, если им требуется дополнительная точность. В версии 3.X тип **`int`** предоставляет неограниченную точность и поэтому относится к обоим типам, **`int`** и **`long`**, в версии 2.X. Синтаксис литерала с суффиксом `L` исключен из версии 3.X.

[2] В версии Python 2.X восьмеричные литералы записываются также с начальными нулями, т.е. литералы **`0777`** и **`0o777`** равнозначны. А в версии Python 3.X последняя из этих двух форм поддерживается для восьмеричных литералов.

С другой стороны, при вызове функций `hex(N)`, `oct(N)` и `bin(N)` создаются символьные строки из цифр для целых чисел, а при форматировании образуются общие символьные строки для целых чисел. Дополнительно об этом — ниже, в подразделах "Форматирование символьных строк", "Преобразования типов" и "Встроенные функции".

Операции

Числовые типы поддерживают все *числовые операции* (см. табл. 6). В выражениях с разными типами Python операнды преобразуются в наивысший по иерархии тип данных, где целочисленные объекты находятся ниже объектов с плавающей точкой, а те — еще ниже комплексных объектов. В версиях Python 2.6 и 3.0 у целочисленных объектов и объектов с плавающей точкой имеется полдесятка *методов* для отдельных типов и другие *атрибуты*. Подробнее об этом см. в руководстве по библиотеке Python. Ниже приведены характерные примеры числовых операций.

```
>>> (2.5).as_integer_ratio()    # атрибуты с плавающей точкой
(5, 2)
>>> (2.5).is_integer()
False

>>> (2).numerator, (2).denominator    # целочисленные атрибуты
(2, 1)
>>> (255).bit_length(), bin(255)    # в версии 3.1+ метод
(8, '0b11111111')
```

Десятичные и дробные числа

В стандартных библиотечных модулях Python предоставляются два дополнительных числовых типа: *десятичный* для чисел с плавающей точкой и фиксированной точностью, а также *дробный* рациональный тип для хранения как числителя, так и знаменателя. Оба эти типа могут служить для устранения неточностей в арифметических операциях с плавающей точкой, как показано в приведенных ниже примерах.

```
>>> 0.1 - 0.3
-0.19999999999999998
>>> from decimal import Decimal
>>> Decimal('0.1') - Decimal('0.3')
Decimal('-0.2')

>>> from fractions import Fraction
>>> Fraction(1, 10) - Fraction(3, 10)
Fraction(-1, 5)
>>> Fraction(1, 3) + Fraction(7, 6)
Fraction(3, 2)
```

Дробные числа автоматически упрощают результаты. Фиксируя точность и поддерживая различные протоколы усечения и округления, десятичные числа оказываются очень удобными для финансовых приложений. Подробнее об этом см. в руководстве по библиотеке Python.

Другие числовые типы

В Python имеется также тип *множества*, описываемый далее в разделе "Множества". А дополнительные числовые типы, в том числе оптимизированные векторы и матрицы, доступны в качестве сторонних расширений с открытым кодом (см., например, пакет *NumPy* по адресу `http://www.numpy.org`). К числу сторонних услуг относится поддержка визуализации, статистических и математических операций с плавающей точкой и повышенной точностью и многое другое (дополнительные сведения по данному вопросу можно найти в Интернете).

Символьные строки

Обычный строковый объект типа `str` представляет собой *неизменяемую последовательность* символов, доступных по *смещению*, обозначающему их расположение в строке. Символы в строковом объекте являются порядковыми номерами кодовых точек в базовом наборе символов, а отдельные символы — строковыми объектами единичной длины. Полная модель строкового объекта в целом разнится. Так, в версии Python 3.X имеются следующие три строковые типы со сходными интерфейсами:

str

Неизменяемая последовательность символов, используемая для представления всех видов текста как в коде ASCII, как и в более расширенном уникоде (Unicode).

bytes

Неизменяемая последовательность коротких целых чисел, используемая для представления байтовых значений в двоичных данных.

bytearray

Изменяемый вариант представления последовательности байтов.

А в версии Python 2.X имеются два следующих строковых типа со сходными интерфейсами:

str

Неизменяемая последовательность символов, используемая для представления как байт-ориентированного (т.е. 8-разрядного) текста, так и двоичных данных.

unicode

Неизменяемая последовательность символов, используемая для представления текста в более расширенном уникоде.

Начиная с выпуска 2.6, в версии Python 2.X, появился также тип `bytearray` для обратной совместимости с версией Python 3.X, но он не предусматривает возможность проводить резкое различие между текстом и двоичными данными. (Этим типом данных можно свободно пользоваться вместе с текстовыми строками в версии 2.X.)

Дополнительные сведения о поддержке уникода в обеих версиях, 3.X и 2.X, изложены ниже, в разделе "Символьные строки в уникоде". Несмотря на то что в остальной части этого раздела упоминаются все строковые типы данных, дополнительные сведения о типах `bytes` и `bytearray` приведены далее, в подразделе "Строковые методы" и в разделах "Символьные строки в уникоде" и "Встроенные функции".

Строковые литералы и их создание

Строковые литералы записываются в виде последовательности символов в кавычках (дополнительно, но не обязательно с предшествующим обозначающим символом). Во всех формах строковых литералов пустая символьная строка обозначается парой смежных кавычек. А в результате выполнения различных встроенных операций возвращаются новые символьные строки:

`'Python"s', "Python's"`

Одиночные и двойные кавычки действуют одинаково. Это позволяет встраивать неэкранируемые кавычки одного вида в кавычках другого вида.

`"""This is a multiline block"""`

Блоки в тройных кавычках позволяют собирать несколько текстовых строк в одной символьной строке, вставляя знаки конца строки (**`\n`**) между исходными строками в кавычках.

`'Python\'s\n'`

Последовательности управляющего кода с предшествующими знаками обратной косой черты (табл. 7) заменяются представляющими их значениями кодовых точек специальных символов (например, **`'\n'`** — это символ в коде ASCII с десятичным значением **`10`** кодовой точки).

`"This" "is" "concatenated"`

Смежные строковые константы сцепляются. *Совет:* эта форма может охватывать и целые строки, если они заключены в круглые скобки.

`r'a raw\string', R'another\one'`

Неформатированные символьные строки, в которых знаки обратной косой черты интерпретируются буквально, кроме тех, что находятся в конце символьной строки. Эта форма удобна для обозначения регулярных выражений и путей к каталогам в Windows и DOS, например: `r'c:\dir1\file'`.

`hex(), oct(), bin()`

Эти функции служат для создания символьных строк из шестнадцатеричных, восьмеричных и двоичных целых

чисел. Дополнительно см. выше раздел "Числа" и ниже раздел "Встроенные функции".

В приведенных ниже формах строковых литералов и вызовах функций образуются специальные символьные строки, описываемые далее в разделе "Символьные строки в уникоде".

`b'...'`

Строковый литерал типа `bytes` в версии Python 3.X обозначает последовательность 8-разрядных байтовых значений, представляющих неформатированные двоичные данные. Для совместимости с версией 3.X эта форма доступна также в версиях Python 2.6 и 2.7, где просто создается обычная символьная строка типа `str`. Дополнительно об этом — ниже, в подразделе "Строковые методы" и разделах "Символьные строки в уникоде" и "Встроенные функции".

`bytearray(...)`

Строковая конструкция `bytearray` представляет собой изменяемый вариант типа `bytes` и доступна в версии Python 2.X, начиная с выпуска 2.6, а также в версии Python 3.X. Дополнительно см. далее подраздел "Строковые методы" и разделы "Символьные строки в уникоде" и "Встроенные функции".

`u'...'`

Строковый литерал в уникоде, внедренный в версии Python 2.X, представляет собой последовательность кодовых точек в уникоде. Ради совместимости с версией 2.X эта форма доступна и в версии Python 3.X, начиная с выпуска 3.3, где она просто образует обычную символьную строку типа `str`, хотя обычные строковые литералы и символьные строки типа `str` поддерживают текст в версии Python 3.X. См. далее раздел "Символьные строки в уникоде".

`str()`, **`bytes()`**, **`bytearray()`**, а также **`unicode()`**, но только в версии 2.X

Эти функции создают символьные строки из объектов с возможным кодированием и декодированием в версии Python 3.X. См. далее раздел "Встроенные функции".

Строковые литералы могут содержать управляющие последовательности для обозначения специальных символов, как показано в табл. 7.

Таблица 7. Управляющие коды строковых констант

Управляющий код	Назначение	Управляющий код	Назначение
`\`*`новая строка`*	Продолжение строки игнорируется	`\t`	Горизонтальная табуляция
`\\`	Обратная косая черта (\)	`\v`	Вертикальная табуляция
`\'`	Одиночная кавычка (')	`\N{`*`id`*`}`	Идентификатор базы данных в уникоде
`\"`	Двойные кавычки (")	`\u`*`hhhh`*	Шестнадцатеричное 16-разрядное значение в уникоде
`\a`	Звуковой сигнал	`\U`*`hhhhhhhh`*	Шестнадцатеричное 32-разрядное значение в уникоде[1]
`\b`	Возврат на одну позицию	`\x`*`hh`*	Шестнадцатеричное значение, состоящее как минимум из двух цифр
`\f`	Перевод страницы	`\`*`ooo`*	Восьмеричное значение, состоящее максимум из трех цифр
`\n`	Перевод строки	`\0`	Пустое значение (не обозначающее конец символьной строки)
`\r`	Возврат каретки	`\`*`другое`*	Не управляющий код

1. Управляющий код `\Uhhhhhhhh` *в точности принимает восемь шестнадцатеричных цифр (*`h`*). Оба обозначения,* `\u` *и* `\U`*, могут использоваться в строковых литералах в уникоде.*

Операции

Во всех строковых типах поддерживаются все *операции над последовательностями* (см. табл. 3) и специальные методы для обработки символьных строк, как поясняется далее в подразделе "Строковые методы". Кроме того, в типе `str` поддерживаются выражения `%` для *форматирования* символьных строк и рассматриваемая далее подстановка шаблонов, а в типе `bytearray` — *операции над изменяемыми последовательностями* (см. табл. 4) и дополнительные методы списочного типа). Дополнительно см. описание модуля `re` сопоставления со строковым шаблоном

далее в разделе "Модуль `re` сопоставления по шаблонам" и встроенных функций обработки символьных строк в разделе "Встроенные функции".

Форматирование символьных строк

В обеих версиях, Python 2.X и 3.X, начиная с выпуска 2.6 и 3.0 соответственно, обычные символьные строки типа `str` поддерживают два разных способа форматирования символьных строк, т.е. операции, в которых объекты форматируются в соответствии с описанием в форматирующих строках, как поясняется ниже.

- Исходное выражение (во всех версиях Python) обозначается с помощью оператора `%` следующим образом: *`fmt`* `%` `(`*`значения`*`)`.
- При вызове нового метода, внедренного в версиях 2.6, 3.0 и последующих версиях, символьные строки обозначаются с помощью следующего синтаксиса: *`fmt`*`.format(`*`значения`*`)`.

В обоих способах новые символьные строки формируются, если это возможно, на основании кодов подстановки, характерных для отдельных типов данных. А получаемые результаты могут быть отображены или присвоены переменным для дальнейшего употребления, как показано ниже.

```
>>> '%s, %s, %.2f' % (42, 'spam', 1 / 3.0)
'42, spam, 0.33'

>>> '{0}, {1}, {2:.2f}'.format(42, 'spam', 1 / 3.0)
'42, spam, 0.33'
```

В последнее время вызов метода считается более развитым способом, тем не менее, исходное выражение широко применяется в уже существующем коде, и поэтому обе формы по-прежнему поддерживаются. Более того, несмотря на то что представление о способе вызова метода косвенно кажется более мнемоническим и согласованным, исходное выражение нередко оказывается более простым и кратким. И хотя оба эти способа и соответствующие им формы, по существу, мало чем отличаются по своим функциональным возможностям и степени сложности, в настоящее время

отсутствуют какие-либо побудительные причины предпочесть один из них другому.

Выражение для форматирования символьных строк

В выражениях для форматирования символьных строк адресаты в левой части оператора `%` заменяются значениями в правой его части подобно тому, как это делается в функции `sprintf()` на языке C. Если требуется заменить несколько значений, они должны быть указаны в виде кортежа в правой части оператора `%`. Если же требуется заменить только один элемент, его можно указать как одиночное значение или одноэлементный кортеж в правой части оператора `%`, тогда как для форматирования самих кортежей допускается их вложение друг в друга. А если в левой части оператора `%` употребляются имена ключей, то в правой его части должен быть предоставлен словарь. И наконец, с помощью знака `*` допускается динамически передавать ширину и точность представления чисел:

```
>>> 'The knights who say %s!' % 'Ni'
'The knights who say Ni!'
>>> '%d %s, %d you' % (1, 'spam', 4.0)
'1 spam, 4 you'
>>> '%(n)d named %(x)s' % {'n': 1, 'x': "spam"}
'1 named spam'
>>> '%(n).0E => [%(x)-6s]' % dict(n=100, x='spam')
'1E+02 => [spam ]'
>>> '%f, %.2f, %+.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, +0.3333'
```

Синтаксис выражения для форматирования символьных строк

В левой части оператора `%` форматирующего выражения заменяемые адресаты должны иметь приведенный ниже формат. Но последняя составляющая в этом формате является необязательной, а текст за пределами заменяемых адресатов сохраняется в буквальном виде.

`%[(`*`имя_ключа`*`)][`*`признаки`*`][`*`ширина`*`][.`*`точность`*`]`*`код_типа`*

Отдельные составляющие этого синтаксиса заменяемых адресатов в форматирующем выражении обозначают следующее:

имя_ключа

Обозначает элемент предполагаемого словаря, указываемый в круглых скобках.

признаки

Может принимать следующие значения: **-** (выравнивание по левому краю), **+** (числовой знак), **пробел** (перед положительными числами указывается пробел, а перед отрицательными — знак **-**) и **0** (заполнение нулями).

ширина

Общая минимальная ширина поля (для ее автоматической выборки из значений служит знак *****).

точность

Обозначает количество цифр (т.е. точность представления) после десятичной точки **.** (для ее автоматической выборки из значений служит знак *****).

код_типа

Символ из табл. 8.

Составляющие *ширина* и *точность* могут быть обозначены знаком *****, чтобы их значения принудительно выбирались из следующего элемента среди значений в правой части оператора **%**, когда ширина и точность представления чисел неизвестны до самого времени выполнения. *Совет:* формат **%s**, как правило, обозначает преобразование объекта любого типа в его строчное представление для вывода на печать.

Таблица 8. Коды типов, указываемые в левой части оператора % форматирующего выражения

Код	Назначение	Код	Назначение
s	Символьная строка (или любой объект, использующий функцию **str()**)	**X**	Прописная буква **x**
r	То же, что и **s**, но в данном случае используется функция **repr()**, а не **str()**	**e**	Экспонента в представлении чисел с плавающей точкой

Код	Назначение	Код	Назначение
`c`	Символ (типа `int` или `str`)	`E`	Прописная буква `e`
`d`	Десятичное целочисленное значение по основанию `10`	`f`	Десятичное числовое значение с плавающей точкой
`i`	Целочисленное значение	`F`	Прописная буква `f`
`u`	То же, что и `d`, но устаревшее обозначение	`g`	Представление чисел с плавающей точкой в формате `e` или `f`
`o`	Восьмеричное целочисленное значение по основанию `8`	`G`	Представление чисел с плавающей точкой в формате `E` или `F`
`x`	Шестнадцатеричное целочисленное значение по основанию `16`	`%`	Литерал `'%'` (обозначается как `%%`)

Метод форматирования символьных строк

При вызове метода `format()` для форматирования символьных строк происходит то же самое, что и в выражении, упоминавшемся в предыдущем подразделе. Но делается такой вызов с помощью обычного синтаксиса вызова методов для объекта форматирующей строки, где заменяемые адресаты обозначаются синтаксисом `{}` вместо `%`.

В заменяемых адресатах допускается именовать аргументы при вызове метода `format()` по их расположению или наименованию ключевого слова; ссылаться на атрибуты, ключи и смещение аргументов; принимать форматирование по умолчанию или предоставлять явным образом коды типов; а также использовать синтаксис вложения адресатов для извлечения значений из списка аргументов, как показано ниже.

```
>>> 'The knights who say {0}!'.format('Ni')
'The knights who say Ni!'
>>> '{0} {1}, {2:.0f} you'.format(1, 'spam', 4.0)
'1 spam, 4 you'
>>> '{n} named {x:s}'.format(n=1, x="spam")
'1 named spam'
>>> '{n:.0E} => [{x:<6s}]'.format(**dict(n=100, x='spam'))
'1E+02 => [spam ]'
>>> '{:f}, {:.2f}, {:+.{}f}'.format(1/3.0, 1/3.0, 1/3.0, 4)
'0.333333, 0.33, +0.3333'
```

У большинства приложений метода `format()` имеются аналоги в образцах применения исходного выражения с оператором `%`, как показано в предыдущем подразделе (например, в ссылках на ключ в словаре и значение `*`). Хотя в этом методе допускается указывать некоторые операции в самой форматирующей строке, как показано ниже.

```
>>> import sys # Method vs expr: attr, key, index

>>> fmt = '{0.platform} {1[x]} {2[0]}'
>>> fmt.format(sys, dict(x='ham'), 'AB')
'win32 ham A'

>>> fmt = '%s %s %s'
>>> fmt % (sys.platform, dict(x='ham')['x'], 'AB'[0])
'win32 ham A'
```

В версиях Python 2.7 и Python 3.1 знак запятой (`,`), указываемый перед обозначением целого числа или числа с плавающей точкой в составляющей *код_типа*, формально описываемом далее в подразделе "Синтаксис метода форматирования", означает вставку разделителей тысяч, а код типа `%` — форматирование числовых значений, указываемых в процентах, поскольку эти средства отсутствуют в самом форматирующем выражении, но напрямую воспринимаются кодом как повторно используемые функции. Ниже приведены характерные тому примеры.

```
>>> '{0:,d}'.format(1000000)
'1,000,000'
>>> '{0:13,.2f}'.format(1000000)
' 1,000,000.00'
>>> '{0:%} {1:,.2%}'.format(1.23, 1234)
'123.000000% 123,400.00%'
```

Кроме того, в версиях Python 2.7 и Python 3.1 номера полей автоматически нумеруются по порядку, если они пропущены в составляющей *имя_поля*, также описываемой далее в подразделе "Синтаксис метода форматирования". Приведенные ниже строки кода дают один и тот же результат, хотя автоматически нумеруемые поля могут быть менее удобочитаемы при наличии многих полей.

```
>>> '{0}/{1}/{2}'.format('usr', 'home', 'bob')
'usr/home/bob'
>>> '{}/{}/{}'.format('usr', 'home', 'bob') # автоматически
'usr/home/bob'
>>> '%s/%s/%s' % ('usr', 'home', 'bob')      # в выражении
'usr/home/bob'
```

Одиночный объект может быть также отформатирован с помощью встроенной функции format(*объект, спецификация_формата*), как поясняется далее в разделе "Встроенные функции". Эта функция используется методом форматирования символьных строк format(), а ее поведение может быть реализовано в классах с помощью метода перегрузки операторов __format__ (см. далее раздел "Методы перегрузки операторов").

Синтаксис метода форматирования

Заменяемые адресаты в символьных строках служат для вызова метода форматирования и принимают приведенную ниже общую форму. Все четыре части этой формы являются необязательными, но они должны указываться без пробелов, которые приведены ниже для ясности.

{имя_поля составляющая !признак_преобразования
:спецификация_формата}

Отдельные составляющие этого синтаксиса заменяемых адресатов в символьных строках обозначают следующее:

имя_поля

Необязательный номер или ключевое слово, обозначающие аргумент, который может быть пропущен, чтобы использовать относительную нумерацию аргументов, начиная с версий 2.7 и 3.1 соответственно.

составляющая

Символьная строка, не содержащая вообще или же содержащая ссылки *.имя* или [*индекс*], используемые для извлечения атрибутов и индексируемых значений аргумента, который может быть пропущен, чтобы использовать полностью значение аргумента.

признак_преобразования

Внедряется с помощью знака `!`, если таковой имеется, и последующих символов `r`, `s` или `a` для вызова встроенных функций `repr()`, `str()` или `ascii()` соответственно по значению.

спецификация_формата

Внедряется с помощью знака `:`, если таковой имеется, и состоит из текста, обозначающего, каким образом должно быть представлено значение, включая такие подробности, как ширина поля, выравнивание, заполнение, десятичная точность и прочее, а завершается необязательным кодом типа.

Вложенная составляющая *спецификация_формата*, указываемая после двоеточия, имеет собственный синтаксис, формально описываемый ниже, где в квадратных скобках, которые не нужно указывать буквально, представлены необязательные составляющие.

[[*заполнение*]*выравнивание*] [*знак*] [#] [0] [*ширина*] [,]
[.*точность*] [*код_типа*]

В приведенном выше синтаксисе вложенной составляющей *спецификация_формата* отдельные его элементы обозначают следующее:

заполнение

Может быть любым символом заполнения, кроме `{` или `}`.

выравнивание

Может быть знаком `<`, `>`, `=`, или `^` для обозначения выравнивания по левому краю, заполнения после алгебраического знака и выравнивания по центру соответственно.

`,` (*запятая*)

Запрашивает запятую в качестве разделителя тысяч в версиях Python 2.7 и 3.1.

ширина и ***.точность***

Как и в форматирующем выражении `%`, составляющая *спецификация_формата* может также содержать вложенные форматирующие строки только с составляющей *имя_поля*, чтобы принимать значения из списка аргументов в динамическом режиме (аналогично знаку `*` в исходных форматирующих выражениях). Если составляющей *ширина* предшествует нуль, то заполнение нулями может быть выполнено с учетом знака (аналогично составляющей *заполнение*), тогда как знак `#` допускает альтернативное преобразование, если таковое имеется.

код_типа

Мало чем отличается от кодов типа из табл. 8 для исходных форматирующих выражений с оператором `%`, но для метода форматирования имеется дополнительный код типа **`b`**, который служит для представления целочисленных значений в двоичном формате (аналогично встроенной функции `bin()`), а также код типа `%`, используемый для форматирования числовых значений в процентах в версиях Python 2.7 и 3.1, тогда как для обозначения десятичных значений служит код типа **`d`** вместо кодов типа **`i`** и **`u`**, которые больше не применяются.

Следует также иметь в виду, что в отличие от формата **`%s`**, типичного для исходных форматирующих выражений, код типа **`s`** требует наличия строкового объекта в качестве аргумента при вызове метода форматирования, чтобы можно было принимать в общем любой тип данных.

Подстановка шаблонных символьных строк

Начиная с версии Python 2.4, подстановка шаблонных символьных строк предоставляется в качестве альтернативы исходному выражению и методу форматирования символьных строк, описанным в предыдущих подразделах. Для полного форматирования постановка достигается с помощью оператора `%` или вызова метода `str.format()`. Во всех четырех из приведенных ниже примерах возвращается символьная строка `'2: PR5E'`.

```
'%(page)i: %(book)s' % {'page': 2, 'book': 'PR5E'}
'%(page)i: %(book)s' % dict(page=2, book='PR5E')

'{page}: {book}'.format(**dict(page=2, book='PR5E'))
'{page}: {book}'.format(page=2, book='PR5E')
```

Для решения более простых задач подстановки служит класс `Template`, использующий знак **$** в строковом объекте типа `string` для обозначения подстановки:

```
>>> import string
>>> t = string.Template('$page: $book')
>>> t.substitute({'page': 2, 'book': 'PR5E'})
'2: PR5E'
```

Подстановочные значения могут быть предоставлены в качестве именованных аргументов или ключей словаря. Ниже приведены характерные тому примеры.

```
>>> s = string.Template('$who likes $what')
>>> s.substitute(who='bob', what=3.14)
'bob likes 3.14'
>>> s.substitute(dict(who='bob', what='pie'))
'bob likes pie'
```

Метод `safe_substitute()` игнорирует отсутствующие ключи вместо того, чтобы генерировать исключение, как показано ниже.

```
>>> t = string.Template('$page: $book')
>>> t.safe_substitute({'page': 3})
'3: $book'
```

Строковые методы

Помимо описанного ранее метода `format()`, при вызове строковых методов предоставляются средства для обработки текста на более высоком уровне, чем в строковых выражениях. В табл. 9 перечислены вызовы имеющихся строковых методов. В этой таблице ***S*** обозначает любой строковый объект (формально тип `str` в версии 3.X). Строковые методы, видоизменяющие текст, всегда возвращают новую символьную строку и вообще не видоизменяют объект непосредственно, поскольку символьные строки неизменяемы.

Дополнительные сведения о строковых методах, перечисленных в табл. 9, приводятся далее в соответствующих разделах или при вызове метода `help(str.`*`метод`*`)` в диалоговом режиме. *Совет:* перечень этих методов может изменяться в зависимости от версии Python. Для того чтобы вывести собственный перечень строковых методов, попробуйте сделать приведенный ниже вызов. (Шаблонные аналоги некоторых строковых методов описываются далее, в разделе "Модуль `re` сопоставления по шаблонам".)

```
sorted(x for x in dir(str) if not x.startswith('__'))
```

Таблица 9. Вызовы строковых методов в версии Python 3.X

S`.capitalize()`
S`.casefold()`, начиная с версии Python 3.3
S`.center(`*ширина*`, [, `*заполнение*`])`
S`.count(`*подстрока*`[, `*начало*`[, `*конец*`]])`
S`.encode([`*кодировка*`[, `*ошибки*`]])`
S`.endswith(`*суффикс*`[, `*начало*`[, `*конец*`]])`
S`.expandtabs([`*ширина_табуляции*`])`
S`.find(`*подстрока*`[, `*начало*`[, `*конец*`]])`
S`.format(*`*args*`, **`*kwargs*`)`
S`.format_map(`*отображение*`)`, начиная с версии Python 3.2
S`.index(`*подстрока*`[, `*начало*`[, `*конец*`]])`
S`.isalnum()`
S`.isalpha()`
S`.isdecimal()`
S`.isdigit()`
S`.isidentifier()`
S`.islower()`
S`.isnumeric()`
S`.isprintable()`
S`.isspace()`
S`.istitle()`
S`.isupper()`
S`.join(`*итерируемый_объект*`)`
S`.ljust(`*ширина*`[, `*заполнение*`])`
S`.lower()`
S`.lstrip([`*chars*`])`
S`.maketrans(`*x*`[, `*y*`[, `*z*`]])`
S`.partition(`*разделение*`)`

```
S.replace(старая, новая[, количество])
S.rfind(подстрока[, начало[, конец]])
S.rindex(подстрока[, начало[, конец]])
S.rjust(ширина[, заполнение])
S.rpartition(разделитель)
S.rsplit([разделитель[, максимум_разделений]])
S.rstrip([chars])
S.split([разделитель[, максимум_разделений]])
S.splitlines([сохранение_концов_строк])
S.startswith(префикс[, начало[, конец]])
S.strip([chars])
S.swapcase()
S.title()
S.translate(таблица[, удаляемые_символы])
S.upper()
S.zfill(ширина)
```

Строковые методы типа bytes и bytearray

У строковых типов bytes и bytearray в версии Python 3.X имеется ряд таких же методов, как и у типа str из предыдущего подраздела, но их функции не перекрываются, поскольку они иные. В частности, тип str представляет текст в уникоде, тип bytes — неформатированные двоичные данные, а тип bytearray — изменяемые объекты. В приведенных ниже примерах кода, выполняемого в версии Python 3.3, в выражении set(dir(*X*)) - set(dir(*Y*)) вычисляются атрибуты, характерные для аргумента *X*.

```
>>> set(dir(str)) - set(dir(bytes))
{'__rmod__', 'encode', 'isnumeric', 'format',
'isidentifier', 'isprintable', 'isdecimal',
'format_map', '__mod__', 'casefold'}

>>> set(dir(bytes)) - set(dir(str))
{'decode', 'fromhex'}

>>> set(dir(bytearray)) - set(dir(bytes))
{'extend', 'remove', 'insert', 'append', 'pop',
'__iadd__', 'reverse', 'clear', '__imul__',
'copy', '__setitem__', '__alloc__', '__delitem__'}
```

В отношении рассматриваемых здесь строковых типов необходимо иметь в виду следующее.

- В типе `str` не поддерживается *декодирование* в уникоде, поскольку он представляет уже декодированный текст, но допускается *кодирование* в тип `bytes`.
- В типах `bytes` и `bytearray` не поддерживается *кодирование* в уникоде, поскольку они представляют неформатированные байты, в том числе медиатекст и уже закодированный текст, но допускается *декодирование* в тип `str`.
- В типах `bytes` и `bytearray` не поддерживается *форматирование*, реализуемое методом `str.format()` и методами `__mod__` и `__rmod__` в операторе `%`.
- В типе `bytearray` имеются дополнительные методы и операторы непосредственного изменения аналогично функции `list()` (например, метод `append()`, оператор `+=`).

Дополнительные сведения об операциях с байтами символьных строк приведены ниже, в подразделе "Символьные строки типа `bytes` и `bytearray`". Подробнее о моделях строковых типов речь пойдет в разделе "Символьные строки в уникоде", а о вызовах конструкторов типов — в разделе "Встроенные функции"

НА ЗАМЕТКУ

Строковые методы, доступные в версии *Python 2.X*, имеют незначительные отличия (например, метод `decode()` в этой версии отличается от модели строковых типов в уникоде). Строковый тип `unicode` в версии Python 2.X имеет практически такой же интерфейс, как и у объектов типа `str` в этой же версии. Подробнее обо всех этих отличиях можно узнать, обратившись к руководству по библиотеке Python в версии 2.X или же выполнив метод `dir(str)` или `help(str.`*`метод`*`)` в диалоговом режиме.

В последующих подразделах избранные методы из табл. 9 более подробно рассматриваются по их функциональному назначению. Во всех документированных вызовах, возвращающих строковый

результат, этим результатом является новая *символьная строка*, а поскольку символьные строки неизменяемы, то они вообще не видоизменяются непосредственно. Пробелами здесь обозначаются символы пробела, табуляции и конца строки, т.е. все, что определяется константой `string.whitespace`.

Методы поиска в символьных строках

***S*.find(*подстрока* [, *начало* [, *конец*]])**

Возвращает относительную позицию первого вхождения указанной *подстроки* в строку *S* в пределах от *начала* и до *конца* (по умолчанию от **0** до `len(`*S*`)`, т.е. до конца строки). Если искомая подстрока не найдена, то возвращается значение **-1**. *Совет:* см. также оператор принадлежности `in` (в табл. 3), с помощью которого можно проверить принадлежность подстроки символьной строке.

***S*.rfind(*подстрока* [, *начало* [, *конец*]])**

Действует аналогично методу `find()`, но просматривает символьную строку из конца в начало, т.е. справа налево.

***S*.index(*подстрока* [, *начало* [, *конец*]])**

Действует аналогично методу `find()`, но вместо возврата значения **-1** генерирует исключение типа `ValueError`, если искомая подстрока не найдена.

***S*.index(*подстрока* [, *начало* [, *конец*]])**

Действует аналогично методу `find()`, но вместо возврата значения **-1** генерирует исключение типа `ValueError`, если искомая подстрока не найдена.

***S*.rindex(*подстрока* [, *начало* [, *конец*]])**

Действует аналогично методу `rfind()`, но вместо возврата значения **-1** генерирует исключение типа `ValueError`, если искомая подстрока не найдена.

***S*.count(*подстрока* [, *начало* [, *конец*]])**

Подсчитывает количество неперекрывающихся вхождений указанной *подстроки* в строку *S* в пределах от *начала* и до *конца* (по умолчанию от **0** до `len(`*S*`)`, т.е. до конца строки).

S.startswith(*подстрока* [, *начало* [, *конец*]])

Возвращает логическое значение `True`, если символьная строка *S* начинается с указанной *подстроки*. Дополнительные, но не обязательные параметры *начало* и *конец* задают пределы для совпадения с указанной *подстрокой*.

S.endswith(*подстрока* [, *начало* [, *конец*]])

Возвращает логическое значение `True`, если символьная строка *S* оканчивается указанной *подстрокой*. Дополнительные, но не обязательные параметры *начало* и *конец* задают пределы для совпадения с указанной *подстрокой*.

Методы разделения и соединения символьных строк

S.split([*разделитель* [, *максимум_разделений*]])

Возвращает список слов в символьной строке *S*, используя *разделитель* для их разделения. Если задан *максимум_разделений*, то символьная строка разделяется не меньше этого числа раз. Если же *разделитель* не указан или отсутствует (`None`), то в качестве разделителя служит любой пробельный символ. Так, вызов метода `'a*b'.split('*')` дает следующий результат: `['a','b']`. *Совет:* для преобразования символьной строки в список символов (например, `['a','*','b']`) лучше вызвать метод `list(S)`.

S.join(*итерируемый_объект*)

Сцепляет *итерируемый_объект* (например, список или кортеж) символьных строк в единую строку, причем символьная строка *S* вводится между элементами этого объекта. Символьная строка *S* может быть указана пустой (`""`) для преобразования итерируемого объекта символов в строку. Так, вызов метода `'*'.join(['a','b'])` дает следующий результат: `'a*b'`.

S.replace(*старая*, *новая* [, *количество*])

Возвращает копию символьной строки *S* со всеми вхождениями подстроки *старая*, замененными подстрокой *новая*. Если указан параметр *количество*, то заменяется первое *количество* вхождений. Этот метод действует

аналогично сочетанию методов `x=S.split(`*старая*`)` и *новая*`.join(x)`.

`S.splitlines([`*`сохранение_концов_строк`*`]``)`

Разделяет символьную строку *S* по концам строк, возвращая список строк. Если параметр *сохранение_концов_строк* не указан, то символы конца строки в получаемом результате не сохраняются.

Методы форматирования символьных строк

`S.format(`*`*args, **kwargs`***`)`**, **`S.format_map(`***`отображение`***`)`**

См. выше подраздел "Форматирование символьных строк". Начиная с версии Python 3.2, вызов метода `S.format_map(`*M*`)` приводит к такому же результату, что и вызов метода `S.format(**`*M*`)`, хотя отображение *M* не копируется.

`S.capitalize()`

Делает прописной первую букву, оставляя строчными все остальные буквы в символьной строке *S*.

`S.expandtabs([`*`ширина_табуляции`***`])`**

Заменяет символы табуляции пробелами, количество которых определяется параметром *ширина_табуляции* (по умолчанию составляет **8** пробелов), в символьной строке *S*.

`S.strip([`*`chars`***`])`**

Удаляет начальные и конечные пробелы из символьной строки *S* (или же символы, определяемые параметром *chars*, если он передается).

`S.lstrip([`*`chars`***`])`**

Удаляет начальные пробелы из символьной строки *S* (или же символы, определяемые параметром *chars*, если он передается).

`S.rstrip([`*`chars`***`])`**

Удаляет конечные пробелы из символьной строки *S* (или же символы, определяемые параметром *chars*, если он передается).

S.swapcase()

Преобразует все строчные буквы в прописные, и наоборот.

S.upper()

Преобразует все строчные буквы в прописные.

S.lower()

Преобразует все прописные буквы в строчные.

S.casefold()

Начиная с версии Python 3.3, этот метод возвращает версию символьной строки *S* для сравнения без учета регистра. Действует аналогично методу `S.lower()`, но грамотно приводит некоторые символы уникода к нижнему регистру.

S.ljust(*ширина* [, *заполнение*])

Выравнивает по левому краю символьную строку *S* в поле заданной *ширины*, заполняя его справа символами, определяемыми параметром *заполнение* (по умолчанию — пробелами). Аналогичных результатов можно добиться с помощью исходного выражения и метода форматирования символьных строк.

S.rjust(*ширина* [, *заполнение*])

Выравнивает по правому краю символьную строку *S* в поле заданной *ширины*, заполняя его справа символами, определяемыми параметром *заполнение* (по умолчанию — пробелами). Аналогичных результатов можно добиться с помощью исходного выражения и метода форматирования символьных строк.

S.center(*ширина* [, *заполнение*])

Выравнивает по центру символьную строку *S* в поле заданной *ширины*, заполняя его справа символами, определяемыми параметром *заполнение* (по умолчанию — пробелами). Аналогичных результатов можно добиться с помощью исходного выражения и метода форматирования символьных строк.

***S*.zfill(*ширина*)**

Заполняет символьную строку *S* слева нулевыми цифрами, чтобы получить строку заданной *ширины*. Аналогичного результата можно добиться и форматированием символьных строк.

***S*.translate(*таблица* [, *удаляемые_символы*])**

Удаляет из строки *S* все символы, определяемые параметром *удаляемые_символы*, если таковой присутствует, а затем преобразует символы по *таблице* — строке длиной 256 символов, предоставляющее преобразуемое значение каждого символа по его порядковому номеру в строке.

***S*.title()**

Возвращает вариант символьной строки с прописными буквами в отдельных словах.

Методы проверки содержимого символьных строк

***S*.is*()**

Метод `is*()` выполняет логическую проверку символьных строк любой длины. При этом проверяется содержимое символьных строк по различным категориям (и при обнаружении пустых строк всегда возвращается логическое значение `False`).

Модуль обработки исходных символьных строк

Начиная с версии Python 2.0, большинство функций обработки символьных строк, ранее доступных в стандартном модуле `string`, стали доступны в виде методов строковых объектов. Если параметр *X* ссылается на строковый объект, то функция из модуля `string` вызывается следующим образом:

```
import string
res = string.replace(X, 'span', 'spam')
```

Начиная с версии Python 2.0, этот вызов равнозначен вызову следующего строкового метода:

```
res = X.replace('span', 'spam')
```

Форма вызова строковых методов предпочтительнее и действует быстрее, а кроме того, импорт строковым методам не требуется. Следует, однако, иметь в виду, что операция `string.join(`*`итерируемый_объект, ограничитель`*`)` становится методом ограничивающей символьной строки *`ограничитель`*`.join(`*`итерируемый_объект`*`)`. Все эти функции исключены из модуля `string` в версии Python 3.X, и вместо них следует применять эквивалентные методы строковых объектов. Дополнительные сведения о содержимом модуля `string` будут приведены далее, в разделе "Модуль `string`".

Символьные строки в уникоде

Весь текст представлен в уникоде, в том числе и текст, закодированный одним символом на каждый байт, состоящий из 8 битов в коде ASCII. В Python поддерживаются расширенные наборы символов и схемы кодирования в *уникоде* (Unicode) в виде символьных строк, где несколькими байтами можно представить символы оперативной памяти и преобразовать текст из различных кодировок в файлах, и наоборот. В разных версиях Python поддержка кодирования текста в уникоде отличается. Так, в версии Python 3.X весь текст интерпретируется в уникоде, а двоичные данные представляются отдельно, тогда как в версии Python 2.X 8-разрядный текст и данные отличаются от текста в расширенном представлении в уникоде, как поясняется ниже.

В версии Python 3.X

Обычный тип `str` и литерал `'ccc'` представляют весь текст как в 8-разрядном коде, так и в расширенном уникоде. Тип `str` представляет неизменяемую последовательность *символов* — декодируемых кодовых точек в уникоде (порядковых идентификаторов) в оперативной памяти.

Отдельный тип `bytes` и литерал `b'ccc'` представляют байтовые значения двоичных данных, включая медиатекст и текст в уникоде. В частности, тип `bytes` представляет неизменяемую последовательность небольших *целочисленных* (8-разрядных байтовых) значений, но

поддерживает большинство операций типа `str`, а содержимое символьных строк выводится в коде ASCII, когда это возможно. Дополнительный тип `bytearray` является изменяемым вариантом типа `bytes` с дополнительными методами списочного типа для непосредственного внесения изменений.

Кроме того, в версии 3.X в качестве содержимого обычных *файлов*, создаваемых функцией `open()`, подразумеваются объекты типа `str` и `bytes` в текстовом и двоичном режиме соответственно. Так, в текстовом режиме содержимое файлов автоматически кодируется при выводе и декодируется при вводе. Начиная с версии Python 3.3, поддерживается также форма литерала `u'ccc'` в уникоде для обратной совместимости с исходным кодом в версии 2.X. С этой целью в версии 3.X создается объект типа `str`.

В версии Python 2.X

Обычный тип `str` и литерал `'ccc'` представляют байтовые значения как 8-разрядного текста, так и двоичных данных, а отдельный тип `unicode` и литерал `u'ccc'` — кодовые точки символов текста в расширенном представлении в уникоде. Оба упомянутых строковых типа представляют неизменяемые последовательности символов и поддерживают практически одинаковые операции.

Кроме того, в версии 2.X содержимое обычных *файлов*, создаваемых функцией `open()`, является байтовым. А в функции `codecs.open()` поддерживаются операции чтения и записи в файлы текста в уникоде с кодированием и декодированием в процессе передачи данных.

Начиная с версии Python 2.6 поддерживается также байтовый литерал `b'ccc'` для прямой совместимости с исходным кодом в версии 3.X. С этой целью в версии 2.X создается объект типа `str`. И несмотря на доступность изменяемых объектов типа `bytearray`, их поддержка менее характерна для данного типа, чем в версии 3.X.

Поддержка уникода в версии Python 3.X

В версии Python 3.X допускаются также другие символы, а не только в коде ASCII. Такие символы кодируются в строках в шестнадцатеричном виде (**\x**) и экранируются управляющими последовательностями символов в 16- и 32-разрядном уникоде (**\u**, **\U**). Кроме того, в функции `chr()` поддерживается представление символов в уникоде, как показано ниже.

```
>>> 'A\xE4B'
'AäB'
>>> 'A\u00E4B'
'AäB'
>>> 'A\U000000E4B'
'AäB'
>>> chr(0xe4)
'ä'
```

Обычные символьные строки могут кодироваться в неформатированные байты, а те — декодироваться в обычные символьные строки с помощью используемых по умолчанию или явно указываемых кодировок (и дополнительных правил обработки ошибок, как поясняется далее, в разделе "Встроенные функции"). Ниже приведены характерные примеры применения разных кодировок.

```
>>> 'A\xE4B'.encode('latin-1')
b'A\xe4B'
>>> 'A\xE4B'.encode()
b'A\xc3\xa4B'
>>> 'A\xE4B'.encode('utf-8')
b'A\xc3\xa4B'

>>> b'A\xC3\xA4B'.decode('utf-8')
'AäB'
```

Файловые объекты также автоматически кодируют выводимые данные и декодируют вводимые данные в текстовом, но не двоичном режиме. Они принимают наименование кодировки, заменяющей используемую по умолчанию кодировку (см. описание функции `open()` далее, в разделе "Встроенные функции"). Ниже приведены характерные тому примеры.

```
>>> S = 'A\xE4B'
>>> open('uni.txt', 'w', encoding='utf-8').write(S)
3
>>> open('uni.txt', 'rb').read()
b'A\xc3\xa4B'
>>>
>>> open('uni.txt', 'r', encoding='utf-8').read()
'AäB'
```

Начиная с выпуска 3.3 в версии Python 3.X поддерживается также литерал `u'ccc'` в уникоде для обратной совместимости с версией 2.X. Хотя это равнозначно литералу `'ccc'` и приводит к созданию обычной для версии 3.X символьной строки типа `str`.

В обеих версиях, 2.X и 3.X, имеется также возможность встраивать представленное в уникоде содержимое непосредственно в исходные файлы программ. Так, если требуется заменить кодировку UTF-8, используемую по умолчанию в Python, то в первой или второй строке исходного кода указывается следующая строка:

```
# -*- кодировка: latin-1 -*-
```

Символьные строки типа bytes и bytearray

Строковые объекты типа `bytes` и `bytearray` в версии Python 3.X представляют 8-разрядные двоичные данные, включая и текст в уникоде. Они выводятся в текстовом виде в коде ASCII, когда это возможно, а также поддерживают большинство обычных операций со строками типа `str`, включая методы и операции над последовательностями, но не форматирование символьных строк. Ниже приведены характерные тому примеры.

```
>>> B = b'spam'
>>> B
b'spam'
>>> B[0]              # Операции над последовательностями
115
>>> B + b'abc'
b'spamabc'
>>> B.split(b'a')   # Методы
[b'sp', b'm']
```

```
>>> list(B)          # Последовательность значений типа int
[115, 112, 97, 109]
```

Кроме того, в типе bytearray поддерживаются операции над изменяемыми последовательностями списочного типа:

```
>>> BA = bytearray(b'spam')
>>> BA
bytearray(b'spam')
>>> BA[0]
115
>>> BA + b'abc'
bytearray(b'spamabc')
>>> BA[0] = 116                    # Изменчивость
>>> BA.append(115)                 # Списочные методы
>>> BA
bytearray(b'tpams')
```

Формально в обоих типах, bytes и bytearray, поддерживаются *операции над последовательностями* (см. табл. 3), а также характерные для конкретного типа методы, как пояснялось ранее в подразделе "Строковые методы типа bytes и bytearray". Кроме того, в типе bytearray дополнительно поддерживаются *операции над изменяемыми последовательностями* (см. табл. 4). А о вызовах конструкторов типов речь пойдет в разделе "Встроенные функции".

В версиях Python 2.6 и 2.7 имеется тип bytearray, но не тип bytes, а для прямой совместимости с версией 3.X поддерживается литерал b'ccc'. Хотя это равнозначно литералу 'ccc' и приводит к созданию обычной для версии 2.X символьной строки типа str.

Поддержка уникода в версии Python 2.X

В версии Python 2.X символьные строки в уникоде обозначаются в виде литерала u'ccc', что приводит к созданию объекта типа unicode. А в версии Python 3.X для представления данных в уникоде применяется обычный строковый тип и литерал. Произвольные символы в уникоде могут быть представлены и с помощью специальной управляющей последовательности **\u*HHHH***, где ***HHHH*** — шестнадцатеричное число, состоящее из четырех цифр от **0000** до **FFFF**. Для этой цели может быть также

использована традиционная управляющая последовательность **\xHH** и аналогичные последовательности символов до **+01FF** в шестнадцатеричном представлении или до **\777** в восьмеричном представлении.

В типе unicode поддерживаются как строковые методы, так и *операции над последовательностями* (см. табл. 3). В версии Python 2.X допускается совместное использование обычных строковых объектов и их аналогов в уникоде. А сочетание строк 8-разрядных символов с символами в уникоде всегда приводит к их преобразованию в уникод с помощью выбираемой по умолчанию кодировки ASCII (например, объединение символьных строк 'a' + u'bc' дает в итоге строку u'abc'). В операциях над разнотипными данными предполагается, что строка 8-разрядных символов содержит 7-разрядные данные из набора национальных символов U.S. ASCII, и если это другие символы, а не в коде ASCII, то возникает ошибка. С помощью встроенных функций str() и unicode() можно выполнять взаимное преобразование обычных символьных строк и их аналогов в уникоде. А строковые методы encode() и decode() служат для применения и отмены кодировок в уникоде.

К числу доступных модулей и встроенных функций, связанных с поддержкой уникода, относится функция codecs.open(), в которой при обмене данными с файлами выполняется преобразование в уникод и обратно подобно тому, как это делается с содержимым файлов во встроенной функции open() из версии 3.X.

Списки

Списки являются *изменяемыми последовательностями* ссылок на объекты, доступными по *смещению* (позиции).

Списочные литералы и их создание

Списочные литералы обозначаются разделяемыми запятыми последовательностями значений, заключаемых в квадратные скобки. Списки составляются динамически с помощью различных операций, примеры которых приведены ниже.

[]

Пустой список.

[0, 1, 2, 3]

Список из четырех элементов с индексами от **0** до **3**.

L = ['spam', [42, 3.1415], 1.23, {}]

Вложенные подсписки: из элемента списка `L[1][0]` извлекается значение **42**.

L = list('spam')

Создается список со всеми элементами в любом итерируемом объекте путем вызова функции конструктора типов.

L = [x ** 2 for x in range(9)]

Создается список путем накопления результатов вычисления заданного выражения во время итерации (генератор списков).

Операции со списками

К их числу относятся все *операции над последовательностями* (см. табл. 3), все *операции над изменяемыми последовательностями* (см. табл. 4), а также следующие характерные для списков методы, в именах которых *L* обозначает любой объект-список:

***L*.append(*X*)**

Вставляет одиночный объект *X* в конце списка *L*, непосредственно изменяя список.

***L*.extend(*I*)**

Вставляет каждый элемент из любого итерируемого объекта *I* в конце списка *L*, непосредственно изменяя список (как и в операции **+**). Действует аналогично выражению `L[len(L):] = I`. *Совет:* для вставки всех элементов из любого итерируемого объекта *I* в начале списка лучше воспользоваться выражением `L[:0] = I`.

***L*.sort(key=None, reverse=False)**

Сортирует список *L* непосредственно (по умолчанию это делается по возрастанию). Если задан параметр `key`, то он обозначает функцию одного аргумента, используемого для

извлечения или вычисления сравнительного значения из каждого элемента списка. А если задан параметр `reverse`, принимающий логическое значение `True`, то элементы списка сортируются таким образом, как будто сравнение производится в обратном порядке. Например: `L.sort(key=str.lower, reverse=True)`. См. также описание функции `sorted()` ниже, в разделе "Встроенные функции".

`L.reverse()`

Обращает элементы непосредственно в списке *L*. См. также описание функции `reversed()` ниже, в разделе "Встроенные функции".

`L.index(X[, i[, j]])`

Возвращает индекс первого вхождения объекта *X* в список *L*. Генерирует исключение, если этот объект не найден в списке. Это метод поиска. Если ему передается параметр *i*, а возможно, и параметр *j*, то он возвращает наименьший индекс *k*, чтобы `L[k] == X` и `i <= k < j`, где *j* по умолчанию равно длине списка `len(L)`.

`L.insert(i, X)`

Вставляет одиночный объект *X* в список *L* по индексу *i* аналогично выражению `L[i:i] = [X]` как для положительных, так и для отрицательных значений индекса *i*. *Совет:* для вставки в список всех элементов из любого итерируемого объекта *I* по индексу *i* лучше воспользоваться выражением `L[i:i] = I`.

`L.count(X)`

Возвращает количество вхождений объекта *X* в список *L*.

`L.remove(X)`

Удаляет первое вхождение объекта *X* из списка *L*. Генерирует исключение, если этот объект не найден в списке. Действует аналогично выражению `del L[L.index(X)]`.

`L.pop([i])`

Удаляет и возвращает последний элемент из списка *L* (или же элемент по индексу *i*). Этот метод удобно применять

вместе с методом append() для реализации стеков. Действует аналогично строкам кода *x*=L[*i*]; del L[*i*]; return x, где *x* по умолчанию равно **-1** и обозначает последний элемент в списке.

L.clear()

Удаляет все элементы из списка *L*. Доступен только в версии 3.X, начиная с выпуска 3.3.

L.copy()

Создает неполную (верхнего уровня) копию списка *L*. Доступен только в версии 3.X, начиная с выпуска 3.3.

НА ЗАМЕТКУ

В версии *Python 2.X* списочный метод sort() имеет следующую сигнатуру:

L.sort(cmp=None, key=None, reverse=False)

где cmp — функция сравнения, принимающая два аргумента и возвращающая значение, меньшее, равное или большее нуля, для обозначения меньшего, равного или большего результата. Функция сравнения исключена из версии 3.X, поскольку она, как правило, использовалась для преобразования сортируемых значений и обращения порядка сортировки. Поэтому лучше пользоваться остальными двумя аргументами метода sort().

Выражения для генераторов списков

Списочный литерал, заключенный в квадратные скобки ([...]), может быть простым списком выражений или же выражением для генераторов списков в следующей форме:

```
[выражение for адресат1 in итерируемый_объект1 [if условие1]
            for адресат2 in итерируемый_объект2 [if условие2]...
            for адресатN in итерируемый_объектN [if условиеN] ]
```

Генераторы списков составляют списки результатов, накапливая все значения из вычисляемого *выражения* на каждом шаге итерации всех вложенных циклов for, когда в них истинно

заданное *условие*. Все циклы `for` от второго до *N*-го и все выражения проверки *условия* с помощью оператора `if` являются необязательными, а каждое *выражение* и *условие* допускает применение переменных, присваиваемых во вложенных циклах `for`. В версии 2.X имена, привязываемые (т.е. присваиваемые) в генераторе списков, формируются в области действия объемлющего генератора списков, тогда как в версии 3.X они локализованы в области действия отдельного генератора списков. Допускается произвольное вложение генераторов списков.

Генераторы списков действуют аналогично встроенной функции `map()`, но только в версии 3.X. Для отображения результатов генерирования списков функции `map()` требуется функция `list()`, поскольку она выполняет итерацию и сама является итерируемой. А в версии 2.X функция `map()` возвращает список. Ниже приведены характерные примеры применения генераторов списков.

```
>>> [ord(x) for x in 'spam']
[115, 112, 97, 109]
>>> list(map(ord, 'spam')) # использовать функцию list()
из версии 3.X
[115, 112, 97, 109]
```

Тем не менее генераторы списков нередко могут избегать создания временной вспомогательной функции, как показано ниже.

```
>>> [x ** 2 for x in range(5)]
[0, 1, 4, 9, 16]
>>> list(map((lambda x: x ** 2), range(5)))
[0, 1, 4, 9, 16]
```

Генераторы списков с условиями действуют аналогично функции `filter()`, которая также является итерируемой, но только в версии 3.X. Соответствующие примеры приведены ниже.

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]
>>> list(filter((lambda x: x % 2 == 0), range(5)))
[0, 2, 4]
```

А генераторы списков с вложенными циклами `for` действуют аналогично обычному оператору цикла `for`:

```
>>> [x + y for x in range(3) for y in [10, 20, 30]]
[10, 20, 30, 11, 21, 31, 12, 22, 32]

>>> res = []
>>> for x in range(3):
... for y in [10, 20, 30]:
... res.append(x + y)
...
>>> res
[10, 20, 30, 11, 21, 31, 12, 22, 32]
```

Протокол итерации

В *протоколе итерации* определяется ряд объектов и методов, используемых в *контекстах итерации*, включая генераторы, операторы цикла `for` и такие встроенные функции, как `map()` и `filter()`, чтобы автоматически перебрать элементы в коллекциях или результаты, получаемые по требованию. Итерация действует следующим образом.

- Контексты итерации оперируют *итерируемым* объектом с методом `__iter__()`.
- В результате вызова метода `__iter__()` для итерируемого объекта возвращается *итератор* — объект с методом `__next__()` (возможно, тот же самый объект).
- В результате вызова метода `__next__()` для итератора возвращается следующий элемент в итерации или же генерируется исключение типа `StopIteration` в конце итерации.

Кроме того, встроенная функция `iter(X)` вызывает метод `X.__iter__()` итерируемого объекта, а встроенная функция `next(I)` вызывает метод итератора `I.__next__()` как для упрощения циклов ручной итерации, так и для реализации слоя переносимости. Некоторые языковые средства, в том числе встроенная функция `map()` и *выражение-генератор*, одновременно служат в качестве контекста итерации (для их предмета) и итерируемого объекта (для их результатов). См. также предыдущие и последующие подразделы.

Классы могут предоставить метод `__iter__()` для перехвата встроенной функции `iter(X)`. Если этот метод определен,

то в результате его выполнения вызывается метод `__next__()` для перебора результатов в контекстах итерации. А если метод `__iter__()` не определен, то вызывается метод индексирования `__getitem__()` в качестве резервного варианта для итерации вплоть до возникновения исключения типа `IndexError`.

В версии *Python 2.X* метод `I.__next__()` объекта итератора называется `I.next()`, а иначе итерация действует таким же образом. В версиях 2.6 и 2.7 встроенная функция `next(I)` вызывает метод `I.next()` вместо метода `I.__next__()`, что делает его удобным для совместимости как с версией 3.X в версии 2.X, так и для совместимости с версией 2.X в версии 3.X.

Выражения-генераторы

Начиная с версии 2.4, выражения-генераторы достигают эффектов, аналогичных генераторам списков, не генерируя физический список для хранения результатов. В выражениях-генераторах определяется ряд результатов, но не материализуется весь список ради экономии оперативной памяти. Вместо этого в них создается *объект-генератор*, возвращающий по очереди элементы в контекстах итерации, автоматически поддерживая *протокол итерации*, рассматривавшийся в предыдущем подразделе. Соответствующий пример приведен ниже.

```
ords = (ord(x) for x in aString if x not in skipStr)
for o in ords:
    ...
```

Выражения-генераторы являются генераторами, обозначаемыми в круглых, а не в квадратных скобках, иначе они полностью поддерживают синтаксис генераторов списков. При создании итерируемого объекта, передаваемого функции, достаточно воспользоваться круглыми скобками, которыми обозначается функция с единственным аргументом:

```
sum(ord(x) for x in aString)
```

Переменные цикла в выражениях-генераторах (например, переменная *x* в предыдущем примере) недоступны за пределами выражения-генератора как в версии Python 2.X, так и в версии 3.X. В частности, в версии 2.X генераторы списков оставляют

присвоенным последнее значение в переменной цикла, тогда как в версии Python 3.X переменные цикла локализованы в самом выражении во всех формах генераторов.

Для перебора результатов за пределами таких контекстов итерации, как циклы `for`, служит метод `I.__next__()` из протокола итерации в версии 3.X, метод `I.next()` из этого же протокола в версии Python 2.X или же встроенная функция `next()` в обеих версиях Python, 2.X и 3.X, которая вызывает соответствующий метод с учетом переносимости. Если требуется, то для получения сразу всех (остальных) результатов следует вызвать функцию `list()`, поскольку генераторы сами по себе являются итераторами и вызывать их метод `__iter__()` не нужно, да и бесполезно. Ниже приведены характерные тому примеры.

```
>>> squares = (x ** 2 for x in range(5))
>>> squares
<generator object <genexpr> at 0x027C1AF8>
>>> iter(squares) is squares # Вызывать метод __iter__()
необязательно
True
>>> squares.__next__()          # Метод (.next() в версии 2.X)
0
>>> next(squares)               # Встроенная функция (в версиях
2.6+ и 3.X)
1
>>> list(squares)               # Вплоть до исключения
StopIteration
[4, 9, 16]
```

Подробнее о механизме, применяемом в выражениях-генераторах, см. выше, в подразделе "Протокол итерации". А все, что касается *функции-генератора*, которая также создает объект-генератор, см. далее, в разделе "Оператор `yield`".

Другие выражения-генераторы

См. также описание генераторов словарей и множеств в других разделах этой книги, в том числе "Словари" и "Множества". Такие выражения подобны тем, что производят все словари и множества сразу. Они поддерживают синтаксис, аналогичный генераторам списков и выражениям-генераторам, но обозначаются фигурными скобками (`{}`), а генераторы словарей

начинаются с выражения *ключ:значение*, содержащего пару "ключ–значение", как показано ниже.

```
>>> [x * x for x in range(10)]          # Генератор списков
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> (x * x for x in range(10))          # Выражение-генератор
<generator object <genexpr> at 0x009E7328>
>>> {x * x for x in range(10)}          # Множество: 3.X, 2.7
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> {x: x * x for x in range(10)}       # Словарь: 3.X, 2.7
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,
8: 64, 9: 81}
```

Словари

Словари являются *изменяемыми отображениями* ссылок на объекты, доступных по *ключу*, а не по позиции. Они представляют собой неупорядоченные таблицы, в которых ключи отображаются на значения, и реализуются внутренним образом в виде динамически расширяемых хеш-таблиц. В разных версиях Python словари значительно отличаются, как поясняется ниже.

- В версии *Python 2.X* методы `keys()/values()/items()` возвращают списки. Для поиска в словаре имеется метод `has_key()`, а также отдельные итерируемые методы `iterkeys()/itervalues()/ iteritems()`. Кроме того, в версии 2.X словари могут сравниваться непосредственно. В версии Python 2.7 стали доступны генераторы словарей в качестве "заплаты", вставляемой из версии 3.X. А представления в стиле версии 3.X поддерживаются с помощью методов `viewkeys()/viewvalues()/viewitems()`.
- В версии *Python 3.X* методы `keys()/values()/items()` возвращают итерируемые объекты *представлений* вместо списков. Метод `has_key()` исключен в пользу выражений `in`, итерируемые методы из версии Python 2.X — в пользу итерации объекта представления, а словари нельзя сравнивать непосредственно, но это все-таки можно делать, вызывая функцию `sorted(D.items())`. Кроме того, в версии 3.X появилось новое выражение — генератор словарей.

- Кроме того, в версии 3.X объекты *представлений* производят результаты по требованию, сохраняют исходный порядок в словаре, отражают последующие изменения в нем и могут поддерживать операции *установки*. Представления ключей всегда подобны множествам, чего нельзя сказать о представлениях значений. Подобны множествам и представления элементов, если все их элементы однозначны и хешируемы (т.е. изменяемы). Подробнее о выражениях для множеств, которые можно применять к некоторым представлениям, см. далее, в разделе "Множества". Для принудительного генерирования сразу всех результатов (например, для отображения списков или их сортировки методом `L.sort()`) следует вызвать функцию `list()`, передав ей соответствующие представления.

Словарные литералы и их создание

Словарные литералы обозначаются в виде разделяемых запятыми последовательностей пар *`ключ`*`:`*`значение`* в фигурных скобках. Встроенная функция `dict()` поддерживает другие шаблоны создания и генераторы словарей, применяющих итерацию в версиях Python 2.7 и 3.X. В результате присваивания новых ключей формируются новые записи в словарях.

Любой *неизменяемый* объект (например, символьная строка, число, кортеж) может быть ключом словаря. Экземпляры класса также могут быть ключами словарей, если они наследуют методы из протокола хеширования (см. описание метода `__hash__` далее, в разделе "Методы перегрузки операторов"). Кортежные ключи поддерживают составные значения (например, `adict[(M,D,Y)]`, где круглые скобки необязательны). Ниже приведены характерные примеры обращения со словарями.

```
{}
```

Пустой словарь (но не множество).

```
{'spam': 2, 'eggs': 3}
```

Словарь, состоящий из двух элементов: ключей `'spam'` и `'eggs'`, а также значений **2** и **3**.

D = {'info': {42: 1, type(''): 2}, 'spam': []}

Вложенные словари: из элемента `D['info'][42]` извлекается значение **1**.

D = dict(name='Bob', age=45, job=('mgr', 'dev'))

Создается словарь путем передачи именованных аргументов конструктору типов.

D = dict(zip('abc', [1, 2, 3]))

Создается словарь путем передачи кортежных пар "ключ–значение" конструктору типов.

D = dict([['a', 1], ['b', 2], ['c', 3]])

Дает такой же результат, как и в предыдущей строке, но в данном примере принимаются любые итерируемые ключи и значения.

D = {c.upper(): ord(c) for c in 'spam'}

Выражение-генератор словарей (в версиях Python 2.7 и 3.X). Полное описание синтаксиса таких выражений см. выше, в подразделе "Выражения для генераторов списков".

Операции со словарями

К их числу относятся все *операции отображения* (см. табл. 5), а также перечисленные ниже методы, характерные для словарей. Во всех этих методах *D* обозначает любой словарный объект.

***D*.keys()**

Возвращает все ключи из словаря *D*. В версии Python 2.X этот метод возвращает список, а в версии Python 3.X — описанный ранее итерируемый объект представления. В выражении `for K in D` также поддерживается неявная итерация ключей.

***D*.values()**

Возвращает все значения, хранящиеся в словаре *D*. В версии Python 2.X этот метод возвращает список, а в версии Python 3.X — описанный ранее итерируемый объект представления.

D.items()

Возвращает кортежные пары (*ключ, значение*) по одной на каждую запись в словаре *D*. В версии Python 2.X этот метод возвращает список, а в версии Python 3.X — описанный ранее итерируемый объект представления.

D.clear()

Удаляет все элементы из словаря *D*.

D.copy()

Возвращает неполную (верхнего уровня) копию словаря *D*.

D.update(*D2*)

Объединяет все записи непосредственно в словаре *D2* аналогично выражению `for (k, v) in D2.items(): D[k] = v`. Начиная с версии Python 2.4 поддерживаются также итерируемые пары "ключ-значение", а также именованные аргументы (например, `D.update(k1=v1, k2=v2)`).

D.get(*K* [, *default*])

Действует аналогично получению из словаря значения *D*[*K*] по ключу *K*, но возвращает значение *default* (а если оно не указано, то значение None), вместо того чтобы генерировать исключение, если ключ *K* не найден в словаре *D*.

D.setdefault(*K*, [, *default*])

Действует аналогично вызову метода `D.get(K, default)`, но также присваивает ключ *K* значению *default*, если оно не найдено в словаре *D*.

D.popitem()

Удаляет и возвращает произвольную кортежную пару (*ключ, значение*).

D.pop(*K* [, *default*])

Если ключ *K* присутствует в словаре *D*, то возвращает значение *D*[*K*] и удаляет ключ *K*, а иначе — значение *default*, если оно задано, в противном случае генерирует исключение типа KeyError.

dict.fromkeys(*I* [, *значение*])

Создает новый словарь с ключами из итерируемого объекта *I* и значениями, каждое из которых устанавливается равным заданному *значению* (по умолчанию — значение None). Вызывается для экземпляра словаря *D* или по имени типа dict.

Следующие методы доступны только в версии *Python 2.X*:

***D*.has_key(*K*)**

Возвращает логическое значение True, если в словаре *D* имеется ключ *K*, а иначе — логическое значение False. Этот метод равнозначен выражению *K* in *D*, но только в версии Python 2.X. Хотя пользоваться этим методом, как правило, не рекомендуется, поскольку он исключен из версии Python 3.X.

***D*.iteritems(), *D*.iterkeys(), *D*.itervalues()**

Возвращают итерируемые пары значений "ключ-значение", только ключи или только значения. Из версии Python 3.X эти методы исключены, поскольку методы items(), keys() и values() возвращают итерируемые объекты представлений в версии 3.X.

***D*.viewitems(), *D*.viewkeys(), *D*.viewvalues()**

Начиная с версии 2.7, эти методы возвращают итерируемые объекты представлений по парам "ключ-значение", только ключи или только значения, чтобы сымитировать объекты представлений, возвращаемые методами items(), keys() и values() в версии 3.X.

Приведенные ниже операции описаны в табл. 5, но связаны с предыдущими методами.

K* in *D

Возвращает логическое значение True, если в словаре *D* имеется ключ *K*, а иначе — логическое значение False. Заменяет метод has_key() в версии Python 3.X.

for K in D

Перебирает ключи *K* в словаре *D* (во всех контекстах итерации). В словаре поддерживается прямая итерация, когда выражение `for K in D` равнозначно выражению `for K in D.keys()`. В первом выражении используется итератор ключей в объекте словаря. В версии Python 2.X метод `keys()` возвращает новый список, что влечет за собой дополнительные издержки. А в версии Python 3.X этот же метод возвращает итерируемый объект представления вместо физически сохраняемого списка, и благодаря этому обе его формы оказываются равнозначными.

Кортежи

Кортежи являются *неизменяемыми последовательностями* ссылок на объекты, доступных по *смещению* (позиции).

Кортежные литералы и их создание

Кортежные литералы обозначаются как разделяемые запятыми последовательности значений, заключаемые в круглые скобки. Но иногда заключение в круглые скобки можно опустить (например, в заголовках цикла `for` и операторах присваивания =), как показано в приведенных ниже примерах.

()

Пустой кортеж.

(0,)

Одноэлементный кортеж, но не простое выражение.

(0, 1, 2, 3)

Кортеж, состоящий из четырех элементов.

0, 1, 2, 3

Еще один кортеж, состоящий из четырех элементов, как и в предыдущем примере. Такой кортеж недействителен в тех случаях, если имеют значение запятые или круглые скобки (например, при указании аргументов функций и выводе на печать в версии 2.X).

`T = ('spam', (42, 'eggs'))`

Вложенные кортежи: из элемента `T[1][1]` извлекается строка `'eggs'`.

`T = tuple('spam')`

Создается кортеж из всех элементов в любом итерируемом объекте путем вызова функции конструктора типов.

Операции с кортежами

К их числу относятся все *операции над последовательностями* (см. табл. 3), а также приведенные ниже характерные для кортежей методы, начиная с версий Python 2.6 и 3.0 соответственно.

***T*.index(*X* [, *i* [, *j*]])**

Возвращает индекс первого вхождения объекта *X* в кортеж *T*. Генерирует исключение, если этот объект не найден в кортеже. Это метод поиска. Если ему передается параметр *i*, а возможно, и параметр *j*, то он возвращает наименьший индекс *k*, чтобы *T*[*k*] == *X* и *i* <= *k* < *j*, где *j* по умолчанию равно длине кортежа `len(`*T*`)`.

***T*.count(*X*)**

Возвращает количество вхождений объекта *X* в кортеж *T*.

Файлы

Встроенная функция `open()` создает *файловый объект* как наиболее общий интерфейс с внешними файлами. Файловые объекты экспортируют методы передачи данных, описываемые в последующих подразделах, где содержимое файлов представлено в виде символьных строк, определяемых в Python. Приведенный ниже перечень методов является частичным, а что касается вызовов менее употребительных методов и атрибутов, то сведения о них приведены в соответствующих руководствах по Python.

Имя функции `file()` может быть использовано в качестве синонима функции `open()` при создании файлового объекта, но только в версии Python 2.X, и поэтому обычно рекомендуется использовать функцию `open()`. А в версии Python 3.X функция

file() больше недоступна. (Классы из модуля io служат для специальной настройки файлов.)

Подробнее о создании файлов см. описание функции open() далее, в разделе "Встроенные функции". А об отличиях текстовых файлов от двоичных и соответствующих подразумеваемых отличиях строковых типов содержимого в версии Python 3.X см. выше, в разделе "Символьные строки в уникоде".

Языковые средства, связанные с файлами, рассматриваются далее в этой книге. В частности, описание модулей dbm, shelve и pickle см. в разделе "Модули сохраняемости объектов"; описание функций обращения к файлам по дескриптору из модуля os и языковых средств обращения к каталогам по пути из модуля os.path — в разделе "Системный модуль os"; сохранение файлов в формате JSON — в разделе "Модуль json"; а применение базы данных SQL — в разделе "Прикладной интерфейс API базы данных SQL в Python".

О преобразовании сокета в файлоподобный объект см. также описание метода *socketobj*.makefile() в соответствующих руководствах. А о преобразовании символьной строки в файлоподобный объект, совместимый с прикладными программными интерфейсами API, в которых предполагается рассматриваемый здесь интерфейс файлового объекта, см. описание методов o.StringIO(*str*) и io.BytesIO(*bytes*) (или метода StringIO.StringIO(*str*) в версии Python 2.X).

Файлы ввода

infile = open(*имя_файла*, 'r')

Создает объект файла ввода, связанный с именованным внешним файлом. В качестве параметра *имя_файла* обычно указывается символьная строка (например, 'data.txt'), которая преобразуется в путь к текущему рабочему каталогу, если только она не содержит префикс пути к каталогу (например, r'c:\dir\data.txt'). Второй аргумент задает *режим* обращения к файлу: 'r' — для чтения текста, а 'rb' — для чтения двоичных данных без преобразования разрывов строк. Этот режим указывать необязательно, а по умолчанию выбирается режим 'r'. В версии

Python 3.X функция `open()` дополнительно принимает необязательное наименование кодировки в уникоде и текстовом режиме, а в версии 2.X у функции `codecs.open()` имеются аналогичные языковые средства.

infile`.read()`

Читает весь файл, возвращая его содержимое в единственной символьной строке. В текстовом режиме (`'r'`) символы конца строки по умолчанию преобразуются в последовательность символов `'\n'`. А в двоичном режиме (`'rb'`) результирующая символьная строка может содержать непечатаемые символы (например, `'\0'`). В версии Python 3.X текст, читаемый из файла ввода в текстовом режиме, *декодируется* из уникода в символьную строку типа `str`, а в двоичном режиме неизмененное содержимое возвращается в символьной строке типа `bytes`.

infile`.read(`*N*`)`

Читает как минимум еще *N* байтов (`1` или больше байтов) из файла ввода. По достижении конца файла возвращается пустая строка.

infile`.readline()`

Читает следующую строку вплоть до маркера окончания файла. По достижении конца файла возвращается пустая строка.

infile`.readlines()`

Читает весь файл в список считанных строк. См. также следующее далее описание альтернативного варианта применения итератора строк файлового объекта.

`for` *line* `in` *infile*

Использует *итератор строк* файлового объекта *infile* для автоматического перебора строк в файле. Эта форма доступна во всех контекстах итерации, включая циклы `for`, функцию `map()` и генераторы (например, `[line.rstrip() for line in open(`*имя_файла*`)]`). Итерация `for` *line* `in` *infile* дает такой же результат, как

и итерация `for line in infile.readlines()`, но в варианте с итератором строк последние извлекаются по требованию вместо загрузки всего файла в оперативную память, которая в данном случае расходуется более эффективно.

Файлы вывода

***outfile* = open(*имя_файла*, 'w')**

Создает объект файла вывода, связанный с внешним файлом с помощью параметра ***имя_файла***, описанного в предыдущем подразделе. В режиме обращения к файлу `'w'` в него выводится текст, а в режиме `'wb'` — двоичные данные без преобразования разрывов строк. В версии Python 3.X функция `open()` дополнительно принимает необязательное наименование кодировки в уникоде и текстовом режиме, а в версии 2.X у функции `codecs.open()` имеются аналогичные языковые средства.

***outfile*.write(*S*)**

Выводит в файл все содержимое символьной строки *S* без форматирования. В текстовом режиме последовательность символов `'\n'` преобразуется в применяемую по умолчанию на конкретной платформе последовательность символов, обозначающую конец строки. В двоичном режиме исходная символьная строка может содержать непечатаемые байты (например, для вывода символьной строки, состоящей из пяти байтов, два из которых обозначают двоичный нуль, служит последовательность символов `'a\0b\0c'`). В версии Python 3.X для вывода в файл символьных строк типа `str` в текстовом режиме их требуется *кодировать* в уникоде, а в двоичном режиме предполагаются символьные строки типа `bytes`, которые выводятся в неизменном виде.

***outfile*.writelines(*I*)**

Выводит в файл все символьные строки из итерируемого объекта *I*, не добавляя автоматически символы конца строки.

Любые файлы

***file*.close()**

Закрывает файл вручную, освобождая используемые ресурсы, хотя в реализации CPython файлы в настоящее время закрываются автоматически, если они по-прежнему открыты на момент "сборки мусора". См. также описание диспетчера контекста файлового объекта далее, в подразделе "Диспетчеры контекста файлов".

***file*.tell()**

Возвращает текущую позицию в файле.

***file*.seek(*смещение*[, *откуда*])**

Задает текущую позицию в файле по *смещению* для произвольного доступа к его содержимому. Параметр *откуда* может быть равным **0** (смещение от начала файла), **1** (смещение на одну позицию в ту или иную сторону от текущей позиции) или **2** (смещение от конца файла). По умолчанию параметр *откуда* равен **0**.

***file*.isatty()**

Возвращает логическое значение `True`, если файл связан с устройством терминального (интерактивного) типа, а иначе — логическое значение `False`. (В прежних версиях Python может возвращать **0** или **1**.)

***file*.flush()**

Очищает содержимое буфера `stdio` текущего файла, что удобно для обращения с буферизованными конвейерами при чтении данных в другом процессе, а также с файлами, создаваемыми и читаемыми в том же самом процессе.

***file*.truncate([*размер*])**

Укорачивает файл не меньше, чем на количество байтов, определяемое параметром *размер*, а если этот параметр не указан, то до текущей позиции в файле. Имеется не на всех платформах.

***file*.fileno()**

Получает номер файла (целочисленный дескриптор файла). Неточно преобразует файловые объекты в дескрипторы файлов, которые могут быть переданы языковым средствам в модуле os. *Совет:* для более точного преобразования дескриптора файла в файловый объект лучше воспользоваться функцией os.fdopen() или open() в версии 3.X.

Атрибуты файлов (некоторые — только для чтения)

***file*.closed**

Принимает логическое значение True, если файл закрыт.

***file*.mode**

Символьная строка режима обращения к файлу (например, 'r' для чтения), передаваемая функции open().

***file*.name**

Символьная строка с именем, соответствующим внешнему файлу.

Диспетчеры контекста файлов

В стандартной реализации Python (CPython) файловые объекты, как правило, закрываются, если во время "сборки мусора" они все еще открыты. Вследствие этого совсем не обязательно закрывать временные файлы явным образом (например, при вызове функции open('name').read() для чтения файла). Ведь в этом случае файловый объект немедленно освобождается из оперативной памяти и закрывается. А в других реализациях Python (например, Jython) накапливание и закрытие файловых объектов может носить менее определенный характер.

Для того чтобы гарантировать закрытие файла после выхода из кодового блока, независимо от того, генерируется ли в нем исключение, следует воспользоваться операторами try/finally и ручными средствами закрытия файлов, как показано ниже.

```
myfile = open(r'C:\misc\script', 'w')
try:
    ...использовать файл myfile...
```

```
finally:
    myfile.close()
```

С другой стороны, можно воспользоваться операторами `with/as`, доступными в версиях Python 2.X и 3.X, начиная с выпуска 2.6 и 3.0 соответственно:

```
with open(r'C:\misc\script', 'w') as myfile:
    ...использовать файл myfile...
```

В первом случае операторы `try/finally` позволяют ввести вызов функции `close()`, чтобы вовремя закрыть файл. А во втором случае используются *диспетчеры контекста* файлового объекта, гарантирующие автоматическое закрытие файла после выхода из охватываемого кодового блока. Подробнее об операторах `try` и `with` см. ниже, в разделе "Операторы и синтаксис".

Примечания к обращению с файлами

В одних режимах открытия файлов (например, `'r+'`) допускается как ввод, так и вывод в файл, а в других (например, `'rb'`) — передача двоичных данных без преобразования маркеров конца строки и кодировок в уникоде (только в версии Python 3.X). Дополнительно см. описание функции `open()` далее, в разделе "Встроенные функции".

Операции передачи данных в файлы выполняются начиная с текущей позиции в файле. Но при вызове метода `seek()` позиция в файле изменяется для произвольного доступа.

Операции передачи данных в файлы могут выполняться *без* буферизации. Подробнее об этом см. описание аргументов функции `open()` далее, в разделе "Встроенные функции", а также параметра командной строки **`-u`** ранее, в разделе "Параметры командной строки в Python".

Для файлового объекта в версии Python 2.X имеется также метод `xreadlines()`, который действует аналогично автоматическому итератору строк файлового объекта, но исключен из версии Python 3.X в силу своей избыточности.

Множества

Множества являются *изменяемыми* и неупорядоченными коллекциями *однозначных и неизменяемых* объектов. Они поддерживают такие математические операции над множествами, как объединение и пересечение. Множества не являются ни последовательностями, поскольку они неупорядочены, ни отображениями, так как они не отображают значения на ключи. Но в то же время они поддерживают итерацию и действуют во многом таким же образом, как и словари, содержащие только ключи, но не значения.

Литералы множеств и их создание

В версиях 3.X и 2.X множества можно создавать, вызывая встроенную функцию `set()` и передавая ей итерируемый объект, элементы которого становятся членами получающегося в итоге множества. В версиях Python 2.7 и 3.X множества можно также создавать с помощью синтаксиса литералов `{...}` и выражений для генераторов множеств, хотя, вызывая функцию `set()`, можно по-прежнему образовывать пустое множество (или пустой словарь `{}`) и составлять множества из существующих объектов.

Множества изменяемы, но их элементы должны быть неизменяемы. Если же вызвать встроенную функцию `frozenset()`, то можно создать неизменяемое множество, которое может быть вложено в другое множество. Ниже приведены характерные примеры создания множеств.

`set()`

Пустое множество (или пустой словарь `{}`).

```
S = set('spam')
```

Множество, состоящее из четырех элементов: значений `'s'`, `'p'`, `'a'` и `'m'` (принимает любой итерируемый объект).

```
S = {'s', 'p', 'a', 'm'}
```

Множество, состоящее из четырех элементов, как и в предыдущем примере (такой способ создания множеств допускается в версиях Python 2.7 и 3.X):

`S = {ord(c) for c in 'spam'}`

Задается выражение для генератора множества (такой способ создания множеств допускается в версиях Python 2.7 и 3.X). Подробное описание подобного синтаксиса см. выше, в подразделе "Выражения для генераторов списков".

`S = frozenset(range(-5, 5))`

Закрепленное множество, состоящее из 10 целых значений от **-5** до **4**.

Операции над множествами

Ниже описываются наиболее примечательные операции над множествами, где *S*, *S1* и *S2* — любое множество. В большинстве операторов выражений требуется указывать два множества, но аналогичные им методы принимают любой итерируемый объект, обозначаемый далее как *другое* (например, операция `{1, 2} | [2, 3]` не пройдет, тогда как операция `{1, 2}.union([2, 3])` вполне работоспособна). Приведенный ниже перечень далеко не полный, поэтому исчерпывающий перечень имеющихся операций над множествами и соответствующих методов см. в руководстве по библиотеке Python.

x* in *S

Членство в множестве: возвращает логическое значение `True`, если множество *S* содержит объект *x*.

***S1* - *S2*, *S1*.difference(*другое*)**

Разность множеств: новое множество содержит элементы из множества *S1*, но не элементы из множества *S2* (или же *другое*).

***S1* | *S2*, *S1*.union(*другое*)**

Объединение множеств: новое множество содержит элементы из множества *S1* или *S2* (или же *другое*), но без дубликатов.

***S1* & *S2*, *S1*.intersection(*другое*)**

Пересечение множеств: новое множество содержит элементы из обоих множеств *S1* и *S2* (или же *другое*).

`S1 <= S2, S1.issubset(другое)`

Подмножество: проверяет наличие каждого элемента из множества *S1* и в множестве *S2* (или же *другом*).

`S1 >= S2, S1.issuperset(другое)`

Надмножество: проверяет наличие каждого элемента из множества *S2* (или же *другого*) и в множестве *S1*.

`S1 < S2, S1 > S2`

Настоящее подмножество или надмножество: проверяет также, являются ли одинаковыми множества *S1* и *S2*.

`S1 ^ S2, S1.symmetric_difference(другое)`

Симметричная разность множеств: новое множество с элементами из множества *S1* или *S2* (или *другое*), но не из того и другого.

`S1 |= S2, S1.update(другое)`

Обновляет (но не закрепляет множества): вводит в множество *S1* элементы из множества *S2* (или же из *другого*).

`S.add(x), S.remove(x), S.discard(x), S.pop(), S.clear()`

Обновляют, но не закрепляют множество, выполняя следующие операции: ввод и удаление элемента из множества по значению; удаление элемента из множества, если таковой присутствует в нем; удаление и возврат произвольного элемента; удаление всех элементов.

`len(S)`

Возвращает длину множества: количество элементов в множестве.

`for x in S`

Выполняет итерацию элементов множества: действует во всех контекстах итерации.

`S.copy()`

Создает неполную (верхнего уровня) копию множества *S*. Действует аналогично вызову функции `set(S)`.

Другие типы и преобразования

К числу базовых встроенных в Python типов данных относятся описываемые далее *логические* типы; `None` — ложный объект-заполнитель; `NotImplemented` — применяется в методах перегрузки операторов; `Ellipsis` — создается литералом `...` в версии 3.X; *объявляемые* типы — доступны с помощью встроенной функции `type()` и всегда являются классами в Python 3.X; а также *типы программных компонентов*, в том числе функций, модулей и классов (в Python все они являются динамическими объектами высшего порядка).

Логические типы

Логический тип `bool` предоставляет две предопределенные константы, `True` и `False`, введенные во встроенную область действия и доступные с версии 2.3. Для большинства целей эти константы могут интерпретироваться как заранее установленные целые значения **1** и **0** соответственно (например, выражение `True + 3` дает в итоге целое значение **4**). Но в то же время тип `bool` является подклассом целочисленного типа `int` и специально настроен на вывод своих экземпляров по-другому. В частности, логическое значение `True` выводится как `"True"`, а не `"1"` и может быть использовано в качестве встроенного мнемонического имени в логических проверках.

Преобразования типов

В табл. 10 и 11 приведены встроенные языковые средства для преобразования одного типа данных в другой. Все эти средства создают *новые* объекты, а не выполняют преобразования непосредственно. В версии Python 2.X поддерживаются также языковые средства `long(S)` и `'X'` для преобразования в длинные целые и строковые значения соответственно, но они исключены из версии Python 3.X. Подробнее о языковых средствах преобразования типов, перечисленных в табл. 10 и 11, см. выше, в разделе "Числа" и подразделе "Форматирование символьных строк".

Таблица 10. Средства преобразования последовательностей

Средство преобразования	Что преобразует	Во что преобразует
`list(X)`,	Символьная строка, кортеж, любой итерируемый объект	Список
`[n for n in X]`[1]		Кортеж
`tuple(X)`	Символьная строка, список, любой итерируемый объект	
`''.join(X)`	Итерируемый объект, состоящий из символьных строк	Символьная строка

1. Эта форма генератора списков может действовать медленнее, чем функция `list()`*, и поэтому подходит не для всех контекстов преобразования. Такой же результат, как и вызов функции* `list(X)`*, в данном контексте дает вызов функции* `map(None, X)`*, но только в версии Python 2.X, тогда как из версии Python 3.X эта форма функции* `map()` *исключена.*

Таблица 11. Средства преобразования символьных строк и объектов

Средство преобразования	Что преобразует	Во что преобразует
`eval(S)`	Символьная строка	Любой объект, поддерживающий синтаксис выражений
`int(S[, основание])`[1], `float(S)`	Символьная строка или число	Целочисленное значение, числовое значение с плавающей точкой
`repr(X)`, `str(X)`	Любой объект в Python	Символьная строка (функция `repr()` возвращает результат на уровне восприятия кода, а функция `str()` — на уровне восприятия пользователем)
`F%X`, `F.format(X)`, `format(X, [F])`	Объекты с кодами форматов	Символьная строка
`hex(X)`, `oct(X)`, `bin(X)`, `str(X)`	Целочисленные типы	Символьные строки, состоящие из шестнадцатеричных, восьмеричных, двоичных, десятичных цифр
`ord(C)`, `chr(I)`	Символ, целочисленный код	Целочисленный код, символ

1. В версии 2.2 функции преобразования (например, `int()`*,* `float()`*,* `str()`*) служат также в качестве конструкторов классов, а следовательно, подлежат подклассификации. В версии Python 3.X все типы данных являются классами, а все классы — экземплярами класса* `type`*.*

Операторы и синтаксис

В этом разделе описываются правила для синтаксиса и именования переменных.

Правила синтаксиса

Ниже перечислены общие правила написания программ на Python.

Управляющая логика

Операторы выполняются последовательно один за другим, если только в программе не используются операторы управления для ветвления ее кода (например, операторы `if`, `while`, `for`, `raise`, вызовы функций и прочее).

Кодовые блоки

Вложенный кодовый блок выделяется отступом всех его операторов на одну и ту же величину с использованием любого количества символов пробела или табуляции. Вложенный кодовый блок может также появиться в той же самой строке кода, где и заголовок оператора, следуя после символа : этого заголовка, если он состоит только из простых (несоставных) операторов.

Как правило, в заданном кодовом блоке должны быть использованы все символы табуляции *или* все пробелы для отступа. Сочетания и тех и других формально анализируются по следующим двум правилам.

1. Определение количества символов табуляции или пробела для перемещения столбца на очередные 8 символов.
2. Выявление дополнительной несогласованности, считая каждый символ табуляции пробелом.

В версии *Python 2.X* допускаются сочетания символов табуляции и пробела, если они удовлетворяют только первому правилу. Тем не менее сочетание символов табуляции и пробела не рекомендуется, поскольку оно приводит к

ошибкам и усложняет исходный код. А для обозначения сочетаний символов табуляции и пробела, которые считаются несогласованными по второму правилу, служит параметр командной строки `-t` или `-tt` (см. выше раздел "Запуск программ на Python из командной строки"). В версии *Python 3.X* сочетания символов табуляции и пробела по-прежнему допускаются, если они действительны и согласованы как по первому, так и по второму правилу, а иначе всегда возникают ошибки, как и при использовании параметра командной строки `-tt` в версии 2.X.

Например, в версиях 2.X и 3.X внешний кодовый блок располагается с отступом на 2 пробела, 1 символ табуляции и 2 пробела (по первому правилу — 10 символов, по второму — 5), что позволяет расположить внутренний блок с отступом на 1 символ табуляции и 5 пробелов (по первому правилу — 13 символов, по второму — 6). Внутренний блок с отступом на 2 символа табуляции и 1 пробел (по первому правилу — 17 символов, по второму — 3) вполне допустим в версии 2.X по умолчанию (т.е. по первому правилу), тогда как в версии 3.X он не допустим (по второму правилу). Но зачастую сопровождаемый код не должен опираться на эти не совсем ясные правила, поэтому следует использовать либо символы табуляции, либо символы пробела.

Операторы

Оператор оканчивается вместе со строкой кода, но может быть продолжен на нескольких строках, если строка кода завершается знаком `\`, незакрытой парой круглых (`()`), квадратных (`[]`) или фигурных (`{}`) скобок или же символьной строкой в тройных кавычках. Многострочные операторы могут располагаться в одной строке кода, если они разделяются точкой с запятой (`;`).

Комментарии

Комментарии начинаются со знака `#` в любом столбце, но не в строковой константе и продолжаются до конца строки кода. Они игнорируются интерпретатором Python.

Документирующие символьные строки

Если функция, файл модуля или класс начинается со строкового литерала (возможно, после комментариев, начинающихся со знака #), то он сохраняется в атрибуте `__doc__` соответствующего объекта. Подробнее о средствах автоматического извлечения и отображения документирующих символьных строк см. далее, в разделе "Встроенные функции", а также в руководстве по библиотеке Python. *Совет:* в версии Python 3.2 по команде **`python -m pydoc -b`** запускается интерфейс модуля формирования документации *PyDoc* на основе браузера (для перехода в режим ГПИ вместо параметра командной строки **`-b`** из прежних версий лучше указать параметр **`-g`**).

Пробелы

Значение обычно имеют только пробелы слева от исходного кода, где группы кодовых блоков располагаются с отступами. А в остальном пустые строки и пробелы игнорируются и считаются необязательными, кроме тех случаев, когда они служат в качестве разделителей лексем или употребляются в строковых константах.

Правила именования

В этом разделе представлены правила, по которым обозначаются определяемые пользователем имена, главным образом *переменных* в программах.

Формат имени

Структура

Определяемые пользователем имена начинаются с буквы или знака подчеркивания (_), после чего следует любое количество букв, цифр или знаков подчеркивания.

Зарезервированные слова

Определяемые пользователем имена не могут быть такими же, как и любое зарезервированное слово Python из перечисленных в табл. 12.[3]

Учет регистра

Определяемые пользователем имена и зарезервированные слова всегда указываются с учетом регистра. Например, имена *SPAM*, *spam* и *Spam* являются разными.

Неиспользуемые лексемы

В синтаксисе Python знаки `$` и `?` не применяются, хотя допускается их появление в строковых константах и комментариях. Так, в символьных строках знак `$` выполняет особую функцию подстановки шаблона (см. выше подраздел "Подстановка шаблонных символьных строк"), а знаки `$` и `?` — функцию сопоставления с шаблоном (см. ниже раздел "Модуль `re` сопоставления по шаблонам").

Создание

Определяемые пользователем имена создаются путем присваивания, но должны существовать, когда на них делаются ссылки (например, счетчики должны быть явно инициализированы нулями). См. ранее раздел "Операторы и их предшествование" и далее раздел "Правила обозначения пространств имен и областей действия".

Таблица 12. Зарезервированные слова в версии Python 3.X

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

[3] Но это правило может быть ни абсолютным, ни строгим вне реализации CPython. Например, в реализации Jython на основе Java допускается использовать в качестве имен переменных зарезервированные слова в некоторых контекстах.

НА ЗАМЕТКУ

В версии *Python 2.X* слова **print** и **exec** являются зарезервированными, поскольку они принимают форму операторов, а не встроенных функций. Кроме того, в версии *Python 2.X* слова **nonlocal**, **True** и **False** *не* являются зарезервированными. Первое из них вообще недоступно, а два последних являются встроенными именами. Слова **with** и **as** являются зарезервированными в версиях 2.6 и 3.0, но не в более ранних выпусках версии 2.X, если только не активизировать диспетчеры контекста. Слово **yield** зарезервировано, начиная с версии 2.3, но в дальнейшем оно постепенно превратилось из оператора в выражение, хотя и остается по-прежнему зарезервированным.

Соглашения о присваивании имен

- Имена, начинающиеся и оканчивающиеся двумя знаками подчеркивания (например, `__init__`), имеют особое значение для интерпретатора, но считаются зарезервированными словами.
- Имена, начинающиеся с одного знака подчеркивания (например, `_x`) и присваиваемые на верхнем уровне модуля, не копируются в операциях импорта типа `from...*` (см. также описание имен экспорта в атрибуте `__all__` модуля ниже, в разделах "Оператор `from`" и "Псевдозакрытые атрибуты"). А в иных контекстах это неформальное соглашение о внутренних именах.
- Имена, начинающиеся, но не оканчивающиеся двумя знаками подчеркивания (например, `__X`) в операторе `class`, снабжаются префиксами имен объемлющих классов (см. далее раздел "Псевдозакрытые атрибуты").
- Имя, состоящее из единственного знака подчеркивания (`_`), используется в интерактивном операторе (и только в нем) для сохранения результата последнего вычисления.
- Имена встроенных функций и исключений (например, `open`, `SyntaxError`) не являются зарезервированными

словами. Они существуют в обнаруженной последней области действия и могут быть переназначены для сокрытия (так называемого *затенения*) встроенного назначения текущей области действия (например, `open = myfunction`).

- Имена классов обычно начинаются с прописной буквы (например, `MyClass`), а имена модулей — со строчной (например, `mymodule`).
- Первый (крайний слева) аргумент в методе класса обычно называется `self` по очень строгому соглашению.
- Имена модулей разрешаются в соответствии с правилами поиска каталогов по пути. Имена, расположенные ранее в пути, могут скрывать другие аналогичные имена как намеренно, так и ненамеренно (см. также оператор "Оператор `import`").

Конкретные операторы

В последующих разделах описываются все операторы Python. В каждом из этих разделов перечисляются форматы синтаксиса операторов, после чего следуют подробные примеры их применения. Каждое появление *набора* в формате составных операторов означает один или несколько других операторов, возможно, с отступом в виде блока под строкой заголовка. Набор должен быть указан с отступом под заголовком, если он содержит другой составной оператор (`if`, `while` и т.д.). В противном случае он может появиться в той же самой строке кода, где и заголовок оператора. Например, обе приведенные ниже конструкции являются действительными.

```
if x < 42:
  print(x)
  while x: x = x - 1

if x < 42: print(x)
```

В последующих разделах подробно описываются операторы, общие для версий Python 2.X и 3.X. А далее в разделе "Операторы в версии Python 2.X" представлены операторы, доступные только в версии 2.X.

Оператор присваивания

Ниже приведены разные форматы оператора присваивания в Python.

```
адресат = выражение
адресат1 = адресат2 = выражение
адресат1, адресат2 = выражение1, выражение2
адресат1 += выражение

адресат1, адресат2, ... = итерируемый_объект_одинаковой_длины
(адресат1, адресат2, ...) = итерируемый_объект_одинаковой_длины
[адресат1, адресат2, ...] = итерируемый_объект_одинаковой_длины
адресат1, * адресат2, ... = итерируемый_объект_совпадающей_длины
```

Во всех приведенных выше форматах оператора присваивания в адресате сохраняются *ссылки* на объекты. Присваивание в приведенных выше форматах рассматриваемого здесь оператора выполняется по следующим правилам.

- В результате вычисления *выражений* получаются объекты.
- *Адресатами* могут быть простые имена (`X`), уточненные атрибуты (`X.attr`), индексы или нарезки (`X[i]`, `X[i:j:k]`).
- *Переменные* в адресатах не объявляются заранее, но должны быть присвоены до их применения в выражении (см. ранее подраздел "Атомарные члены и динамическая типизация").

Первый из приведенных выше форматов обозначает *элементарное* присваивание. Второй формат обозначает *групповое* присваивание, при котором один и тот же объект, получающийся в результате вычисления заданного выражения, присваивается каждому адресату. Третий формат обозначает присваивание *кортежей*, т.е. попарное присваивание адресатам результатов вычисления выражений слева направо. Четвертый формат обозначает *комбинированное* присваивание, т.е. укороченный вариант операции приращения с присваиванием, как поясняется в следующем подразделе.

Четыре последних формата обозначают присваивание *последовательностей*, при котором составляющие любой последо-

вательности или другого итерируемого объекта присваиваются соответствующим адресатам слева направо. Последовательность или итерируемый объект в правой части оператора присваивания могут быть любого типа, но должны быть одинаковой длины, если только в левой части оператора присваивания не присутствует адресат, обозначаемый знаком звездочки (*), как показано в последнем формате данного оператора. И наконец, последний формат обозначает *расширенное* присваивание *последовательностей*, доступное только в версии Python 3.X и позволяющее накапливать в адресатах, обозначаемых знаком звездочки (*), произвольное количество элементов (см. далее подраздел "Расширенное присваивание последовательностей (только в версии 3.X)").

Присваивание может также выполняться *неявно* в других контекстах, имеющихся в Python (например, при управлении переменными цикла `for` или передаче аргументов функциям). А некоторые форматы оператора присваивания могут применяться где-то в другом месте исходного кода (например, присваивание последовательностей в цикле `for`).

Комбинированное присваивание

В табл. 13 перечислен ряд дополнительно доступных форматов оператора присваивания. Это форматы так называемого *комбинированного присваивания*, при котором неявно выполняется приращение присваиваемого двоичного выражения. Например, два приведенных ниже формата приблизительно равнозначны.

```
X = X + Y
X += Y
```

Таблица 13. Операторы комбинированного присваивания

`X+=Y`	`X&=Y`	`X-=Y`	`X|=Y`
`X*=Y`	`X^=Y`	`X/=Y`	`X>>=Y`
`X%=Y`	`X<<=Y`	`X**=Y`	`X//=Y`

Тем не менее ссылка на адресат *X* во втором формате комбинированного присваивания должна быть вычислена только *один раз*, а *непосредственно* выполняемые операции могут применяться к изменяемым объектам в качестве оптимизации (например, при

присваивании *list1 += list2* автоматически вызывается метод *list1*.extend(*list2*) вместо более медленной операции сцепления, подразумеваемой под знаком **+**). Кроме того, в классах допускается перегружать операторы непосредственного присваивания с помощью методов, имена которых начинаются на `i` (например, метод `__iadd__()` перегружает оператор присваивания **+=**, а метод `__add__()` — оператор сложения **+**). Формат `X //= Y` (целочисленное деление) был внедрен в версии 2.2.

Обычное присваивание последовательностей

В версиях Python 2.X и 3.X любая последовательность значений или другой итерируемый объект могут быть присвоены любой последовательности имен, при условии, что они имеют одинаковую длину. Эта форма элементарного присваивания последовательностей действует в большинстве контекстов присваивания, как показано в приведенных ниже примерах.

```
>>> a, b, c, d = [1, 2, 3, 4]
>>> a, d
(1, 4)

>>> for (a, b, c) in [[1, 2, 3], [4, 5, 6]]:
...     print(a, b, c)
...
1 2 3
4 5 6
```

Расширенное присваивание последовательностей (только в версии 3.X)

Только в версии Python 3.X было расширено присваивание последовательностей , чтобы накапливать произвольное количество элементов, предваряя знаком звездочки одну из переменных в адресате присваивания. Для расширенного присваивания последовательностей их длина не обязательно должна совпадать, а в переменной, помеченной знаком звездочки, в виде нового списка накапливаются все несовпадающие элементы, как показано в приведенных ниже примерах.

```
>>> a, *b = [1, 2, 3, 4]
>>> a, b
```

```
(1, [2, 3, 4])

>>> a, *b, c = (1, 2, 3, 4)
>>> a, b, c
(1, [2, 3], 4)

>>> *a, b = 'spam'
>>> a, b
(['s', 'p', 'a'], 'm')

>>> for (a, *b) in [[1, 2, 3], [4, 5, 6]]:
...     print(a, b)
...
1 [2, 3]
4 [5, 6]
```

ПРИМЕЧАНИЕ

Предполагается ли обобщение форм синтаксиса со знаком звездочки, начиная с версии Python 3.5? В прежних версиях, вплоть до Python 3.3, специальные формы синтаксиса *X и **X могут появляться в следующих трех местах: в *операторах присваивания*, где в переменной *X накапливаются несовпадающие элементы присваиваемых последовательностей; в *заголовках функций*, где в двух формах накапливаются несовпадающие позиционные и именованные аргументы; а также в *вызовах функций*, где в двух формах итерируемые объекты и словари распаковываются в отдельные элементы (аргументы).

В версии Python 3.4 разработчики посчитали, что обобщение форм синтаксиса со знаком звездочки может быть использовано и в *литералах структур данных*, где оно может служить для распаковки коллекций в отдельные элементы подобно тому, как это первоначально делалось при вызове функций. В частности, обобщенную форму синтаксиса со знаком звездочки допускается применять в *кортежах*, *списках*, *множествах* и *генераторах*, как показано в приведенных ниже примерах.

```
[x, *iter] # распаковать элементы объекта iter: списки
(x, *iter), {x, *iter} # сделать то же самое для кортежей и
                       # множеств
{'x': 1, **dict} # распаковать элементы объекта iter: словари
[*iter for iter in x]  # распаковать элементы объекта iter:
                       # генераторы
```

Это служит дополнением к трем первоначальным ролям формы синтаксиса со знаком звездочки в операторах присваивания, заголовках и вызовах функций. Но по мере развития языка Python на применение данной формы синтаксиса со знаком звездочки могут быть наложены определенные ограничения. Так, до появления версии 3.4 подобное изменение в синтаксисе Python было отложено, и его судьба остается до сих пор неопределенной. Оно является предметом дискуссии с 2008 года и вряд ли будет пересмотрено, по крайней мере, до версии Python 3.5, а возможно, и не появится вообще. Поэтому рекомендуется регулярно обращаться за справкой к разделу "What's New in Python" (Что нового в Python) документации на Python по адресу `http://docs.python.org/3/whatsnew/index.html`.

Оператор выражения

Ниже приведен формат оператора выражения в Python.

```
выражение
функция([значение, имя=значение, *имя, **имя...])
объект.метод([значение, имя=значение, *имя, **имя...])
```

Любое выражение может быть указано в виде оператора (например, само по себе в строке кода). С другой стороны, операторы нельзя указывать в любом другом контексте выражений (например, операторы присваивания не имеют никакого результата и не могут быть вложены).

Операторы выражений обычно применяются для вызова функций и методов, не возвращающих никакого полезного значения, а также для вывода на печать в диалоговом режиме. Кроме того, они нередко применяются в выражениях с оператором `yield` и в вызовах встроенной функции `print()` в версии Python 3.X, хотя и

то и другое их применение описывается в этой книге как отдельные операторы.

Синтаксис вызовов

При вызове функций и методов отдельные аргументы разделяются запятыми и, как правило, совпадают по расположению с аргументами в заголовках `def` функций. Кроме того, при вызове функций допускается перечислять имена отдельных аргументов, чтобы принимать передаваемые значения, используя синтаксис *имя=значение* именованных аргументов. В этом случае именованные аргументы совпадают по имени, а не по расположению.

Синтаксис вызовов с произвольным числом аргументов

Специальная форма синтаксиса со знаком звездочки может быть использована для указания списка аргументов при вызовах методов и функций, чтобы *распаковать* коллекции в произвольное количество отдельных аргументов. Так, если аргументы *pargs* и *kargs* указываются в приведенном ниже вызове функции как итерируемый объект и словарь соответственно, то в таком формате функция `f` вызывается с *позиционными* аргументами из итерируемого объекта *pargs*, а также *именованными* аргументами из словаря *kargs*.

```
f(*pargs, **kargs)
```

Ниже приведены характерные примеры подобных вызовов функций.

```
>>> def f(a, b, c, d): print(a, b, c, d)
...

>>> f(*[1, 2], **dict(c=3, d=4))
1 2 3 4
```

Такой синтаксис предназначается как симметричный синтаксису заголовков функций с произвольным количеством аргументов, например `def f(*pargs, **kargs)`, где *накапливаются* несовпадающие аргументы. При вызовах функций элементы, помеченные знаком звездочки, распаковываются в отдельные

аргументы и могут быть объединены с другими позиционными и именованными аргументами по правилам упорядочения (например, `g(1, 2, foo=3, bar=4, *pargs, **kargs)`).

В версии Python 2.X аналогичный результат достигается при вызове встроенной функции `apply()`, как показано ниже. Но эта функция исключена из версии Python 3.X. Дополнительные сведения о синтаксисе вызовов см. ниже, в разделе "Оператор `def`", включая и табл. 15.

```
apply(f, pargs, kargs)
```

Оператор `print`

В версии Python 3.X вывод текста в стандартный поток принимает форму вызова встроенной функции, который обычно программируется в виде оператора выражения, которое указывается, например, само по себе в строке кода. Ниже приведена форма такого вызова.

```
print([значение [,значение]*]
      [, sep=str] [, end=str]
      [, file=объект] [, flush=bool])
```

Каждое *значение* является выражением, при вычислении которого получается объект, выводимый на печать в виде символьной строки, возвращаемой встроенной функцией `str()`. Рассматриваемый здесь вызов настраивается с помощью четырех дополнительных только именованных аргументов, которые вкратце описываются ниже (если эти аргументы опускаются или передаются со значением `None`, то используются значения по умолчанию).

`sep`

Символьная строка, размещаемая между значениями (по умолчанию пробел: `' '`).

`end`

Символьная строка, размещаемая в конце выводимого на печать текста (по умолчанию последовательность символов перевода строки: `'\n'`).

file

Файлоподобный объект, в который выводится текст (по умолчанию стандартный поток вывода: `sys.stdout`).

flush

Принимает логическое значение `True` или `False` (по умолчанию), чтобы активизировать или деактивизировать принудительный вывод в поток (начиная с версии Python 3.3).

Для того чтобы подавить или заменить пробельные разделители и переводы строк, в качестве аргументов `sep` и `end` достаточно передать пустую или специальную символьную строку соответственно. А для того чтобы переадресовать вывод в сценарии, в качестве аргумента `file` достаточно передать файл или файлоподобный объект (дополнительно см. выше раздел "Файлы"). Ниже приведены характерные примеры вывода на печать с помощью оператора `print`.

```
>>> print(2 ** 32, 'spam')
4294967296 spam

>>> print(2 ** 32, 'spam', sep='')
4294967296spam

>>> print(2 ** 32, 'spam', end=' ');  print(1, 2, 3)
4294967296 spam 1 2 3

>>> print(2 ** 32, 'spam', sep='',
...                file=open('out', 'w'))
>>> open('out').read()
'4294967296spam\n'
```

По умолчанию в операциях `print` просто вызывается метод `write()` объекта, обращение к которому происходит в настоящий момент из стандартного потока вывода `sys.stdout`. Так, приведенный ниже фрагмент кода равнозначен вызову функции `print(X)`.

```
import sys
sys.stdout.write(str(X) + '\n')
```

Для того чтобы переадресовать текст, выводимый на печать с помощью оператора `print`, в файл или объект класса, достаточно

передать любой объект с методом write() в качестве именованного аргумента file, как пояснялось ранее, или же переназначить стандартный поток вывода sys.stdout любому подобному объекту, как показано ниже (дополнительно см. ранее раздел "Файлы").

```
sys.stdout = open('log', 'a') # Объект с методом write()
print('Warning-bad spam!')    # Вывод в метод write() данного объекта
```

А поскольку стандартный поток вывода sys.stdout может быть переназначен, то именованный аргумент file строго не требуется. Тем не менее он позволяет нередко избежать как явных вызовов метода write(), так и сохранения и восстановления исходного состояния стандартного потока вывода sys.stdout в переадресуемой операции print, когда исходный поток вывода все же требуется. Дополнительное описание функции print() в версии 3.X см. далее, в разделе "Встроенные функции".

Операторы print в версии Python 2.X

В версии Python 2.X вывод на печать организуется с помощью специального оператора, а не встроенной функции, в следующей форме:

```
print [значение [, значение]* [,]]
print >> файл [, значение [, значение]* [,]]
```

В версии Python 2.X оператор print отображает печатаемое представление каждого *значения* в стандартном потоке вывода (исходно в стандартном потоке вывода sys.stdout) и вводит пробелы между значениями. Завершающая запятая подавляет перевод строки, который обычно вводится в конце списка, что равнозначно использованию аргумента end=' ' в функции вывода на печать из версии Python 3.X, как показано в приведенных ниже примерах.

```
>>> print 2 ** 32, 'spam'
4294967296 spam

>>> print 2 ** 32, 'spam',;  print 1, 2, 3
4294967296 spam 1 2 3
```

Оператор print в версии Python 2.X позволяет также именовать файлоподобный объект вывода в качестве места назначения выводимого на печать текста вместо стандартного потока вывода sys.stdout, как показано ниже.

```
fileobj = open('log', 'a')
print >> fileobj, "Warning-bad spam!"
```

Если файловый объект принимает значение None, то используется стандартный поток вывода sys.stdout. Рассматриваемый здесь синтаксис **>>** в версии Python 2.X равнозначен именованному аргументу file=*F* в версии Python 3.X. Но у оператора вывода на печать в версии Python 2.X отсутствует эквивалент аргумента sep=*S*, хотя строки могут быть предварительно отформатированы и выведены на печать как единый элемент.

Круглые скобки вполне пригодны для употребления в операторе print из версии 2.X, но в то же время они создают кортежи для многих элементов. Для того чтобы воспользоваться этим вариантом оператора print в версии Python 3.X, достаточно ввести приведенную ниже строку кода в начале сценария или в диалоговом режиме. Это можно сделать как в версии 2.X для прямой совместимости с версией 3.X, так и в версии 3.X для обратной совместимости с версией 2.X.

```
from __future__ import print_function
```

Условный оператор if

Ниже приведен формат условного оператора if в Python.

```
if проверка:
    набор
[elif проверка:
    набор]*
[else:
    набор]
```

Условный оператор if выбирает одно из нескольких действий (блоков операторов). Он выполняет набор, связанный с первой проверкой на истину в частях if или elif, или же набор из части else рассматриваемого здесь условного оператора,

если первая проверка даст ложный результат. Части `elif` и `else` условного оператора `if` являются дополнительными, но не обязательными.

Оператор цикла `while`

Ниже приведен формат оператора цикла `while` в Python.

```
while проверка:
    набор
[else:
    набор]
```

Оператор `while` организует типичный цикл, в котором первый набор выполняется до тех пор, пока проверка в начале цикла не даст истинный результат. А в части `else` рассматриваемого здесь оператора выполняется дополнительный набор, если цикл завершится без выполнения оператора `break` в первом наборе.

Оператор цикла `for`

Ниже приведен формат оператора цикла `for` в Python.

```
for адресат in итерируемый_объект:
    набор
[else:
   набор]
```

Оператор цикла `for` представляет собой итерацию последовательности (или другого итерируемого объекта), при которой элементы *итерируемого_объекта* присваиваются *адресату* и для каждого из них (т.е. на каждом шаге цикла) выполняется первый набор. Оператор `for` выполняет дополнительный набор, если цикл завершится без выполнения оператора `break` в первом наборе. В качестве *адресата* может служить все, что допускается указывать в левой части оператора присваивания = (например, `for (x, y) in tuplelist`).

В версии Python 2.2 в рассматриваемом здесь операторе цикла `for` сначала предпринимается попытка получить объект *итератора I* с помощью операции `iter(`*итерируемый_объект*`)`, а

затем вызывать повторно метод *I*.__next__() для этого объекта до тех пор, пока не возникнет исключение типа StopIteration (в версии Python 2.X метод *I*.__next__() называется *I*.next()). Если же объект итератора нельзя получить (например, метод __iter__ не определен), то в рассматриваемом здесь операторе цикла выполняется повторное индексирование *итерируемого_объекта* с последовательным повышением номера индекса до тех пор, пока не возникнет исключение типа IndexError.

Итерация происходит во многих контекстах, имеющихся в Python, включая операторы цикла for, генераторы и функцию map(). Подробнее о списках, применяемых в механизме оператора цикла for и всех остальных контекстах итерации, см. выше, в подразделе "Протокол итерации".

Оператор pass

Ниже приведен формат оператора pass в Python.

pass

Это оператор-заполнитель, не выполняющий никаких действий. Он применяется в тех случаях, когда это требуется в синтаксисе Python (например, для обозначения фиктивного тела функций). Аналогичных результатов можно достигнуть и с помощью многоточия (...), но только в версии Python 3.X.

Оператор break

Ниже приведен формат оператора break в Python.

break

Этот оператор сразу же выполняет выход из объемлющего (внутреннего) оператора цикла while или for, минуя связанный с ним оператор else, если таковой имеется. *Совет:* для выхода из нескольких уровней вложения циклов можно воспользоваться операторами raise и try.

Оператор continue

Ниже приведен формат оператора continue в Python.

continue

Этот оператор осуществляет немедленный переход в верхнюю часть ближайшего объемлющего оператора цикла `while` или `for`. Выполнение кода возобновляется в строке заголовка цикла.

Оператор `del`

Ниже приведены разные форматы оператора `del` в Python.

```
del имя
del имя[i]
del имя[i:j:k]
del имя.атрибут
```

Оператор `del` удаляет переменные, элементы, ключи, нарезки и атрибуты. В первой форме этого оператора *имя* буквально обозначает имя переменной, а в трех остальных формах *имя* может быть любым выражением, в результате вычисления которого получается субъектный объект, указываемый в круглых скобках для обозначения приоритетности. Например: `del a.b()[1].c.d`.

Этот оператор служит, главным образом, для обращения со структурами данных, а не управления памятью. Он также удаляет ссылки на объекты, к которым раньше происходило обращение, что может привести к их "сборке в "мусор" (т.е. освобождению из оперативной памяти), если нигде в коде больше нет ссылок на них. Но "сборка мусора" выполняется автоматически и, как правило, не должна вызываться оператором `del`.

Оператор `def`

Ниже приведен формат оператора `def` в Python.

```
[декорация]
def имя ([arg,... arg=значение,... *arg, **arg]):
    набор
```

Оператор `def` образует новые функции, которые могут также служить методами в классах. Он создает объект функции и присваивает его переменной *имя*. При каждом обращении к объекту функции формируется новая локальная область действия, где присвоенные имена оказываются по умолчанию локальными для вызова функции, если только они не объявлены как `global` или

nonlocal в версии 3.X. Подробнее об областях действия см. далее, в разделе "Правила обозначения пространств имен и областей действия".

Аргументы передаются путем присваивания. В заголовке оператора def они могут быть определены в любом из четырех форматов, перечисленных в табл. 14. Форматы аргументов из табл. 14 могут быть также использованы в вызове функции, где они интерпретируются так, как показано в табл. 15. (Подробнее о синтаксисе вызова функций см. выше, в разделе "Оператор выражения".)

Таблица 14. Форматы аргументов в определениях функций

Формат аргумента	Интерпретация
имя	Должно совпадать по имени или позиции
имя=значение	Значение по умолчанию, если *имя* не передано
**имя*	Накапливает дополнительные позиционные аргументы в виде **имени** нового кортежа
***имя*	Накапливает дополнительные именованные аргументы в виде **имени** нового словаря
**другое, имя [=значение]*	Аргументы, указываемые после знака * (только в версии Python 3.X)
*, *имя* [=*значение*]	То же, что и в предыдущей строке, если иначе не указан знак *

Таблица 15. Форматы аргументов в вызовах функций

Формат аргумента	Интерпретация
значение	Позиционный аргумент
имя=значение	Именованный аргумент (совпадающий по имени)
**итерируемый_объект*	Распаковывает позиционные аргументы из последовательности или другого итерируемого объекта
***словарь*	Распаковывает именованные аргументы из словаря

Только именованные аргументы в версии Python 3.X

В версии Python 3.X (и только в ней) определение функции обобщено, чтобы поддерживать только именованные аргументы, которые должны передаваться по ключевым словам и требуются, если они не программируются значениями по умолчанию. Только именованные аргументы программируются после знака *, где они

могут быть указаны без имени, если это только именованные, а не произвольные позиционные аргументы, как показано в приведенных ниже примерах.

```
>>> def f(a, *b, c): print(a, b, c) # Обязательный именованный
                                    # аргумент c
...
>>> f(1, 2, c=3)
1 (2,) 3

>>> def f(a, *, c=None): print(a, c) # Необязательный именованный
                                     # аргумент c
...
>>> f(1)
1 None
>>> f(1, c='spam')
1 spam
```

Аннотации функций в версии Python 3.X

Только в версии Python 3.X определение функции также обобщено, чтобы аннотировать возвращаемые значения и аргументы значениями объектов для использования в расширениях. Аннотации обозначаются в форме *:значение* после имени аргумента и перед значением по умолчанию или в форме *->значение* после списка аргументов. Они накапливаются в атрибуте `__annotations__` функции, а иначе — не интерпретируются как специальные языковые средства непосредственно в Python, как показано в приведенных ниже примерах.

```
>>> def f(a:99, b:'spam'=None) -> float:
...     print(a, b)
...
>>> f(88)
88 None
>>> f.__annotations__
{'a': 99, 'b': 'spam', 'return': <class 'float'>}
```

Лямбда-выражения

Функции могут быть также созданы в приведенной ниже форме лямбда-выражения типа `lambda`. В этой форме создается новый объект функции, который возвращается для вызова в дальнейшем вместо присваивания по имени.

lambda *arg, arg,...: выражение*

В форме lambda каждый аргумент *arg* оказывается таким же, как и в операторе def (см. табл. 14), а *выражение* — подразумеваемым возвращаемым значением при последующих вызовах. А выполнение кода в *выражении*, по существу, откладывается до момента вызова, как показано ниже.

```
>>> L = lambda a, b=2, *c, **d: [a, b, c, d]
>>> L(1, 2, 3, 4, x=1, y=2)
[1, 2, (3, 4), {'y': 2, 'x': 1}]
```

Форма lambda является выражением, а не оператором, и поэтому она может быть использована в тех местах, где нельзя использовать оператор def (например, в выражении словарного литерала или в списке аргументов при вызове функции). Форма lambda вычисляется как единое выражение вместо выполнения операторов, и поэтому она не предназначена для создания сложных функций (в данном случае лучше воспользоваться оператором def).

Атрибуты функций и значения аргументов по умолчанию

Изменяемые значения аргументов по умолчанию вычисляются один раз при выполнении оператора def, но не при каждом вызове функции, а следовательно, они могут сохранять свое состояние между последовательными вызовами. Тем не менее некоторые считают, что к такому поведению следует относиться осторожно, а для сохранения состояния лучше пользоваться классами и ссылками на объемлющие области действия. Во избежание нежелательных изменений состояния рекомендуется использовать значения по умолчанию None для изменяемых аргументов и явных проверок, как показано в комментариях к приведенному ниже примеру.

```
>>> def grow(a, b=[]): # def grow(a, b=None):
...     b.append(a)    # if b == None: b = []
...     print(b)       # ...
...
>>> grow(1); grow(2)
[1]
[1, 2]
```

В версиях Python 2.X и 3.X поддерживается также присоединение произвольных атрибутов к функциям в качестве еще одной формы сохранения текущего состояния. Хотя атрибуты

поддерживают состояние только по отдельным объектам функций, т.е. по каждому вызову, если при каждом вызове формируется новый объект функции:

```
>>> grow.food = 'spam'
>>> grow.food
'spam'
```

Декораторы функций и методов

Начиная с версии Python 2.4 определения функций можно предварять синтаксисом декорации, описывающим следующую за ним функцию. Объявления, называемые *декораторами*, обозначаются знаком @ и предоставляют явный синтаксис для приемов функционального программирования. Ниже приведен синтаксис декоратора функции.

```
@декоратор
def F():
    ...
```

Этот синтаксис равнозначен следующей повторной привязке имени функции вручную:

```
def F():
    ...
F = декоратор(F)
```

В итоге имя функции повторно привязывается к результату передачи функции через вызываемый объект *декоратор*. Декораторы функций можно использовать для управления функциями или последующими их вызовами с помощью объектов-заместителей. Декораторы можно применять к любому определению функции, включая методы, объявляемые в классе:

```
class C:
    @декоратор
    def M():      # То же, что и M = декоратор(M)
      ...
```

В более общем смысле следующая вложенная декорация:

```
@A
@B
@C
def f(): ...
```

равнозначна приведенному ниже коду без декораторов.

```
def f(): ...
f = A(B(C(f)))
```

Декораторы могут также принимать списки аргументов, как показано ниже.

```
@spam(1, 2, 3)
def f(): ...
```

В этом случае декоратор `spam` должен быть одной функцией, возвращающей другую функцию, т.е. так называемой *фабричной функцией*. Результат ее выполнения используется в качестве отдельного декоратора и может, если требуется, сохранять состояние аргументов. Декораторы должны появляться в строке кода, предшествующей определению функции, а не в той же самой строке, где она объявляется (например, наличие декоратора и определения функции `@A def f(): ...` в одной и той же строке кода не допускается).

Благодаря тому что декораторы принимают и возвращают вызываемые объекты, некоторые встроенные функции, в том числе `property()`, `staticmethod()` и `classmethod()`, могут быть использованы в качестве декораторов других функций (см. далее раздел "Встроенные функции"). Начиная с выпусков 2.6 и 3.0 синтаксис декораторов поддерживается в версиях Python 2.X и 3.X и для классов (см. далее раздел "Оператор `class`").

Оператор `return`

Ниже приведен формат оператора `return` в Python.

`return [`*`выражение`*`]`

Оператор `return` выполняет выход из объемлющей функции и возвращает значение *`выражение`* в результате вызова функции. Если *`выражение`* опущено, то по умолчанию оно подразумевается равным `None`. Это же значение возвращается по умолчанию функциями, выход из которых осуществляется без оператора `return`. *Совет:* если требуется получить от функции результат, состоящий из нескольких значений, его лучше возвратить в виде кортежа.

Описание особой семантики для применения оператора `return` в функции-генераторе см. ниже, в разделе "Оператор `yield`".

Оператор `yield`

Ниже приведены разные форматы оператора `yield` в Python.

```
yield выражение                  # Во всех версиях Python
yield from итерируемый_объект    # Начиная с версии 3.3
```

В версиях 2.X и 3.X в выражении с оператором `yield` определяется *функция-генератор*, которая производит результаты по требованию. Функции, содержащие оператор `yield`, компилируются особым образом. В результате их вызова создается и возвращается *объект-генератор* — итерируемый объект, автоматически поддерживающий *протокол итерации*, чтобы предоставлять результаты в соответствующих контекстах итерации.

Как правило, оператор `yield` обозначается как оператор выражения (например, сам по себе в строке кода), переводит функцию в состояние ожидания и возвращает значение *выражение*. На следующем шаге итерации восстанавливается прежнее местоположение и переменное состояние функции, а управление выполнением кода возобновляется сразу же после оператора `yield`.

Для завершения итерации или всей функции-генератора в целом следует воспользоваться оператором `return`. В прежних версиях по завершении функции-генератора оператор `return` не должен возвращать никакого значения, а начиная с версии 3.3 он может возвращать значение, сохраняемое в качестве атрибута объекта исключения (см. далее подраздел "Изменения в функциях-генераторах, начиная с версии Python 3.3"). Ниже приведен характерный пример применения функции-генератора.

```
def generateSquares(N):
    for i in range(N):
      yield i ** 2

>>> G = generateSquares(5)  # Имеет методы __init__, __next__
>>> list(G)                 # формировать теперь результаты
[0, 1, 4, 9, 16]
```

Когда оператор `yield` используется как выражение (например, `A = yield X`), он возвращает объект, передаваемый методу `send()` генератора в вызывающем коде, и поэтому он должен быть заключен в круглые скобки, если только он не оказывается единственным элементом в правой части оператора присваивания = (например, `A = (yield X) + 42`). В этом режиме значения посылаются генератору в результате вызова метода `send(`*значение*`)`, генератор возобновляет свою работу, а выражение с оператором `yield` возвращает *значение*. Если же вызывается обычный метод `__next__()` или встроенная функция `next()` для продвижения итерации вперед, то оператор `yield` возвращает значение `None`.

У функций-генераторов имеется метод `throw(`*тип*`)`, генерирующий исключение в генераторе при выполнении самого последнего оператора `yield`, а также метод `close()`, генерирующий новое исключение типа `GeneratorExit`, чтобы прекратить итерацию в генераторе. Оператор `yield` является стандартным языковым средством, начиная с версии 2.3, а методы `send()`, `throw()` и `close()` функции-генератора доступны с версии Python 2.5.

Метод `__iter__()` из класса, содержащий оператор `yield`, возвращает генератор с автоматически создаваемым методом `__next__()`. Подробнее о списках, применяемых в механизме функций-генераторов, см. выше, в подразделе "Протокол итерации", а о соответствующих языковых средствах для создания объектов-генераторов — в подразделе "Выражения-генераторы".

Изменения в функциях-генераторах, начиная с версии Python 3.3

В выпуске 3.3 версии Python 3.X (и только в ней) в операторе `yield` стало поддерживаться выражение `from`, которое, по существу, действует аналогично оператору цикла `for`, перебирая элементы итерируемого объекта, как показано ниже. Кроме того, это расширение оператора `yield` позволяет подчиненным генераторам получать посылаемые им, а также генерируемые значения непосредственно из области действия вызывающего кода.

```
for i in range(N): yield i  # Во всех версиях Python
yield from range(N)         # Начиная с версии 3.3
```

Кроме того, начиная с версии 3.3, любое значение в операторе `return` становится доступным в виде атрибута `value` неявно создаваемого экземпляра генерируемого исключения типа `StopIteration`, если функция-генератор прекращает итерацию и явно завершается оператором `return`. Это значение игнорируется при автоматической итерации, но может быть запрошено при ручной итерации или в другом коде, получающем доступ к исключению (см. далее раздел "Встроенные функции"). В версии Python 2.X, а также в версии 3.X до выпуска 3.3 оператор `return` со значением в функции-генераторе интерпретируется как синтаксическая ошибка.

Оператор `global`

Ниже приведен формат оператора `global` в Python.

```
global имя [, имя]*
```

Оператор `global` служит для объявления пространства имен. Когда он применяется в классе или операторе определения функции, каждое *имя*, появляющееся в данном контексте, интерпретируется как ссылка на глобальную (на уровне модуля) переменную под этим именем, независимо от того, присваивается ли *имя* или оно уже существует.

Этот оператор позволяет создавать или изменять глобальные переменные в функции или классе. По правилам, принятым в Python для определения области действия, объявлять нужно только *присваиваемые* глобальные имена, а необъявленные имена становятся локальными, если они присваиваются. Но глобальные ссылки автоматически обнаруживаются в объемлющем модуле. Дополнительно см. далее, в разделе "Правила обозначения пространств имен и областей действия".

Оператор `nonlocal`

Ниже приведен формат оператора `nonlocal` в Python. Этот оператор доступен только в версии Python 3.X.

```
nonlocal имя [, имя]*
```

Оператор nonlocal служит для объявления пространства имен. Когда он применяется во вложенной функции, каждое *имя*, появляющееся в данном контексте, интерпретируется как ссылка на локальную переменную под этим именем в области действия объемлющей функции, независимо от того, присваивается ли *имя* или нет.

Необходимо, чтобы *имя* существовало в объемлющей функции. А изменить его во вложенной функции позволяет оператор nonlocal. По правилам, принятым в Python для определения области действия, объявлять нужно только *присваиваемые* нелокальные имена, а необъявленные имена становятся локальными, если они присваиваются. Но нелокальные ссылки автоматически обнаруживаются в объемлющих функциях. Дополнительно см. далее, в разделе "Правила обозначения пространств имен и областей действия".

Оператор import

Ниже приведен формат оператора import в Python.

```
import [пакет.]* модуль [as имя]
          [, [пакет.]* модуль [as имя]]*
```

Оператор import предоставляет доступ к модулю, импортируя весь модуль в целом. Модули, в свою очередь, содержат имена, извлекаемые путем следующего уточнения: *модуль.атрибут*. В операторах присваивания на самом верхнем уровне исходного файла Python создаются атрибуты объекта модуля. Дополнительное выражение as в операторе import позволяет присвоить переменную *имя* объекту импортируемого модуля и удалить исходное имя *модуль*, что удобно для получения более коротких синонимов длинных имен модулей или путей к пакетам. А дополнительный префикс *пакет* обозначает пути к каталогам пакета, как поясняется в следующем подразделе.

В данном операторе *модуль* обозначает *целевой модуль*, который обычно представляет собой файл с исходным кодом или скомпилированным байт-кодом Python. Кроме того, *модуль* указывается без расширения имени файла (например, **.py**), который,

как правило, должен располагаться в каталоге по пути поиска модулей, если только он не включен в путь *пакет*.

В крайних слева составляющих *модуль* или *пакет* абсолютных путей импорта в качестве *пути поиска модулей* служит переменная окружения `sys.path`, содержащая список имен каталогов, инициализируемый, исходя из каталога верхнего уровня программы, настроек в переменной окружения `PYTHONPATH`, содержимого файла пути с расширением `.pth` и настроек Python по умолчанию. С другой стороны, модули могут располагаться в единственном каталоге для доступа к вложенным компонентам пакета (см. далее подраздел "Импорт пакетов") или относительного импорта в выражениях `from` (см. далее подраздел "Синтаксис относительного импорта пакетов"). А начиная с версии Python 3.3 пути поиска модулей могут произвольно охватывать каталоги для доступа к пакетам пространств имен (см. далее подраздел "Пакеты пространств имен в версии Python 3.3").

Когда модуль импортируется в первый раз, исходный код из его файла компилируется, если требуется, в *байт-код* и сохраняется, если это возможно, в файле с расширением **`.pyc`**, а затем исполняется сверху вниз для формирования атрибутов объекта модуля путем присваивания. В версиях Python 2.X и 3.1 и предшествующих им версиях файлы байт-кода сохраняются в каталоге с файлами исходного кода с тем же самым базовым именем (например, *`модуль.pyc`*). А начиная с версии Python 3.2 байт-код сохраняется в подкаталоге *`__pycache__`* каталога с файлами исходного кода по базовому имени с обозначением номера версии (например, *`модуль.cpython-33.pyc`*).

В дальнейшем во время импорта используется импортированный модуль, но функция `imp.reload()` (или `reload()` в версии 2.X) принудительно импортирует еще раз импортированные ранее модули. Подробнее об импорте по имени символьной строки см. описание функции `__import__()`, используемой в операторе `import`, далее, в разделе "Встроенные функции", а также в документации на функцию `importlib.import_module(`*`имя_модуля`*`)` из стандартной библиотеки.

В стандартной реализации CPython во время импорта могут также загружаться скомпилированные расширения C и C ++

с атрибутами, соответствующими именам из внешних языков программирования. В других реализациях во время импорта могут также именоваться библиотеки классов из других языков программирования (например, в реализации Jython может быть сформирована оболочка для модулей Python, сопрягающихся с библиотекой Java).

Импорт пакетов

Если в именах используется префикс *пакет*, то они обозначают имена объемлющих каталогов, тогда как имена, обозначаемые через точку, отражают иерархию каталогов. Импорт в форме `import` *`каталог1.каталог2.модуль`* обычно служит для загрузки файла модуля из каталога по пути *`каталог1/каталог2/модуль.py`*, где *`каталог1`* должен находиться в каталоге, перечисляемом в пути поиска модулей (переменная окружения `sys.path` служит для абсолютного импорта пакетов), а *`каталог2`* располагается в *`каталоге1`*, а не в переменной `sys.path`.

В обычных пакетах каждый каталог, перечисленный в операторе `import`, должен содержать (желательно пустой) файл *`__init__.py`*, который служит в качестве пространства имен для модулей на уровне каталога. Этот файл выполняется при первом импорте из каталога, а все остальные имена в файле *`__init__.py`* становятся атрибутами объекта модуля из данного каталога. Аналогичные конфликты имен, обусловленные линейным характером переменной окружения `PYTHONPATH`, могут быть разрешены и для пакетов в каталогах.

Подробнее о ссылках внутри пакетов в операторах `from` см. далее, в подразделе "Синтаксис относительного импорта пакетов"). А об альтернативном типе пакетов, не требующем наличия файла *`__init__.py`*, см. в следующем подразделе "Пакеты пространств имен в версии Python 3.3").

Пакеты пространств имен в версии Python 3.3

Начиная с Python 3.3, операции импорта расширены таким образом, чтобы распознавать *пакеты пространств имен*. Это пакеты модулей, которые виртуально соединяют несколько каталогов, включенных в путь поиска модулей.

Пакеты пространств имен не должны (и не могут) содержать файл *__init__.py*. Они служат в качестве резервного варианта для расширения обычных модулей и пакетов, распознаваемых только в том случае, если имя соответствующего компонента не найдено, но совпадающих с одним или больше каталогом, обнаруживаемым при просмотре пути поиска модулей. Это языковое средство активизируется в операторах `import` и `from`.

Алгоритм импорта

Внедрение пакетов пространств имен не изменяет уже сложившийся порядок импорта (например, проверку уже импортированных модулей и файлов байт-кода). Тем не менее поиск модуля расширяется описанным ниже образом.

Во время импорта в Python выполняется перебор каждого каталога, присутствующего в *пути поиска модулей*, который определяется с помощью переменной окружения `sys.path` для крайних слева составляющих абсолютного импорта модулей и по расположению пакета для относительного импорта модулей и составляющим, включенным в пути к пакетам. Помимо поиска импортируемого модуля или пакета *spam* в каждом *каталоге*, указанном в пути поиска модулей, в Python, начиная с версии 3.3, осуществляется также проверка по критериям совпадения в следующем порядке.

1. Если найден файл *каталог\spam__init__.py*, то импортируется и возвращается *обычный пакет*.
2. Если файл *каталог\spam.{py* или другое расширение модуля} не найден, то импортируется и возвращается простой модуль.
3. Если найдена составляющая пути *каталог\spam*, которая оказывается каталогом, то она регистрируется, а просмотр продолжается до следующего каталога в пути поиска.
4. Если же ничего из перечисленного выше не найдено, то просмотр продолжается до следующего каталога в пути поиска.

Если просмотр пути поиска завершается без возврата модуля или пакета на шаге 1 или 2 приведенной выше процедуры и

зарегистрирован, по крайней мере, один *каталог* на шаге 3, то сразу же создается *пакет пространства имен*. У нового пакета пространства имен имеется атрибут `__path__` с установленным итерируемым объектом, состоящим из символьных строк путей к каталогам, обнаруженных и зарегистрированных во время просмотра пути поиска на шаге 3, но отсутствует атрибут `__file__`.

Атрибут `__path__` используется в дальнейшем для поиска всех составляющих пути к пакету всякий раз, когда требуются дополнительные вложенные элементы, и делается это аналогично поиску отдельного каталога с обычным пакетом. Для составляющих пути более низкого уровня этот атрибут выполняет ту же роль, что и переменная окружения `sys.path` для крайних слева составляющих путей абсолютного импорта на верхнем уровне, становясь родительским путем для доступа к элементам низкого уровня по тому же самому четырехэтапному алгоритму импорта.

Оператор `from`

Ниже приведены разные форматы оператора `from` в Python.

```
from [пакет.]* модуль import
             [(] имя [as другое_имя]
             [, имя [as другое_имя]]* [)]

from [пакет.]* модуль import *
```

Оператор `from` импортирует модуль таким же образом, как и оператор `import` (см. предыдущий раздел), но в то же время копирует имена переменных из модуля, чтобы использовать их без следующего уточнения: *атрибут*. Во втором формате данного оператора (`from ... import *`) копируются *все* имена, присвоенные на верхнем уровне модуля, за исключением тех имен, которые начинаются со знака подчеркивания или не перечислены в атрибуте `__all__` модуля со списком символьных строк, если таковой определен.

Если в данном операторе используется выражение `as`, оно создает синоним имени, как и в операторе `import`, воздействуя на любую составляющую *имя*. А если используется *пакет* как

путь импорта, то и он действует таким же образом, как и в операторе `import` (например, `from` *`каталог1.каталог2.модуль`* `import` *X*), для импорта как обычных модулей, так и пакетов пространств имен в версии 3.3., хотя путь к пакету придется перечислить только один раз в самом операторе `from`, а не в каждой ссылке на атрибут. Начиная с версии Python 2.4 имена, импортируемые из модуля, могут быть заключены в круглые скобки, чтобы охватить несколько строк без знаков обратной косой черты (это особый вид синтаксиса, пригодный только для оператора `from`).

В версии Python 3.X форма `from ... import *` рассматриваемого здесь оператора недействительна в функции или классе, поскольку она делает невозможной классификацию областей действия имен во время их определения. По правилам для области действия в формате `*` также формируются предупреждения (начиная с версии 2.2), если этот формат оказывается в функции или классе.

Оператор `from` применяется также для активизации будущих (но пока еще ожидающих утверждения) языковых дополнений в следующем формате: `from __future__ import` *`имя_средства`*. Этот формат должен присутствовать только в самом начале файла модуля (ему могут предшествовать только строки документации или комментарии). С другой стороны, он может быть введен в любой момент и в диалоговом режиме работы.

Синтаксис относительного импорта пакетов

В версиях Python 2.X и 3.X оператор `from` (но *не* `import`) допускает указывать начальные точки в именах модулей для обозначения ссылок на модули в пакете. В этом случае импорт модулей осуществляется *относительно* каталога пакета, в котором находится только импортируемый модуль. *Относительный* импорт ограничивает исходный путь поиска модулей только каталогом, содержащим пакет. А в остальных случаях импорт оказывается *абсолютным*, когда модули находятся в переменной окружения `sys.path`. Ниже приведены общие формы синтаксиса обоих видов импорта.

```
from источник import имя [, имя]*   # Абсолютный импорт:
                                    # переменная sys.path
```

```
from . import модуль [, модуль]*     # Относительный импорт:
                                     # только пакетов
from . источник import имя [, имя]* # Относительный импорт:
                                     # только пакетов

from .. import модуль [, модуль]*    # Родительский
                                     # каталог в пакете
from .. источник import имя [, имя]* # Родительский
                                     # каталог в пакете
```

В приведенных выше формах оператора `from` в качестве *источника* может служить идентификатор или разделяемый точками путь к пакету, *имя* и *модуль* являются простыми идентификаторами, а начальные точки обозначают импорт относительно пакета. Оставшееся расширение `as` для переименования, которое здесь не показано, воздействует на *имя* и *модуль* в этих формах оператора `from` таким же образом, как и в его обычной форме.

Синтаксис с начальными точками служит в версиях Python 2.X и 3.X для того, чтобы сделать импорт модулей явным относительно пакета. Но, в отличие от версии Python3.X, в версии Python 2.X поиск для импорта *без* начальных точек начинается сначала в каталоге самого пакета. Поэтому, начиная с версии Python 2.6, для полной совместимости с версией Python3.X импорт необходимо организовывать следующим образом:

```
from __future__ import absolute_import
```

В силу более широкой применимости путям абсолютного импорта пакетов (относительно каталога в переменной окружения `sys.path`) нередко отдается предпочтение как над неявными формами импорта относительно пакета только в версии Python 2.X, так и над явными формами импорта в версиях Python 2.X и 3.X.

Оператор `class`

Ниже приведен формат оператора `class` в Python.

```
[декорация]
class имя [ ( super [, super]* [, metaclass=M] ) ]:
    набор
```

Оператор `class` образует новые объекты класса, являющиеся фабриками для создания объектов-экземпляров этого класса. Новый объект класса наследует от каждого перечисленного класса *`super`* в заданном порядке и присваивается переменной *имя*. Оператор `class` вносит новую область действия локальных имен, а все имена, присваиваемые в этом операторе, формируют атрибуты объекта класса, общие для всех экземпляров данного класса.

Ниже перечислены наиболее важные языковые средства оператора `class` в частности и классов вообще. Подробнее о классах и объектно-ориентированном программировании см. ниже, в разделах "Объектно-ориентированное программирование" и "Методы перегрузки операторов".

Суперклассы (называемые также базовыми классами), от которых новые классы наследуют атрибуты, перечисляются в заголовке оператора `class`, где они заключаются в круглые скобки (например, `class Sub(Super1, Super2)`).

В результате присваивания в *`наборе`* оператора `class` формируются *атрибуты класса*, наследуемые его экземплярами. В частности, вложенные операторы `def` образуют *методы*, тогда как операторы присваивания создают простые члены класса.

При обращении к классу формируются его *объекты-экземпляры*. У каждого объекта-экземпляра могут быть свои атрибуты. Кроме того, он наследует атрибуты как своего класса, так всех его суперклассов.

Методы-функции получают специальный первый аргумент *`self`*, который по очень строгому соглашению является объектом-экземпляром и подразумеваемым субъектом вызова метода, предоставляющим доступ к атрибутам со сведениями о состоянии экземпляра.

Во встроенных функциях `staticmethod()` и `classmethod()` поддерживаются дополнительные разновидности методов, а в версии Python 3.X методы могут интерпретироваться как простые функции, если они вызываются средствами класса.

Методы *перегрузки операторов*, специально именуемые как *__X__*, перехватывают встроенные операции.

Там, где это гарантируется, классы обеспечивают сохранение состояния и структуру программы, а также поддерживают *повторное использование кода* посредством специальной настройки в новых классах.

Декораторы классов в версиях Python 2.6, 2.7 и 3.0

Начиная с выпусков 2.6 и 3.0 в версиях Python 2.X и 3.X синтаксис декораторов допускается применять не только в определениях функций, но и в операторе `class`. Общая форма синтаксиса декоратора класса

```
@декоратор
class C:
  def meth():
    ...
```

равнозначна следующей повторной привязке имени вручную:

```
class C:
  def meth():
    ...
C = декоратор(C)
```

В итоге имя класса повторно привязывается к результату передачи класса через вызываемый объект *`декоратор`*. Аналогично декораторам функций, декораторы классов могут быть вложенными и поддерживают аргументы декоратора. Декораторы классов могут использоваться для управления классами или последующих обращений к ним для создания экземпляров с помощью объектов-заместителей.

Метаклассы

Метаклассы представляют собой классы, которые обычно являются производными от класса `type` и служат для специальной настройки создания объектов самих классов. Ниже приведен характерный пример метакласса.

```
class Meta(type):
    def __new__(meta, cname, supers, cdict):
```

```
    # Этот метод выполняется вместе с методом
    # __init__ при вызове type.__call__
    c = type.__new__(meta, cname, supers, cdict)
    return c
```

В версии Python 3.X метаклассы определяются в классах с помощью именованных аргументов в заголовке оператора `class` следующим образом:

```
class C(metaclass=Meta): ...
```

А в версии Python 2.X для этой цели служат атрибуты класса:

```
class C(object):
    __metaclass__ = Meta
    ...
```

Код метакласса выполняется по завершении оператора `class` аналогично коду декоратора класса. О преобразовании из операторов `class` в методы метаклассов см. далее, в разделе "Встроенные функции".

Оператор `try`

Ниже приведены разные форматы оператора `try` в Python.

```
try:
    набор
except [тип [as значение]]:      # или [, значение] в
                                 # версии 2.X
    набор
[except [тип [as значение]]:
    набор]*
[else:
    набор]
[finally:
    набор]

try:
    набор
finally:
    набор
```

Оператор `try` перехватывает исключения. В операторах `try` можно указывать выражения `except` с наборами, которые служат

в качестве заголовков для исключений, возникающих во время выполнения набора в операторе try; выражения else, которые вычисляются, если исключение не возникает во время выполнения набора в операторе try; а также выражения finally, которые вычисляются независимо от того, возникает исключение или нет. В выражениях except исключения перехватываются и обрабатываются, а в выражениях finally выполняются завершающие действия (выход из кодового блока).

Исключения могут генерироваться автоматически интерпретатором Python или явно прикладным кодом в операторе raise (см. далее раздел "Оператор raise"). В выражениях except в качестве *типа* указывается выражение, предоставляющее класс перехватываемого исключения, а также дополнительное имя переменной *значение*, которая может быть использована для перехвата экземпляра класса сгенерированного исключения. В табл. 16 перечисляются все выражения, которые могут появиться в операторе try.

Таблица 16. Форматы выражений оператора `try`

Формат выражения	Интерпретация
`except:`	Перехват всех (или всех остальных) исключений
`except` ***тип*****`:`**	Перехват только конкретного исключения
`except` ***тип*** **`as`** ***значение*****`:`**	Перехват исключения и его экземпляра
`except (`*тип1*`, `*тип2*`) :`	Перехват любых исключений
`except (`*тип1*`, `*тип2*`) as `*значение*`:`	Перехват любого исключения и его экземпляра
`else:`	Выполнение этого кодового блока, если исключения не возникают
`finally:`	Выполнение этого кодового блока в любом случае

В операторе try непременно должно присутствовать выражение except, или finally, или и то и другое. Порядок появления этих составляющих в операторе try следующий: try→except→else→finally, где указывать составляющие except и finally необязательно, но должна присутствовать, по крайней мере, составляющая except, если имеется составляющая else. Выражение finally корректно взаимодействует с операторами

return, break и continue, и если любой из этих операторов передает управление блоку try, то после выхода из этого кодового блока выполняются действия в выражении finally.

Ниже приведены типичные разновидности форматов выражений оператора try.

except *имя_класса* as *X*:

Перехват исключения класса и присваивание объекта *X* экземпляру сгенерированного исключения. Объект *X* предоставляет доступ к любым присоединяемым атрибутам со сведениями о состоянии; выводимым на печать символьным строкам или методам, вызываемым для экземпляра сгенерированного исключения. В прежних строковых исключениях объект *X* присваивался дополнительным данным, передававшимся вместе с символьной строкой (строковые исключения были исключены в версиях Python 2.X и 3.X, начиная с выпуска 2.6 и 3.0 соответственно).

except (*тип1*, *тип2*, *тип3*) as *X*:

Перехват любых исключений, именуемых в кортеже, а также присваивание объекту *X* дополнительных данных.

В версии Python 3.X имя *X*, присутствующее в выражении as, *локализуется* в пределах кодового блока except и удаляется при выходе из него. А в версии 2.X это имя не является локальным для данного кодового блока. Подробнее о типичном доступе к классу исключения и его экземпляру (т.е. к *типу* и *значению*) после генерирования данного исключения см. описание функции sys.exc_info() ниже, в разделе "Модуль sys".

Формы оператора try в версии Python 2.X

В версии Python 2.X оператор try действует так, как описано выше, но выражение as, используемое в заголовках except для доступа к экземпляру сгенерированного исключения, обозначается запятой, как показано ниже, причем оба обозначения (as и запятая) доступны в выпусках 2.6 и 2.7 версии 2.X ради совместимости с версией 3.X. А в прежних выпусках версии 2.X обозначение as отсутствовало.

except *имя_класса*, *X*:

Перехват исключения класса и присваивание объекта *X* экземпляру сгенерированного исключения (начиная с версии 2.6 следует использовать обозначение `as`).

except (*имя1*, *имя2*, *имя3*) , *X*:

Перехват любых исключений и присваивание объекта *X* дополнительным данным (начиная с версии 2.6 следует использовать обозначение `as`).

Оператор `raise`

В версии Python 3.X оператор `raise` принимает следующие формы:

```
raise экземпляр [from (другое_исключение | None)]
raise класс [from (другое_исключение | None)]
raise
```

В первой форме генерируется исключение, экземпляр класса которого создается вручную (например, `raise Error(args)`). Во второй форме генерируется исключение и создается новый экземпляр указанного *класса*, что равнозначно выражению `raise class()`. А в третьей форме повторно генерируется самое последнее исключение. Описание необязательного выражения `from` в операторе `raise` см. в следующем далее разделе "Цепочки исключений в версии Python 3.X".

Оператор `raise` инициирует исключения. С его помощью можно явным образом генерировать как встроенные, так и определяемые пользователем исключения. Подробнее об исключениях, предопределенных в Python, см. ниже, в разделе "Встроенные исключения".

В операторе `raise` управление передается совпадающему выражению `except` из самого последнего блока оператора `try`, а в отсутствие такого совпадения — на верхний уровень процесса, где программа завершается и выводится стандартное сообщение об ошибке. Попутно выполняются действия в любых указанных выражениях `finally`. Выражение `except` считается совпавшим, если оно обозначает класс экземпляра сгенерированного исключения

или же любой из его суперклассов (см. далее подраздел "Исключения классов"). Объект-экземпляр генерируемого исключения присваивается переменной as в совпадающем выражении except, если таковое задано.

Цепочки исключений в версии Python 3.X

Только в версии Python 3.X допускается объединение в цепочки исключений в необязательном выражении from, где *другое_исключение* — это класс или экземпляр другого исключения, который присоединяется к атрибуту __cause__ сгенерированного исключения. Если же сгенерированное исключение не перехватывается, то оба исключения выводятся как часть стандартного сообщения об ошибке. Ниже приведен характерный тому пример.

```
try:
    ...
except Exception as E:
    raise TypeError('Bad') from E
```

Начиная с версии Python 3.3 в форме raise from можно также указывать значение None, чтобы отменить любые цепочки исключений, накопленные на момент выполнения данного оператора:

```
raise TypeError('Bad') from None
```

Исключения классов

Начиная с версий Python 2.6 и 3.0 все исключения обозначаются как классы, которые должны быть производными от встроенного класса Exception (в версии 2.X это требуется только для классов нового стиля). В суперклассе Exception предоставляются настройки по умолчанию для отображения символьных строк, а также возможность сохранять аргументы конструктора в кортежном атрибуте args.

В исключениях классов поддерживаются легко расширяемые *категории* исключений. Благодаря тому что в операторах try перехватываются все подклассы указываемого в них суперкласса, категории исключений могут видоизменяться путем изменения ряда подклассов, но без нарушения уже существующих операторов try. Объект-экземпляр генерируемого исключения также обеспечивает сохранение дополнительных сведений об исключении, как показано в приведенном ниже примере.

```
class General(Exception):
    def __init__(self, x):
        self.data = x

class Specific1(General): pass
class Specific2(General): pass

try:
    raise Specific1('spam')
except General as X:
    print(X.data)     # вывести символьную строку 'spam'
```

Формы оператора `raise` в версии Python 2.X

До выпуска 2.6 в версии Python 2.X допускалось обозначать исключения как символьными строками, так и классами. Вследствие этого оператор `raise` в версии Python 2.X может принимать следующие формы, многие из которых существуют ради обратной совместимости.

```
raise строка              # Совпадение с аналогичным
                          # строковым объектом
raise строка, данные      # Присваивание данных переменной
                          # исключения

raise класс, экземпляр    # Совпадение с классом или любым
                          # его суперклассом
raise экземпляр           # равнозначно inst.__class__, inst

raise класс               # равнозначно class()
raise класс, arg          # равнозначно class(arg), noninst
raise класс, (arg [, arg]*) # равнозначно class(arg, arg,...)
raise                     # Повторное генерирование текущего
                          # исключения
```

Начиная в версии 2.5 строковые исключения перестали быть рекомендованными к употреблению, о чем предупреждается особо. В версии Python 2.X допускается указывать и третий элемент в операторах `raise`. Им должен быть объект обратной трассировки стека, используемый вместо текущего местоположения в качестве места, где возникло исключение.

Оператор assert

Ниже приведен формат оператора assert в Python.

```
assert выражение [, сообщение]
```

Оператор assert выполняет проверки при отладке. Если вычисление *выражения* дает ложный результат, то возникает исключение типа AssertionError, которому передается *сообщение* в качестве аргумента конструктора, при условии, что оно предоставляется. Если же утверждения больше не нужны в окончательной версии программы, то в командной строке следует указать параметр -O.

Оператор with

Ниже приведены разные форматы оператора with в Python.

```
with выражение [as переменная]:          # начиная с
      набор                              # версии 3.0/2.6

with выражение [as переменная]
      [, выражение [as переменная]]*: # начиная с
       набор                          # версии 3.1/2.7
```

Оператор with заключает в оболочку вложенные кодовые блоки в описанном ранее диспетчере контекста, где могут выполняться действия при входе в кодовый блок. Этим гарантируется, что действия все равно будут выполнены при выходе из кодового блока, независимо от того, возникнет исключение или нет. Оператор with может стать альтернативой блоку операторов try/finally для выполнения действий при выходе из кодового блока, но только для объектов, имеющих диспетчеры контекста.

Предполагается, что из *выражения* возвращается объект, поддерживающий протокол управления контекстом. Такой объект может также возвращать значение, которое будет присвоено *переменной* при наличии выражения as. В классах могут быть определены специальные диспетчеры контекста, а в некоторых встроенных типах данных вроде файлов и потоков исполнения — предоставлены диспетчеры контекста с завершающими

действиями для закрытия файлов, разблокировки потоков исполнения и прочего, как показано в приведенном ниже примере.

```
with open(r'C:\misc\script', 'w') as myfile:
    ... обработать файл myfile, автоматически закрыть
    ... его при выходе из набора...
```

Подробнее о применении диспетчеров контекста файлов см. выше, в разделе "Файлы". А по поводу других встроенных типов данных, поддерживающих данный оператор и протокол, обращайтесь к соответствующим руководствам по Python.

Оператор `with` поддерживается с версий Python 2.6 и 3.0, а в версии 2.5 он может быть активизирован следующим образом:

```
from __future__ import with_statement
```

Поддержка нескольких диспетчеров контекста в версиях Python 2.7 и 3.1

Начиная с версий Python 2.7 и 3.1 в операторе `with` можно также указывать несколько (*вложенных*) диспетчеров контекста. Любое количество диспетчеров контекста можно указывать через запятую как элементы списка, причем несколько диспетчеров контекста действуют аналогично вложенным операторам `with`. В общем, начиная с версий Python 2.7 и 3.1 следующий фрагмент кода:

```
with A() as a, B() as b:
    ...операторы...
```

равнозначен приведенному ниже фрагменту кода, работоспособному и в версиях 2.6 и 3.0.

```
with A() as a:
    with B() as b:
        ...операторы...
```

Например, в приведенном ниже фрагменте кода оба завершающих действия в файле автоматически выполняются при выходе из блока операторов независимо от последствий возникновения исключения.

```
with open('data') as fin, open('res', 'w') as fout:
    for line in fin:
        fout.write(transform(line))
```

Протокол диспетчеров контекста

Интеграция объектов с оператором `with` осуществляется в соответствии со следующей моделью вызова методов (см. далее раздел "Методы для операций с диспетчерами контекста").

1. В результате вычисления *выражения* получается объект, который называется диспетчером контекста и должен определять имена методов `__enter__` и `__exit__`.
2. Вызывается метод `__enter__()` из диспетчера контекста. Значение, возвращаемое этим методом, присваивается *переменной*, если таковая имеется, а в противном случае — отвергается.
3. Выполняется код во вложенном *наборе*.
4. Если в *наборе* генерируется исключение, вызывается метод `__exit__(`*тип, значение, объект обратной трассировки стека*`)` с указанием подробностей исключения. Если этот метод возвращает логическое значение `False`, то генерируется исключение, а иначе действие исключения прекращается.
5. Если же исключение *не* генерируется в *наборе*, метод `__exit__` по-прежнему вызывается, но в качестве трех его аргументов передается значение `None`.

Операторы в версии Python 2.X

В версии Python 2.X поддерживается описанный ранее оператор `print`, но не оператор `nonlocal`, тогда как оператор `with` полностью поддерживается, лишь начиная с версии 2.6. Кроме того, синтаксис операторов `raise`, `try` и `def` несколько отличается в Python 2.X, как отмечалось ранее. А упомянутая ранее семантика, характерная для версии 3.X, как правило, не распространяется на версию 2.X (это, например, относится к пакетам пространств имен).

Ниже приведен дополнительный оператор, доступный только в версии Python 2.X.

exec *кодовая_строка* [**in** *глобальный_словарь* [, *локальный_словарь*]]

Оператор exec выполняет код в динамическом режиме. В качестве *кодовой_строки* может быть указан любой оператор Python (или несколько операторов, разделяемых символами новой строки) в формате символьной строки, которая компилируется и выполняется в пространстве имен, содержащем оператор exec, или же в глобальном либо локальном словаре пространства имен, если таковые указаны (по умолчанию — *глобальный_словарь*). В качестве *кодовой_строки* может быть также указан объект скомпилированного кода. Дополнительно см. описание функций compile(), eval() и execfile() (только в версии Python 2.X) далее, в разделе "Встроенные функции".

В версии Python 3.X этот оператор становится функцией exec() (см. далее раздел "Встроенные функции"). Кроме того, в версии Python 2.X допускается синтаксис exec(a, b, c) ради обратной и прямой совместимости. *Совет:* этим синтаксисом не стоит пользоваться для вычисления символьных строк ненадежного кода, поскольку они могут содержать зловредный исполняемый код.

Правила обозначения пространств имен и областей действия

В этом разделе рассматриваются правила привязки и поиска имен (дополнительно см. выше подразделы "Формат имени", "Соглашения о присваивании имен" и "Атомарные члены и динамическая типизация"). В любом случае имена создаются при первом присваивании, хотя они должны уже существовать, когда на них делаются ссылки. Уточненные и неуточненные имена разрешаются по-разному.

Уточненные имена: пространства имен объектов

Уточненные имена (например, *X* в выражении *объект.X*) называются иначе *атрибутами* и существуют в пространстве имен объекта. Присваивание таких имен в некоторых лексических

областях действия[4] служит для инициализации пространств имен объектов (например, атрибутов модуля или класса), как поясняется ниже.

Присваивание: ***объект.X = значение***

Создается или изменяется имя атрибута *X* в пространстве имен *объекта*. Это обычный случай (подробнее об этом см. далее раздел "Формальные правила наследования").

Ссылка: ***объект.X***

Осуществляется поиск имени атрибута *X* сначала в *объекте*, а затем экземпляров во всех доступных классах выше по иерархии. Это и есть определение *наследования* (подробнее об этом см. далее раздел "Формальные правила наследования").

Неуточненные имена: лексические области действия

Неуточненные имена (например, *X* в начале выражения) подчиняются правилам определения лексической области действия, как поясняется ниже. При присваивании такие имена привязываются к локальной области действия, если только они не объявлены глобальными или нелокальными в версии 3.X.

Присваивание: ***X = значение***

Имя *X* делается локальным по умолчанию. В частности, имя *X* создается или изменяется в текущей локальной области действия. Если же имя *X* объявляется как `global`, то оно создается или изменяется в области действия объемлющего модуля. А если имя *X* объявляется как `global` (только в Python 3.X), то оно изменяется в области действия объемлющей функции. Локальные переменные, как правило, хранятся в стеке вызовов для быстрого доступа во время выполнения и непосредственно доступны только в коде, находящемся в той же самой области действия.

[4] Лексические области действия обозначают физически (синтаксически) вложенные структуры кода в исходном коде программы.

Ссылка: **X**

Осуществляется поиск имени *X* как минимум в четырех категориях областей действия в следующем порядке:

а) Текущая *локальная* область действия (внутренняя объемлющая функция).

б) Локальные области действия всех лексически *объемлющих функций* (других уровней функций от внутреннего до внешнего).

в) Текущая *глобальная* область действия (объемлющий модуль).

г) *Встроенная* область действия (соответствующая модулю `builtins` в версии Python 3.X и модулю `__builtin__` в версии Python 2.X).

Контексты локальной и глобальной областей действия определены в табл. 17. Объявления `global` инициируют поиск в глобальной области действия, тогда как объявления `nonlocal` ограничивают поиск объемлющими функциями в версии 3.X.

Особые случаи: генераторы, исключения

В версии Python 3.X переменные цикла локализуются во всех *генераторах* (то же самое происходит и в версии Python 2.X, но только для генераторов списков). Кроме того, в версии Python 3.X локализуется и удаляется переменная исключения, указанная в выражении `except` оператора `try` (а в версии 2.X она не локализуется). Дополнительно см. выше подраздел "Выражения для генераторов списков" и раздел "Оператор `try`".

Таблица 17. Области действия неуточненных имен

Контекст кода	Глобальная область действия	Локальная область действия
Модуль	То же, что и в локальной области действия	Сам модуль
Функция, метод	Объемлющий модуль	Определение или вызов функции
Класс	Объемлющий модуль	Оператор `class`

Контекст кода	Глобальная область действия	Локальная область действия
Сценарий, диалоговый режим работы	То же, что и в локальной области действия	Модуль `__main__`
`exec()`, `eval()`	Глобальная (или передаваемая) область действия вызывающего кода	Локальная (или передаваемая) область действия вызывающего кода

Вложенные области действия и замыкания

Поиск в *объемлющих функциях*, упоминавшихся в п.b) правил "Ссылка" предыдущего раздела, происходит в *статически вложенной области действия* и сделан стандартным в версии 2.2. Например, приведенная ниже функция вполне работоспособна, поскольку ссылка на переменную `x` в теле функции `f2()` доступна в области действия объемлющей функции `f1()`.

```
def f1():
  x = 42
  def f2():
     print(x) # переменная x сохраняется в области действия
              # функции f1()
  return f2   # вызывается в дальнейшем: f1()()=>42
```

Вложенные функции, сохраняющие ссылки на объемлющую область действия (например, функция `f2()` в приведенном выше примере кода), называются *замыканиями* — языковыми средствами сохранения, которые иногда оказываются альтернативой или дополнением классов и стали более полезными с внедрением оператора `nonlocal` в версии 3.X (см. выше раздел "Оператор `nonlocal`"). Вложение областей действия происходит произвольно, но поиск осуществляется только в объемлющих функциях (но не классах), как показано в приведенном ниже примере.

```
def f1():
  x = 42
  def f2():
    def f3():
      print(x) # находит переменную x в области действия
```

```
                # функции f1()
    f3()        # функция f1() выводит значение 42
  f2()
```

Объемлющие области действия и аргументы по умолчанию

До версии Python 2.2 упомянутые выше функции не могли бы действовать нормально, поскольку переменная x не является локальной (в области действия вложенной функции), глобальной (в объемлющей функции f1() из модуля) или встроенной. Для того чтобы сделать такие функции работоспособными до версии 2.2 или по мере надобности, в *аргументах по умолчанию* сохраняются значения из непосредственно объемлющей области действия, поскольку эти значения вычисляются до выполнения оператора def, как показано ниже.

```
def f1():
  x = 42
  def f2(x=x):
    print(x) # функция f1() выводит значение 42
  return f2
```

Данный прием оказывается вполне работоспособным и в последних версиях Python. Он применяется и в лямбда-выражениях, где подразумевается вложенная область действия, как и в операторе def. На практике лямбда-выражения чаще всего оказываются вложенными, как показано в приведенных ниже примерах.

```
def func(x):
  action = (lambda n: x ** n) # используется, начиная с
                              # версии 2.2
  return action               # func(2)(4)=16

def func(x):
action = (lambda n, x=x: x ** n) # аргументы по умолчанию
return action                    # func(2)(4)=16
```

Несмотря на то что применение аргументов по умолчанию считается устаревшим приемом, иногда они все же требуются для ссылки на *переменные цикла* при создании функций в теле циклов. В противном случае такие переменные отражают только свое *последнее* значение в цикле, как показано ниже.

```
for I in range(N):
  actions.append(lambda I=I: F(I)) # текущее значение I
```

Объектно-ориентированное программирование

Классы служат в Python основным средством *объектно-ориентированного программирования* (ООП). В Python поддерживаются также методики *функционального программирования* с помощью таких средств, как генераторы, лямбда-выражения, отображения, замыкания, декораторы и функциональные объекты высшего порядка, которые могут служить дополнением или альтернативой ООП в некоторых контекстах.

Классы и экземпляры

Объекты классов обеспечивают следующее поведение по умолчанию.

- В операторе `class` создается объект *класса*, которому присваивается имя.
- При присваивании в операторе `class` создаются *атрибуты* класса, наследующие состояние и поведение объекта.
- Методы класса являются вложенными операторами `def` со специальным первым аргументом для получения подразумеваемого экземпляра субъекта.

Объекты-экземпляры формируются из классов следующим образом.

- Объект класса вызывается подобно функции, а в итоге создается новый *объект-экземпляр*.
- Каждый объект-экземпляр наследует атрибуты класса и получает собственное пространство имен атрибутов.
- В результате присваивания атрибутов первого аргумента в методах (например, `self.X = V`) создаются атрибуты на каждый экземпляр.

Правила наследования приведены ниже.

- Наследование происходит во время уточнения атрибута по форме *объект*`.`*атрибут*, если *объект* является классом или экземпляром.
- Классы наследуют атрибуты от всех классов, перечисленных в строке заголовка их оператора `class` (т.е. суперклассов). А перечисление больше одного класса означает *множественное наследование*.
- Экземпляры наследуют атрибуты от класса, где они формируются, плюс все суперклассы данного класса.
- При наследовании сначала осуществляется поиск экземпляра, а затем его класса и далее всех доступных суперклассов. При этом используется первая версия атрибута, найденного по имени. Поиск суперклассов, как правило, осуществляется сначала в глубину, а затем слева направо. Но в классах нового стиля поиск происходит только в ширину перед тем, как начать обход вверх по ромбовидным деревьям.

Подробнее о наследовании см. ниже, в разделе "Формальные правила наследования".

Псевдозакрытые атрибуты

По умолчанию все имена атрибутов в модулях и классах глобально доступны. Специальные соглашения допускают сокрытие некоторого ограниченного количества данных, но они предназначены главным образом для предотвращения конфликтов имен (дополнительно см. выше подраздел "Соглашения о присваивании имен").

Закрытые атрибуты модулей

Имена атрибутов в модулях, обозначенные одним знаком подчеркивания (например, *_X*) и не перечисленные в атрибуте `__all__` модуля, не копируются, когда клиент пользуется формой `from` *модуль* `import *`. Но это не строгое соблюдение закрытости, поскольку такие имена по-прежнему доступны в других формах оператора импорта.

Закрытые атрибуты классов

Имена атрибутов, обозначаемые в операторах `class` двумя начальными знаками подчеркивания (например, *`__X`*), корректируются во время компиляции таким образом, чтобы включать в себя имя объемлющего класса в качестве префикса (например, *`_Класс__X`*). Благодаря добавлению префикса имени класса такие имена локализуются в пределах объемлющего класса, а следовательно, они отличаются как в объекте-экземпляре *`self`*, так и в иерархии классов.

Это помогает избежать непреднамеренных конфликтов, которые могут возникнуть у одинаково именованных методов, а также атрибутов в единственном объекте-экземпляре в нижней части цепочки наследования. Так, если задан *`атрибут`*, то все присваивания *`self.атрибут`* в любом месте структуры объектно-ориентированного кода изменяют пространство имен единственного экземпляра. Но это не строгое соблюдение закрытости, поскольку такие атрибуты по-прежнему доступны по другому скорректированному имени.

Псевдозакрытое управление доступом может быть также реализовано с помощью классов-посредников, делающих достоверным доступ к атрибутам в методах `__getattr__()` и `__setattr__()` (см. далее раздел "Методы перегрузки операторов").

Классы нового стиля

В версии Python 3.X применяется единая модель классов, где все классы относятся к *новому стилю* независимо от того, происходят ли они от встроенного класса `object` или нет. А в версии Python 2.X применяются две модели классов: *классическая* (по умолчанию во всех выпусках версии 2.X) и *нового стиля* (начиная с версии 2.2). В последнем случае классы обозначаются как производные от встроенного типа или класса `object` (например, `class A(object)`).

Классы нового стиля, в том числе все классы в версии Python 3.X, отличаются от классических классов следующими особенностями.

- Ромбовидные шаблоны множественного наследования имеют несколько иной порядок поиска. Поиск в них осуществляется скорее в ширину, чем в глубину, перед тем, как начать обход снизу вверх (см. далее раздел "Формальные правила наследования").
- Классы теперь обозначают типы, а типы являются классами. Так, в результате вызова встроенной функции `type(I)` возвращается класс, из которого получается экземпляр, а не тип экземпляра, что, как правило, равнозначно выражению *I*`.__class__`. От класса `type` могут быть произведены подклассы для создания специальных классов. Все классы наследуют от встроенного класса `object`, предоставляющего по умолчанию небольшой набор методов.
- Методы `__getattr__()` и `__getattribute__()` больше не выполняются для атрибутов, неявно извлекаемых при выполнении встроенных операций. Они не вызываются для оператора *`__X__`* по именам методов, перегружающих встроенные операции. Поиск таких имен начинается в классах, а не в экземплярах. Для перехвата и делегирования доступа к именам таких методов их, как правило, нужно переопределить в классах-оболочках или классах-посредниках.
- У классов нового стиля имеется ряд новых средств, в том числе сегменты, свойства, дескрипторы и метод `__getattribute__()`. Большинство этих средств служат для целей построения. Подробнее об атрибуте `__slots__`, методе `__getattribute__()`, а также методах `__get__()`, `__set__()` и `__delete__()` для получения, установки и удаления дескрипторов см. ниже, в разделе "Методы перегрузки операторов", а о функции `property()` — в разделе "Встроенные функции".

Формальные правила наследования

Наследование происходит по ссылкам на имена атрибутов всякий раз, когда *`объект`* делается производным от класса. Так, по ссылке *`объект.имя`* осуществляется поиск в структуре

объектно-ориентированного кода. В классах нового и классического стиля наследование происходит по-разному, хотя обычный код зачастую выполняется одинаково в обеих моделях классов.

Классические классы: правило DFLR

В классических классах (по умолчанию в версии 2.X) поиск при наследовании по ссылкам на имена осуществляется в следующем порядке.

1. Сначала экземпляр.
2. Затем его класс.
3. Далее все суперклассы его класса с обходом сначала в глубину, а затем слева направо.

Используется первое обнаруженное вхождение. Такой порядок называется *DFLR* (Обход в глубину и слева направо).

Такой поиск ссылок может быть начат как с экземпляра, так и с класса. Результаты *присваивания* атрибутов обычно сохраняются в целевом объекте без поиска. И для метода `__getattr__()`, который выполняется, если не удалось найти имя, а также метода `__setattr__()`, который выполняется во всех остальных случаях присваивания атрибутов, особых исключений не делается.

Классы нового стиля: правило MRO

При наследовании классов нового стиля (обязательно в версии 3.X и дополнительно, но не обязательно в версии 2.X) применяется правило MRO (Порядок разрешения методов), т.е. линеаризованный обход дерева классов, причем вложенный элемент наследования становится доступным в атрибуте `__mro__` данного класса. Наследование по правилу MRO осуществляется приблизительно в следующем порядке.

1. Перечисление всех классов, наследуемых экземпляром, по правилу поиска DFLR для классических классов, причем класс включается в результат поиска столько раз, сколько он встречался при обходе.
2. Просмотр в полученном списке дубликатов классов, из которых удаляются все, кроме последнего (крайнего справа) дубликата в списке.

По правилу MRO в результирующую последовательность для данного класса включается сам класс, его суперклассы и все находящиеся выше по иерархии классы вплоть до неявно или явно указываемого корневого класса `object` на вершине иерархического дерева. Эта последовательность упорядочивается таким образом, чтобы каждый класс присутствовал в ней перед своими родителями, причем несколько родителей сохраняют тот порядок, в котором они появляются в атрибуте `__bases__` кортежа суперклассов.

Общие родители в *ромбовидных* иерархических деревьях наследования присутствуют только на позиции их *последнего* обхода по правилу MRO, и поэтому поиск начинается с нижних по иерархии классов, если в дальнейшем при наследовании атрибутов используется список, составленный по правилу MRO. Вследствие этого обход делается в большей степени в ширину, чем в глубину, но только в ромбовидных иерархических деревьях, и включение, а следовательно, и обход каждого класса производится только *один* раз независимо от количества приводящих к нему классов.

Упорядочение по правилу MRO применяется при наследовании и вызове встроенной функции `super()`, которая всегда вызывает *следующий* по правилу MRO класс (относительно точки вызова). Этот класс может вообще не быть суперклассом, но им можно воспользоваться для диспетчеризации вызовов методов, обходя каждый класс лишь один раз в иерархическом дереве классов.

Пример наследования в неромбовидных иерархических деревьях

```
class D:          attr = 3      #  D:3   E:2
class B(D):       проход        #  |     |
class E:          attr = 2      #  B     C:1
class C(E):       attr = 1      #   \   /
class A(B, C):    проход        #     A
X = A()                         #     |
print(X.attr)                   #     X

# DFLR = [X, A, B, D, C, E]
# MRO = [X, A, B, D, C, E, object]
# Всегда выводит строку "3" в обеих версиях 3.X и 2.X
```

Пример наследования в ромбовидных иерархических деревьях

```
class D:          attr = 3      #   D:3   D:3
class B(D):       проход        #   |     |
class C(D):       attr = 1      #   B     C:1
class A(B, C):    проход        #    \   /
X = A()                         #      A
print(X.attr)                   #      |
                                #      X

# DFLR = [X, A, B, D, C, D]
# MRO = [X, A, B, C, D, object] (сохраняет только
                                  # последний дубликат D)
# Выводит строку "1" в версии 3.X, строку "3" в версии 2.X
# (выводит строку "1", если D(object))
```

Алгоритм наследования классов нового стиля

В зависимости от кода класса наследование классов нового стиля включает дескрипторы, метаклассы и списки по правилу MRO, как поясняется в приведенной ниже процедуре. Источники для поиска имен в этой процедуре просматриваются по порядку номеров, слева направо или же в и том и другом порядке.

Поиск атрибутов осуществляется следующим образом.

1. Поиск начинается с *экземпляра* `I`, его класса и суперклассов в следующем порядке.

   а) Осуществляется поиск атрибута `__dict__` во всех классах по атрибуту `__mro__`, найденному в атрибуте `__class__` экземпляра `I`.

   б) Если на шаге а) найден дескриптор данных, вызывается его метод `__get__()` и процедура завершается.

   в) Иначе возвращается значение в атрибуте `__dict__` экземпляра `I`.

   г) Иначе вызывается дескриптор, не описывающий данные, или возвращается значение, найденное на шаге а).

2. Поиск продолжается в *классе* `C`, включая его суперклассы и дерево метаклассов, следующим образом:

а) Осуществляется поиск атрибута `__dict__` всех метаклассов по атрибуту `__mro__`, найденному в атрибуте `__class__` класса *C*.

б) Если на шаге а) найден дескриптор данных, вызывается его метод `__get__()` и процедура завершается.

в) Иначе вызывается дескриптор или возвращается значение в атрибуте `__dict__` класса по собственному атрибуту `__mro__` класса *C*.

г) Иначе вызывается дескриптор, не описывающий данные, или возвращается значение, найденное на шаге а).

3. В правилах 1 и 2 для неявного поиска имен методов во *встроенных* операциях (например, выражениях), по существу, используются источники, найденные на шаге а), тогда как поиск средствами функции `super()` настраивается специально.

Кроме того, метод `__getattr__()` может быть выполнен, если он определен, когда атрибут не найден. Метод `__getattribute__()` может выполняться для извлечения каждого атрибута, а подразумеваемый суперкласс `object` предоставляет некоторые данные по умолчанию на вершине иерархического дерева каждого класса или метакласса (т.е. в конце каждого списка по правилу MRO).

В *особых* случаях источники для поиска имен во встроенных операциях пропускаются, а вызов встроенной функции `super()` предотвращает обычное наследование, как поясняется в правиле 3. Для объектов, возвращаемых функцией `super()`, атрибуты разрешаются в результате специального просмотра с учетом контекста лишь ограниченной части списка, составленного для класса по правилу MRO. При этом выбирается первый же найденный дескриптор или значение вместо полного наследования, которое выполняется только при неудачном исходе данного просмотра. Подобнее о функции `super()` см. далее, в разделе "Встроенные функции".

Для присваивания имен атрибутов выполняется следующее подмножество процедур поиска.

- Применительно к *экземпляру* подобные операции присваивания выполняются в соответствии с шагами a)–c) упомянутого выше правила 1, хотя на шаге b) вызывается метод `__set__()`, а не `__get__()`, тогда как на шаге c) процедура завершается и полученный результат сохраняется в экземпляре, а не извлекается.
- Применительно к *классу* подобные операции присваивания выполняются по той же самой процедуре обхода дерева метаклассов данного класса, т.е. приблизительно по правилу 2, но на шаге c) процедура завершается и полученный результат сохраняется в классе.

В методе `__setattr__()` по-прежнему перехватываются все операции присваивания атрибутов, хотя пользоваться атрибутом `__dict__` экземпляра для присваивания имен с помощью этого метода оказывается не так удобно, поскольку в некоторых расширениях нового стиля, в том числе сегментах, свойствах и дескрипторах, атрибуты реализуются на уровне класса. Это своего рода "виртуальный" механизм доступа к данным экземпляра. У некоторых экземпляров атрибут `__dict__` может вообще отсутствовать, когда для оптимизации используются сегменты.

Предшествование и контекст классов нового стиля

В процедурах наследования классов нового стиля основополагающая операция разрешения имен выполняется по правилам предшествования. Ниже приведен порядок выполнения этой операции, где соответствующие шаги упомянутого выше алгоритма наследования указываются в круглых скобках.

Для экземпляров порядок наследования следующий.

1. Дескрипторы данных из иерархического дерева классов (шаг б) правила 1).
2. Значения объекта-экземпляра (шаг в) правила 1).
3. Дескрипторы, не описывающие данные, из иерархического дерева классов (шаг г) правила 1).
4. Значения из иерархического дерева классов (шаг г) правила 1).

Для классов порядок наследования следующий.

1. Дескрипторы данных из иерархического дерева метаклассов (шаг б) правила 2).
2. Значения объекта-экземпляра (шаг в) правила 2).
3. Значения из иерархического дерева классов (шаг в) правила 2).
4. Дескрипторы, не описывающие данные, из иерархического дерева классов (шаг г) правила 2).
5. Значения из иерархического дерева метаклассов (шаг г) правила 2).

В Python выполняется хотя бы один поиск (по правилу 1) или два поиска (по правилу 2) имени в иерархическом дереве, несмотря на наличие четырех или пяти имен для поиска. См. также описание в предыдущем подразделе процедуры поиска в специальных случаях для объектов, возвращаемых встроенной функцией `super()` для классов нового стиля.

Дополнительные сведения о метаклассах см. выше, в подразделе "Метаклассы", а о дескрипторах — ниже, в разделе "Методы для операций с дескрипторами". Подробнее о применении методов `__setattr__()`, `__getattr__()` и `__getattribute__()` и файлах `object.c` и `typeobject.c` исходного кода из дистрибутива Python, в которых реализуются экземпляры и классы соответственно, см. ниже, в разделе "Методы перегрузки операторов".

Методы перегрузки операторов

Встроенные операции могут перехватываться и реализовываться в классах с помощью специальных методов, имена которых начинаются и оканчиваются двумя знаками подчеркивания. Такие имена не резервируются и могут наследоваться из суперклассов обычным образом. Интерпретатор Python обнаруживает и автоматически вызывает хотя бы один такой метод для каждой встроенной операции.

Методы перегрузки операторов вызываются интерпретатором Python, когда экземпляры появляются в выражениях и других

контекстах. Так, если в классе определен метод `__getitem__()`, а *X* является экземпляром этого класса, то выражение `X[i]` равнозначно следующему вызову данного метода: `X.__getitem__(i)`. Хотя в настоящее время непосредственная форма вызова методов, как правило, не дает никаких преимуществ и может даже повлечь за собой снижение производительности.

Имена методов перегрузки операторов в какой-то степени произвольны. В частности, метод `__add__()` класса может и не выполнять ни сложение, ни сцепление, хотя он должен, как правило, служить одной и той же цели. Более того, в классах могут попеременно использоваться методы перегрузки операторов для обработки чисел и коллекций, а также изменяемых и неизменяемых операций. У большинства методов перегрузки операторов отсутствуют стандартные имена, за исключением методов из класса `object` для классов нового стиля. При выполнении операции возникает исключение, если соответствующий метод не определен (например, при выполнении операции **+**, для которой отсутствует метод `__add__()`).

В последующих разделах рассматриваются имеющиеся методы перегрузки операторов. При описании этих методов круглые скобки, завершающие их имена, опускаются ради краткости там, где это уместно. Приведенное далее описание касается в основном версии Python 3.X, хотя в нем рассматриваются общие характеристики этих методов для большинства версий Python. А их отличительные особенности для разных версий приведены далее, в разделе "Методы перегрузки операторов в версии Python 2.X".

Методы для всех видов операций

`__new__(cls [, arg]*)`

Вызывается для создания и возврата нового экземпляра класса *cls*. Получает аргументы *arg*, передаваемые конструктору класса *cls*. Если этот метод возвращает экземпляр класса *cls*, то вызывается метод `__init__` этого экземпляра с новым экземпляром *self* и теми же самыми аргументами для конструктора, в противном случае метод `__init__` не выполняется. Как правило, служит для

вызова метода __new__ из суперкласса по явно указываемому имени суперкласса или с помощью функции super() (см. далее раздел "Встроенные функции"), а также для управления и возврата результирующего экземпляра. Этот метод автоматически объявляется статическим.

Не применяется в обычных классах и предназначается для специальной настройки процесса создания как экземпляров подклассов изменяемых типов, так и классов в специальных *метаклассах*. Дополнительно о применении данного метода вместе с аргументами для создания класса см. описание функции type() далее, в разделе "Встроенные функции".

__init__(*self*[, *arg*]*)

Вызывается в операции *class*(*args...*). Это *метод-конструктор*, инициализирующий экземпляр *self*. Когда он вызывается по имени класса, экземпляр *self* предоставляется автоматически. Аргументы передаются классу указанного имени и могут принимать любую форму определения функции (см. выше разделы "Оператор выражения" и "Оператор def", включая табл. 14).

Несмотря на то что метод __init__ формально вызывается после метода __new__, он более предпочтителен для формирования новых объектов во всех классах на уровне приложения. Он не должен возвращать значение, а если требуется, то должен вручную вызывать метод __init__ из суперкласса, передавая ему экземпляр self посредством явного указания имени суперкласса или вызова функции super() (см. далее раздел "Встроенные функции"). Интерпретатор Python автоматически вызывает метод __init__ лишь один раз.

__del__(*self*)

Вызывается для сборки экземпляра в "мусор". Это *метод-деструктор*, очищающий оперативную память при освобождении из нее экземпляра *self*. Встроенные объекты автоматически освобождаются из оперативной памяти,

когда имеется их контейнер, если только на них не делаются ссылки из других мест в коде. Исключения, возникающие при выполнении этого метода, игнорируются и просто выводятся в виде сообщений в стандартный поток `sys.stderr`. *Совет:* операторы `try/finally` позволяют завершать действия в кодовом блоке более предсказуемым способом, а оператор `with` предоставляет аналогичные возможности для поддерживаемых типов объектов.

`__repr__(self)`

Вызывается в операции `repr(self)`, при интерактивном эхоотображении, вложенных вхождениях, а также в выражении `'self'` (только в версии Python 2.X). Вызывается также в функции `str(self)` и `print(self)`, если отсутствует метод `__str__`. Как правило, этот метод возвращает строковое представление объекта *`self`* на низком уровне кода.

`__str__(self)`

Вызывается в операции `str(self)` и `print(self)` или использует метод `__repr__` как резервный вариант, если он определен. Как правило, этот метод возвращает строковое представление объекта *`self`* на высоком уровне, удобном для восприятия пользователем.

`__format__(self, спецификация_формата)`

Вызывается встроенной функцией `format()` или расширяющим ее методом `str.format()` для символьных строк типа `str`, чтобы получить отформатированное строковое представление объекта *`self`* для каждой символьной строки *`спецификация_формата`*, синтаксис которой для встроенных типов данных такой же, как и для аналогично называемой составляющей в вызове метода `str.format()`. Дополнительно см. выше подразделы "Синтаксис метода форматирования" и "Метод форматирования символьных строк", а далее — раздел "Встроенные функции". Этот метод внедрен в версиях Python 2.6 и 3.0.

__bytes__(*self*)

Вызывается функцией `bytes()` для возврата строкового представления типа `bytes` объекта `self` (только в версии Python 3.X).

__hash__(*self*)

Вызывается в операциях `dictionary[`*`self`*`]`, `hash(`*`self`*`)` и других операциях над хешированными коллекциями, включая и те, что выполняются над объектами типа `set`. Этот метод возвращает однозначный и неизменяемый целочисленный хеш-ключ и едва заметно взаимодействует с методом `__eq__`. По умолчанию в обоих случаях гарантируется, что объекты оказываются равными только в том случае, если они сравниваются сами с собой (подробнее об этом см. в документации на Python).

__bool__(*self*)

Вызывается для проверки значения на истинность и во встроенной функции `bool()`, а возвращает логическое значение `False` или `True`. Если метод `__bool__` не определен, то вызывается метод `__len__()`, при условии, что он определен и обозначает истинное значение как имеющее ненулевую длину. Если же в классе не определен ни метод `__len__`, ни `__bool__`, то все экземпляры этого класса считаются истинными. Этот метод внедрен в версии Python 3.X, а в версии Python 2.X он называется `__nonzero__`, но последний действует таким же образом, как и метод `__bool__`.

__call__(*self* [, *arg*]*)

Вызывается в операции `self(`*`args`*`...)`, когда экземпляр вызывается как функция. Агрумент *`arg`* может принимать любую форму, принятую в определении функции. Например, оба следующих определения:

```
def __call__(self, a, b, c, d=5):
def __call__(self, *pargs, **kargs):
```

соответствуют двум приведенным ниже вызовам.

```
self(1, 2, 3, 4)
self(1, *(2,), c=3, **dict(d=4))
```

Подробнее о формах определения аргументов *args* см. ранее в разделе "Оператор def", включая табл. 14.

__getattr__(self, имя)

Вызывается по ссылке *self.имя*, где *имя* — символьная строка, обозначающая доступ к неопределенному атрибуту (этот метод не вызывается, если *имя* существует в объекте *self* или наследуется ним). Возвращает объект или генерирует исключение типа AttributeError.

Доступен в обоих разновидностях классов классического и нового стиля. В классах нового стиля версий Python 2.X и 3.X этот метод не выполняется для атрибутов *__X__*, неявно извлекаемых во *встроенных операциях* (например, выражениях). Подобные имена следует переопределить в классах-оболочках, классах-посредниках или суперклассах. См. также приведенное далее описание метода __dir__.

__setattr__(self, имя, значение)

Вызывается в операции *self.имя=значение* (для всех видов присваивания атрибутов). *Совет:* во избежание рекурсивных циклов атрибуты рекомендуется присваивать с помощью ключа __dict__ или суперкласса (например, object). Оператор *self.attr=x* в методе __setattr__ снова вызывает этот же метод, чего не происходит в операции self.__dict__['attr']=x.

Для того чтобы избежать рекурсии, достаточно вызвать этот метод явно из суперкласса object класса нового стиля следующим образом: object.__setattr__(*self, attr, значение*). Такой способ может оказаться более предпочтительным или даже обязательным в иерархических деревьях тех классов, где на уровне классов реализуются атрибуты "виртуальных" экземпляров, в том числе *сегменты*, *свойства* или *дескрипторы* (например, сегменты могут исключать атрибут __dict__ экземпляра).

__delattr__(*self*, *имя*)

Вызывается в операции del *self.имя* (при удалении всех атрибутов). *Совет:* и в этом случае следует избегать рекурсивных циклов, направляя операции удаления атрибутов через атрибут __dict__ или суперкласс, как и в методе __setattr__.

__getattribute__(*self*, *имя*)

Вызывается безусловно для реализации доступа к атрибутам из экземпляров класса. Если в классе определяется также метод __getattr__, он вообще не вызывается, при условии, что этого не происходит явным образом. Этот метод должен возвращать (вычисляемое) значение атрибута или генерировать исключение типа AttributeError. Во избежание в данном методе бесконечной рекурсии в его реализации должен всегда вызываться одноименный метод из суперкласса для доступа к любым атрибутам по мере необходимости, как в следующем примере: object.__getattribute__(*self*, *имя*).

Этот метод доступен как в версии Python 3.X, так и в версии Python2.X, но только для классов *нового стиля*. И в обоих случаях он не выполняется для атрибутов __*X*__, неявно извлекаемых во *встроенных операциях* (например, в выражениях). Подобные имена следует переопределить в классах-оболочках, классах-посредниках или суперклассах. См. также приведенное далее описание метода __dir__.

__lt__(*self*, *другое*)
__le__(*self*, *другое*)
__eq__(*self*, *другое*)
__ne__(*self*, *другое*)
__gt__(*self*, *другое*)
__ge__(*self*, *другое*)

В указанном выше порядке эти методы используются в следующих операциях: *self* < *другое*, self <= *другое*, self == *другое*, self != *другое*, self > *другое*, а также self >= *другое*. Они были внедрены в версии 2.1,

называются *методами расширенного сравнения* и вызываются во всех выражениях сравнения в версии Python 3.X. Например, в операции сравнения *X* < *Y* вызывается метод *X*.__lt__(*Y*), если он определен. А в версии Python 2.X (и только в ней) эти методы вызываются как более предпочтительные, чем метод __cmp__. Кроме того, в версии 2.X метод __ne__ вызывается в операции сравнения self <> *другое*.

Эти методы могут *возвращать* любое значение, но если оператор сравнения используется в логическом контексте, то возвращаемое значение интерпретируется как логическое, получаемое в результате выполнения данного оператора. Эти методы могут также возвращать (но не формировать) специальный объект типа NotImplemented, если перегружаемая ими операция не поддерживается для указанных операндов (как будто метод вообще не переопределен, что вынуждает обращаться к более общему методу __cmp__ в версии Python 2.X, если таковой определен).

У операторов сравнения отсутствуют взаимосвязи. Например, операция *X* == *Y* дает истинный результат, но не подразумевает, что операция *X* != *Y* дает ложный результат. Поэтому метод __ne__ должен быть определен наряду с методом __eq__, если поведение операторов предполагается симметричным. У этих методов отсутствуют также правосторонние варианты (с переставленными аргументами), которые можно использовать, если левый аргумент не поддерживает операцию, а правый аргумент поддерживает ее. Так, методы __lt__ и __gt__, __le__ и __ge__ являются зеркальным отражением друг друга, а методы __eq__ и __ne__ — их собственным зеркальным отражением. Для сортировки рекомендуется использовать метод __lt__ в версии Python 3.X, а о роли метода __eq__ в хешировании см. в документации на Python.

__slots__

Этому атрибуту класса может быть присвоена символьная строка, последовательность и другой итерируемый объект, состоящий из символьных строк, задающих имена атрибутов экземпляров класса. Если атрибут `__slots__` определен в классе *нового стиля*, включая все классы в версии Python 3.X, он формирует дескриптор управления на уровне класса (см. далее раздел "Методы для операций с дескрипторами"), резервирует место для объявляемых атрибутов в экземплярах и предотвращает автоматическое создание атрибута `__dict__` для каждого экземпляра, если только символьная строка `'__dict__'` не входит в сам атрибут `__slots__`. В последнем случае экземпляры содержат также атрибут `__dict__`, тогда как атрибуты, не названные в атрибуте `__slots__`, могут быть введены динамически.

Сегменты могут подавлять атрибут `__dict__` для каждого экземпляра, и поэтому они позволяют оптимизировать использование свободного пространства. Тем не менее пользоваться сегментами обычно не рекомендуется, кроме патологических случаев по двум причинам: они могут нарушить некоторые виды кода, и на их применение накладываются сложные ограничения (подробнее об этом см. в документации на Python).

Для поддержки классов с атрибутом `__slots__` языковые средства, которые обычно перечисляют атрибуты или получают доступ к ним по имени символьной строки, должны пользоваться нейтральными к хранению средствами вроде функций `getattr()`, `setattr()` и `dir()`, пригодных для сохранения атрибутов `__dict__` и `__slots__`.

__instancecheck__(*self, экземпляр*)

Возвращает логическое значение `True` для функции `isinstance()`, если *`экземпляр`* считается прямым или косвенным экземпляром класса. Внедрен в версиях 2.6 и Python 3.X. Подробнее о применении этого метода см. в документации на Python.

__subclasscheck__(*self*, *подкласс*)

Возвращает логическое значение `True` для функции `issubclass()`, если *подкласс* должен считаться прямым или косвенным подклассом, производным от данного класса. Доступен начиная с версий 2.6 и Python 3.X. Подробнее о применении этого метода см. в документации на Python.

__dir__(*self*)

Вызывается в операции `dir(`*self*`)` (см. далее раздел "Встроенные функции") и возвращает последовательность имен атрибутов. Позволяет некоторым классам делать свои атрибуты доступными для самоанализа с помощью функции `dir()`, когда эти атрибуты вычисляются динамически такими средствами, как метод `__getattr__`, но известными самому классу. Возможно, этот метод и нельзя применять в динамическом режиме, но некоторые *общие заместители* все же позволяют делегировать его вызов замещаемым объектам для поддержки языковых средств атрибутов. Доступен с версии Python 3.X, а для его поддержки в версиях Python 2.6 и 2.7 сделаны соответствующие "заплаты".

Методы для операций над коллекциями (последовательностями и отображениями)

__len__(*self*)

Вызывается в операции `len(`*self*`)`, а возможно, и для проверок на истинность значения. Возвращает размер коллекции. Для логических проверок интерпретатор Python ищет сначала метод `__bool__`, затем метод `__len__` и далее определяет истинность объекта. Нулевая длина означает ложность объекта. (В версии Python 2.X метод `__bool__` называется `__nonzero__`.)

__contains__(*self*, *элемент*)

Вызывается в операции *элемент* `in` *self* для специальной проверки на членство (в противном случае для

членства используется метод __iter__, если он определен, а иначе — метод __getitem__). Этот метод возвращает логическое значение True или False.

__iter__(*self*)

Вызывается в операции iter(*self*). Внедрен в версии 2.2 как составная часть *протокола итерации*. Возвращает объект (возможно, *self*) с методом __next__. В итоге метод __next__() этого объекта неоднократно вызывается во всех контекстах итерации (например, в циклах for) и должен возвратить следующий результат или сгенерировать исключение типа StopIteration, чтобы прекратить дальнейшее продвижение результатов.

Если метод __iter__ не определен ни в одном из классов, итерация осуществляется с помощью резервного метода __getitem__. В методе __iter__ из отдельного класса может быть также использован оператор yield для возврата генератора с автоматически создаваемым методом __next__. В версии Python 2.X метод __next__ называется next. Дополнительно см. выше раздел "Оператор for" и подраздел "Протокол итерации".

__next__(*self*)

Вызывается встроенной функцией next(*self*) и во всех контекстах итерации для продвижения результатов. Этот метод является составной частью *протокола итерации*, как пояснялось выше при описании метода __iter__. Внедрен в версии Python 3.X, а в версии Python 2.X он называется next, но действует таким же образом.

__getitem__(*self, ключ*)

Вызывается в операциях *self*[*ключ*], *self*[*i*:*j*:*k*], *x* in *self*, а возможно, и во всех контекстах итерации. Этот метод реализует все операции, связанные с индексированием, включая и операции над последовательностями и отображениями. В контекстах итерации (например, в операторах in и for) индекс неоднократно увеличивается от нуля и вплоть до возникновения исключения типа IndexError, если только не определен более предпочтительный метод

__iter__. Совместно методы __getitem__ и __len__ образуют протокол последовательности.

В версии Python 3.X этот и два следующих метода называются иначе операциями *нарезки*, где *ключ* является объектом нарезки. Такие объекты могут передаваться и в другие выражения нарезки, а также иметь атрибуты start, stop и step, каждый из которых может принимать значение None (Отсутствует). Дополнительно см. описание функции slice() далее, в разделе "Встроенные функции".

__setitem__(*self*, *ключ*, *значение*)

Вызывается в операциях *self*[*ключ*] и *self*[*i*:*j*:*k*]=*значение*. Этот метод служит для присваивания значений по ключу или индексу коллекции или же по нарезке последовательности.

__delitem__(*self*, *ключ*)

Вызывается в операциях del *self*[*ключ*] и del *self*[*i*:*j*:*k*]. Этот метод служит для удаления элементов из коллекции по ключу или индексу или же из последовательности по нарезке.

__reversed__(*self*)

Если этот метод определен, то он вызывается встроенной функцией reversed() для реализации специальной итерации в обратном порядке. Возвращает новый итерируемый объект для перебора в обратном порядке всех объектов, находящихся в контейнере. Если метод __reversed__ не определен ни в одном из классов, то предполагается вызов функции reversed() и применение протокола последовательности (т.е. методов __len__ и __getitem__).

Методы для числовых операций в двоичной форме

Методы для числовых операций и сравнения, не поддерживающие соответствующую операцию над предоставляемыми

аргументами, должны возвращать (но не формировать) специальный встроенный объект типа NotImplemented, как будто метод вообще не определен. Операции, не поддерживаемые ни для одного из типов операндов, должны оставаться неопределенными.

Примеры назначения операторов во встроенных типах приведены в табл. 1, хотя назначение операторов определяется классами, в которых осуществляется их перегрузка. Например, метод __add__ вызывается в операции + как для арифметического сложения, так и для сцепления последовательности, но в новых классах он может иметь произвольную семантику.

Основные методы для операций над двоичными числами

__add__(*self*, *другое*)

Вызывается в операции *self* + *другое*.

__sub__(*self*, *другое*)

Вызывается в операции *self* - *другое*.

__mul__(*self*, *другое*)

Вызывается в операции *self* * *другое*.

__truediv__(*self*, *другое*)

Вызывается в операции *self* / *другое* в версии Python 3.X. А в версии Python 2.X вместо этого вызывается метод __div__, если только не разрешено настоящее деление (см. выше раздел "Примечания к применению операторов").

__floordiv__(*self*, *другое*)

Вызывается в операции *self* // *другое*.

__mod__(*self*, *другое*)

Вызывается в операции *self* % *другое*.

__divmod__(*self*, *другое*)

Вызывается в операции divmod(*self*, *другое*).

__pow__(*self*, *другое* [, *по модулю*])

Вызывается в операциях pow(*self*, *другое* [, *по модулю*]) и *self* ** *другое*.

__lshift__(*self*, *другое*)

Вызывается в операции *self* **<<** *другое*.

__rshift__(*self*, *другое*)

Вызывается в операции *self* **>>** *другое*.

__and__(*self*, *другое*)

Вызывается в операции *self* **&** *другое*.

__xor__(*self*, *другое*)

Вызывается в операции *self* **^** *другое*.

__or__(*self*, *другое*)

Вызывается в операции *self* **|** *другое*.

Методы для правосторонних операций над двоичными числами

__radd__(*self*, *другое*)
__rsub__(*self*, *другое*)
__rmul__(*self*, *другое*)
__rtruediv__(*self*, *другое*)
__rfloordiv__(*self*, *другое*)
__rmod__(*self*, *другое*)
__rdivmod__(*self*, *другое*)
__rpow__(*self*, *другое*)
__rlshift__(*self*, *другое*)
__rrshift__(*self*, *другое*)
__rand__(*self*, *другое*)
__rxor__(*self*, *другое*)
__ror__(*self*, *другое*)

Эти методы являются правосторонними аналогами методов, перегружающих двоичные операторы из предыдущего подраздела. У методов перегрузки двоичных операторов имеются правосторонние варианты, имена которых начинаются с префикса **r** (например, `__add__` и `__radd__`). У правосторонних вариантов этих методов имеются те же самые списки аргументов, но объект *self* оказывается в правой части оператора. Например, в операции *self* **+** *другое* вызывается метод

self.__add__(*другое*), тогда как в операции *другое* + *self* — метод *self*.__radd__(*другое*).

Методы для правосторонних операций над двоичными числами вызываются только в том случае, если экземпляр указывается справа, а левый операнд не является экземпляром класса, реализующего данную операцию, как показано ниже.

- В операции *экземпляр* + *неэкземпляр* выполняется метод __add__.
- В операции *экземпляр* + *экземпляр* выполняется __add__.
- В операции *экземпляр* + *экземпляр* выполняется метод __radd__.

Если в операции указываются экземпляры двух разных классов, где эта операция перегружается, то предпочтение отдается методу из класса левого экземпляра. Так, метод __radd__ нередко изменяет порядок выполнения операции сложения, вызывая метод __add__.

Комбинированные методы для операций над двоичными числами

__iadd__(*self*, *другое*)
__isub__(*self*, *другое*)
__imul__(*self*, *другое*)
__itruediv__(*self*, *другое*)
__ifloordiv__(*self*, *другое*)
__imod__(*self*, *другое*)
__ipow__(*self*, *другое*[, *по модулю*])
__ilshift__(*self*, *другое*)
__irshift__(*self*, *другое*)
__iand__(*self*, *другое*)
__ixor__(*self*, *другое*)
__ior__(*self*, *другое*)

Это методы комбинированного (непосредственного) присваивания. В указанном выше порядке они вызываются при выполнении операций комбинированного присваивания в следующих форматах: +=, -=, *=, /=, //=, %=, **=, <<=, >>=, &=, ^= и |=.

Эти методы должны пытаться выполнить операцию непосредственно, видоизменяя объект *self* и возвращая результат, которым может быть и сам объект *self*. Если же метод перегрузки соответствующего оператора не определен, то в качестве резервного варианта вызывается обычный метод. Так, для вычисления выражения *X* += *Y*, где *X* — экземпляр класса, в котором определен метод __iadd__, делается вызов *X*.__iadd__(*Y*), в противном случае рассматривается возможность вызвать методы __add__ и __radd__.

Методы для других операций над числами

__neg__(*self*)

Вызывается в операции -*self*.

__pos__(*self*)

Вызывается в операции +*self*.

__abs__(*self*)

Вызывается в операции abs(*self*).

__invert__(*self*)

Вызывается в операции ~*self*.

__complex__(*self*)

Вызывается в операции complex(*self*).

__int__(*self*)

Вызывается в операции int(*self*).

__float__(*self*)

Вызывается в операции float(*self*).

__round__(*self*[, *n*])

Вызывается в операции round(*self* [, *n*]). Внедрен в версии Python 3.X.

__index__(*self*)

Вызывается для реализации операции *оператор*.index(), а также в других контекстах, где интерпретатор

Python требует наличия целочисленного объекта, включая вхождения экземпляров в виде индексов, границ нарезки и аргументов встроенных функций `bin()`, `hex()` и `oct()`. Этот метод должен возвращать целочисленное значение.

Действует одинаково в версиях Python 2.X и 3.X, но не вызывается для функций `hex()` и `oct()` в версии 2.X, поскольку в этой версии им требуются методы `__hex__` и `__oct__`. В версии Python 3.X метод `__index__` обобщает и заменяет методы `__oct__` и `__hex__` из версии Python 2.X и автоматически форматирует возвращаемое целочисленное значение.

Методы для операций с дескрипторами

Перечисленные ниже методы применяются только в том случае, если экземпляр класса, в котором они определены, т.е. класса *дескриптора*, присваивается атрибуту другого класса, называемого *классом-владельцем*. Эти методы автоматически вызываются из дескриптора для доступа к атрибуту в классе-владельце и его экземплярах.

`__get__(self, экземпляр, владелец)`

Вызывается для получения атрибута из класса-владельца или экземпляра данного класса. В качестве *`владельца`* всегда указывается класс-владелец; в качестве *`экземпляра`* — экземпляр, из которого получается нужный атрибут, или же значение `None`, если атрибут получается непосредственно из класса-владельца; а в качестве *`self`* — экземпляр класса дескриптора. Возвращает значение атрибута или генерирует исключение типа `AttributeError`. Оба аргумента *`self`* и *`экземпляр`* могут содержать сведения о состоянии.

`__set__(self, экземпляр, значение)`

Вызывается для установки нового *`значения`* атрибута в *`экземпляре`* класса-владельца.

__delete__(*self*, *экземпляр*)

Вызывается для удаления атрибута в *экземпляре* класса-владельца.

Дескрипторы и их методы доступны для классов *нового стиля*, включая все классы в версии 3.X. Они полностью работоспособны и в версии 2.X, но только в том случае, если классы дескрипторов и классы-владельцы относятся к новому стилю. Дескриптор с методом `__set__` называется *дескриптором данных*, и поэтому именно ему отдается предпочтение над другими именами при наследовании (см. ранее раздел "Формальные правила наследования").

НА ЗАМЕТКУ

В данном контексте *дескрипторы классов* отличаются от *дескрипторов файлов* (о первом см. ранее раздел "Файлы", а в последнем — далее раздел "Средства для дескрипторов файлов".

Методы для операций с диспетчерами контекста

Перечисленные ниже методы реализуют протокол диспетчера контекста, используемый в операторе `with` (о механизме, в котором эти методы применяются, см. ранее раздел "Оператор `with`").

__enter__(*self*)

Входит в динамический контекст, связанный с данным объектом. В операторе `with` значение, возвращаемое данным методом, присваивается адресату, указанному в выражении `as` данного оператора, если таковое указано.

__exit__(*self*, *тип*, *значение*, *объект обратной трассировки стека*)

Выходит из динамического контекста, связанного с данным объектом. Параметры, указываемые после *self*, описывают исключение, возникающее в результате выхода из динамического контекста. Если же выход из динамического

контекста осуществляется без исключения, то все три аргумента принимают значение None. В противном случае аргументы остаются теми же, как и результаты вызова функции sys.exc_info() (см. далее раздел "Модуль sys").

Возвращает истинное значение, чтобы предотвратить распространение исключения в вызывающий код.

Методы перегрузки операторов в версии Python 2.X

В предыдущем разделе отмечались семантические отличия в методах перегрузки операторов, доступных в *обеих* версиях, Python 3.X и 2.X. А в этом разделе отмечаются отличия в их содержании, наблюдаемые в обеих версиях Python.

Некоторые методы, описанные в предыдущем разделе, действуют и в версии 2.X, но только по отношению к классам *нового стиля*, которые являются необязательным расширением версии 2.X. К их числу относятся методы __getattribute__, __slots__, а также методы для операций с дескрипторами. Другие методы могут вести себя иначе в версии 2.X по отношению к классам нового стиля (например, метод __getattr__ для встроенных операций), причем некоторые из этих методов доступны только в последующих выпусках версии 2.X (например, методы __dir__, __instancecheck__, __subclasscheck__). Ниже описываются методы, характерные только для версии 3.X или 2.X.

Методы, характерные только для версии Python 3.X

Перечисленные ниже методы поддерживаются в версии Python 3.X, но отсутствуют в версии Python 2.X.

- **__round__**
- **__bytes__**
- **__bool__** (в версии Python 2.X вместо этого метода следует использовать метод **__nonzero__** или **__len__**)
- **__next__** (в версии Python 2.X вместо этого метода следует использовать метод **next**)

- **__truediv__** (доступен в версии Python 2.X только в том случае, если активизировано настоящее деление; см. ранее раздел "Примечания к применению операторов")
- **__index__** для применения в операциях `oct()` и `hex()` (в версии Python 2.X вместо этого метода следует использовать методы **__oct__** и **__hex__**).

Методы, характерные только для версии Python 2.X

Перечисленные ниже методы в поддерживаются версии Python 2.X, но отсутствуют в версии Python 3.X.

__cmp__(*self*, *другое*) (и метод **__rcmp__**)

Вызывается в операциях *self* > *другое*, *другое* == *self*, cmp(*self*, *другое*) и прочих, а также во всех операциях сравнения, для которых конкретный метод (например, `__lt__`) не определен или не наследуется. Он возвращает значение **-1**, **0** или **1**, если объект *self* меньше, равен или больше, чем *другое*. Если же методы для операций расширенного сравнения или метод `__cmp__` не определены ни в одном из классов, то экземпляры классов сравниваются по их идентичности (адресу в оперативной памяти). Правосторонний вариант `__rcmp__` этого метода не поддерживается с версии 2.1.

В версии Python 3.X следует использовать более конкретные, описанные ранее методы для операций сравнения: `__lt__`, `__ge__`, `__eq__` и пр. В частности, для сортировки в Python 3.X следует использовать метод `__lt__`.

__nonzero__(*self*)

Вызывается по истинному значению, а иначе используется метод `__len__`, если таковой определен. В версии Python 3.X этот метод переименован в **__bool__**.

__getslice__(*self*, *low*, *high*)

Вызывается в операции *self*[*low*:*high*] для нарезки последовательности. Если же метод `__getslice__` не определен ни в одном из классов и недоступен для расширен-

ной трехэлементной нарезки, то *объект нарезки* передается методу `__getitem__`.

В версии Python 2.X этот и два следующих метода не рекомендуются к употреблению, но по-прежнему поддерживаются. Если они определены, то вызываются для выражений нарезки как более предпочтительные, чем их поэлементные аналоги. В версии Python 3.X эти три метода полностью исключены, а в операциях нарезки вместо них всегда вызываются методы `__getitem__`, `__setitem__` или `__delitem__` с объектом нарезки в качестве аргумента. Дополнительно см. далее раздел "Встроенные функции".

`__setslice__(self, low, high, значение)`

Вызывается в операции `self[low:high]=значение` для присваивания заданного значения нарезке последовательности. См. также приведенное выше примечание к методу `__getitem__` относительно нерекомендованности его к употреблению в версии Python 2.X.

`__delslice__(self, low, high)`

Вызывается в операции `del self[low:high]` для удаления нарезки последовательности. См. также приведенное выше примечание к методу `__getitem__` относительно нерекомендованности его к употреблению в версии Python 2.X.

`__div__(self, другое)` (а также методы **`__rdiv__`**, **`__idiv__`**)

Вызываются в операции `self / другое`, если только не активизировано настоящее деление (в этом случае выполняется метод `__truediv__`). В версии Python 3.X эти методы относятся к категории методов `__truediv__`, `__rtruediv__` и `__itruediv__`, поскольку операция `/` всегда означает настоящее деление. Дополнительно см. ранее раздел "Примечания к применению операторов". *Совет:* для поддержки обеих моделей деления в одном методе следует выполнить присваивание `__truediv__ = __div__`.

__long__(*self*)

Вызывается в операции long(*self*). В версии Python 3.X тип int полностью включает в себя тип long, и поэтому этот метод исключен из данной версии.

__oct__(*self*)

Вызывается в операции oct(*self*). Этот метод возвращает строковое представление объекта в восьмеричной форме. А в версии Python 3.X этот метод возвращает целочисленное значение для операции __index__().

__coerce__(*self*, *другое*)

Вызывается в арифметическом выражении с разнотипными членами и функцией coerce(). Этот метод возвращает кортеж (*self*, *другое*), преобразуемый к общему типу. Если метод __coerce__ определен, он, как правило, вызывается перед любыми попытками вызвать конкретный метод перегрузки операторов (например, перед вызовом метода __add__). Он должен возвращать кортеж, состоящий из операндов, преобразуемых к общему типу (или значение None, если такое преобразование невозможно). Подробнее о правилах приведения типов см. в руководстве по языку Python (Python Language Reference).

__unicode__(*self*)

В версии 2.X вызывается в операции unicode(*self*), чтобы возвратить символьную строку в уникоде для объекта *self* (см. далее раздел "Встроенные функции в версии Python 2.X"). Этот метод является аналогом метода __str__ в уникоде.

__metaclass__

Атрибут класса, присваиваемый его метаклассу. В версии Python 3.X вместо него следует использовать синтаксис присваивания именованных аргументов *метакласс*=M в строке заголовка класса (см. ранее подраздел "Метаклассы").

Встроенные функции

Все встроенные имена (функций, исключений и прочего) существуют в подразумеваемой внешней встроенной области действия, которая соответствует модулю `builtins`, называемому `__builtin__` в версии Python 2.X. При поиске имен эта область действия ищется в последнюю очередь, и поэтому встроенные функции всегда доступны в программах без дополнительного импорта. Тем не менее их имена не относятся к зарезервированным словам и могут быть скрыты (затенены) присваиванием одного и того же имени как в глобальных, так и в локальных областях действия. Для получения дополнительных сведений о любой рассматриваемой здесь функции достаточно сделать вызов `help(`*функция*`)`.

В этом разделе основное внимание уделяется встроенным функциям в версии Python 3.X, но в то же время приводятся их характеристики, общие для всех версий Python. А в конце этого раздела приведены особенности встроенных функций, характерные для отдельных версий Python.

`abs(`*N*`)`

Возвращает абсолютное значение числа *N*.

`all(`*итерируемый_объект*`)`

Возвращает логическое значение `True` только в том случае, если истинны все элементы *итерируемого_объекта*.

`any(`*итерируемый_объект*`)`

Возвращает логическое значение `True` только в том случае, если истинен любой элемент *итерируемого_объекта*. *Совет:* в результате обеих операций, `filter(bool, I)` и `[x for x in I if x]`, все истинные значения накапливаются в итерируемом объекте *I*.

`ascii(`*объект*`)`

Аналогично функции `repr()`, возвращает символьную строку, содержащую выводимое на печать представление объекта, но экранирует управляющими последовательностями **`\x`**, **`\u`** или **`\U`** символы, не представленные в коде ASCII и получаемые в результирующей символьной строке

из функции repr(). Получаемый результат аналогичен возвращаемому функцией repr() в версии Python 2.X.

bin(*N*)

Преобразует целое число в символьную строку двоичных цифр (по основанию **2**). В итоге получается действительное в Python выражение. Если аргумент *N* не является объектом типа int в Python, то должен быть определен метод __index__(), возвращающий целое значение. *Совет:* для преобразования из двоичного числа см. функцию int(*string*, 2), а также двоичные литералы 0bNNN и код типа **b**, употребляемый при вызове метода str.format().

bool([*X*])

Возвращает логическое значение объекта *X*, используя стандартную процедуру проверки на истинность. Если же объект *X* имеет ложное логическое значение или пропускается, эта функция возвращает логическое значение False, а иначе — логическое значение True. Кроме того, словом bool обозначается класс, производный от типа int. Класс bool не подлежит далее подклассификации, и единственными его экземплярами являются логические значения True и False.

bytearray([*arg* [, *кодировка* [, *ошибки*]]])

Возвращает новый массив байтов. Тип bytearray представляет изменяемую последовательность мелких целых значений в пределах от **0** до **255**, которая выводится, если это возможно, текстом в коде ASCII. По существу, это изменяемый вариант типа bytes, поддерживающий большинство операций над изменяемыми последовательностями, а также большинство методов строкового типа str. В качестве аргумента *arg* может быть указана символьная строка с названием *кодировки* (а возможно, и *ошибок*), как и в описываемой далее функции str(); целочисленный размер для инициализации массива пустыми (NULL) байтами (т.е. нулевыми значениями); итерируемый объект мелких числовых значений, используемых для инициализации массива, например, в виде символьной строки типа bytes

или даже bytearray; объект, соответствующий интерфейсу типа "представления памяти", называвшемуся ранее буфером, для инициализации массива; или же ничего вообще не указывается, чтобы создать массив нулевой длины. Дополнительно см. ранее подраздел "Символьные строки типа bytes и bytearray".

bytes([*arg* [, *кодировка* [, *ошибки*]]])

Возвращает новый объект типа bytes, который представляет неизменяемую последовательность целочисленных значений в пределах от **0** до **255**. Тип bytes является разновидностью типа bytearray. У него имеются те же самые методы для операций над неизменяемыми символьными строками и последовательностями. Он обычно используется для представления 8-разрядных байтовых символьных строк двоичных данных (например, медиатекста или текста в уникоде). Аргументы конструктора интерпретируются таким же образом, как и в функции bytearray(). В версии Python 3.X объекты типа bytes могут быть также созданы с помощью литерала b'ccc', а в версии 2.X при этом создается обычный строковый объект типа str. Дополнительно см. ранее подраздел "Символьные строки типа bytes и bytearray".

callable(*объект*)

Возвращает логическое значение True, если ***объект*** является вызываемым, а иначе — логическое значение False. Эту функцию можно вызывать в версии Python 2.X, тогда как в версии Python 3.X она была сначала исключена из выпусков 3.0 и 3.1, но затем снова возвращена в выпуске 3.2, поэтому в прежних версиях следует использовать вызов функции hasattr(*объект*, '__call__').

chr(*I*)

Возвращает строку, состоящую из одного символа, кодовую точку которого в уникоде обозначает целочисленный аргумент *I*. Эта функция выполняет действие, обратное функции ord(). Например, в результате вызова функции chr(97) получается символьная строка 'a', а в результате вызова функции ord('a') — код **97** символа a.

classmethod(*функция*)

Возвращает метод класса для указанной *функции*. В качестве первого неявного аргумента метод класса получает ближайший (по иерархии) класс экземпляра субъекта подобно тому, как метод экземпляра получает экземпляр. Эта функция удобна для управления данными в отдельных классах. Начиная с версии 2.4 допускается использовать форму `@classmethod` декоратора функции (см. ранее раздел "Оператор `def`").

**compile(*строка*, *имя_файла*, *вид*
[, *признаки*[, *без_наследования*]])**

Компилирует *строку* в объект кода. В качестве аргумента *строка* указывается символьная строка, содержащая код программы на Python, а в качестве аргумента *имя_файла* — символьная строка, используемая для вывода сообщений об ошибках (обычно содержит имя файла, из которого прочитан код, или символьную строку `'<string>'`, если вводится в диалоговом режиме). В качестве аргумента *вид* указывается символьная строка `'exec'`, если *строка* содержит операторы; символьная строка `'eval'`, если *строка* содержит выражение; или же символьная строка `'single'`, если требуется вывести результат вычисления оператора выражения, отличающийся от `None`. Получающийся в итоге объект кода может быть выполнен в результате вызова встроенной функции `exec()` или `eval()`. Два последних необязательных аргумента данной функции определяют, какие именно операторы будут в дальнейшем оказывать воздействие на компиляцию символьной строки. Если эти аргументы отсутствуют, символьная строка компилируется с учетом последующих операторов на месте вызова функции `compile()` (подробнее об этом см. в документации на Python).

complex([*real* [, *imag*]])

Формирует объект комплексного числа, обозначаемого также суффиксом *J* или *j*: *real+imagJ*. По умолчанию аргумент *imag* равен нулю. Если же оба аргумента, *real* и *imag*, опущены, то возвращается значение `0j`.

delattr(*объект*, *имя*)

Удаляет атрибут *имя* (в виде символьной строки) из заданного *объекта*. Действует аналогично операции `del` *объект*`.`*имя*, но *имя* является символьной строкой, а не взятой буквально переменной (например, вызов функции `delattr(a, 'b')` аналогичен операции `del a.b`).

dict([*отображение* | *итерируемый_объект* | *ключевые_слова*])

Возвращает новый словарь, инициализируемый из отображения; последовательность или другой итерируемый объект, состоящий из пар "ключ–значение"; или же ряд именованных аргументов. Если же ни один из аргументов не задан, эта функция возвращает пустой словарь. Обозначает также имя класса подклассифицируемого типа.

dir([*объект*])

Если ни один из аргументов этой функции не передается, то возвращается список имен в текущей локальной области действия (пространстве имен). Когда же в качестве аргумента передается любой *объект* с атрибутами, эта функция возвращает список имен атрибутов, связанных с этим *объектом*. Эта функция подходит для обращения с модулями, классами и экземплярами классов, а также встроенными объектами с атрибутами (списками, словарями и т.д.). В результат ее выполнения входят наследуемые атрибуты, а кроме того, получаемый результат сортируется. В то же время атрибуты `__dict__` следует применять для формирования списка атрибутов только из одного объекта. В результате вызова данной функции выполняется специальный метод *объект*`.__dir__()`, если таковой определен. В итоге могут быть предоставлены имена вычисленных атрибутов в динамических классах или классах-посредниках.

divmod(*X*, *Y*)

Возвращает кортеж (`X / Y, X % Y`).

enumerate(*итерируемый_объект*, *начало*=0)

Возвращает итерируемый объект типа enumerate. В качестве аргумента *итерируемый_объект* должна быть указана последовательность или другой итерируемый объект, поддерживающий протокол итерации. Метод __next__() итератора, возвращаемый функцией enumerate(), в свою очередь, возвращает кортеж, содержащий *подсчет* (от *начала* или нуля по умолчанию) и соответствующее *значение*, получаемое в результате перебора *итерируемого_объекта*. Эта функция удобна для получения индексируемых рядов позиций и элементов в таких контекстах итерации, как циклы for (например, (0, x[0]), (1, x[1]), (2, x[2]), ...). Эта функция стала доступна в версии 2.3. О фиксированных перечислениях в версии Python 3.4 см. далее раздел "Модуль enum".

eval(*выражение* [, *глобальные* [, *локальные*]])

Вычисляет *выражение*, которое должно быть представлено символьной строкой, содержащей выражение Python, или объектом скомпилированного кода. Вычисляется *выражение* в областях действия пространства имен самой вызываемой функции eval(), если только не переданы аргументы *глобальные* и/или *локальные* из словаря пространств имен. Если передается только аргумент *глобальные*, то по умолчанию им становится и аргумент *локальные*. При вызове этой функции возвращается результат вычисления *выражения*. О предварительной компиляции см. ранее описание функции compile(), а о выполнении символьных строк с операторами — следующее ниже описание функции exec(). *Совет:* этой функцией не следует пользоваться для вычисления символьных строк ненадежного кода, поскольку они выполняются как программный код.

exec(*операторы* [, *глобальные* [, *локальные*]])

Вычисляет аргумент *операторы*, который должен быть представлен символьной строкой, содержащей операторы Python, или объектом скомпилированного кода. Если

аргумент *операторы* представлен символьной строкой, эта строка подвергается синтаксическому анализу как набор операторов Python, который затем выполняется в отсутствие синтаксических ошибок. Если же этот аргумент представлен объектом кода, он просто выполняется. Аргументы *глобальные* и *локальные* действуют таким же образом, как и в функции `eval()`, а функция `compile()` может быть использована для предварительной компиляции объектов кода. В версии Python 2.X эта функция доступна в виде оператора (см. ранее раздел "Операторы в версии Python 2.X") и в истории развития Python не раз меняла свою форму с функции на оператор, и наоборот. *Совет:* этой функцией не следует пользоваться для вычисления символьных строк ненадежного кода, поскольку они выполняются как программный код.

`filter(`*`функция, итерируемый_объект`*`)`

Возвращает те элементы *итерируемого_объекта*, для которых *функция* возвращает истинное значение, причем *функция* принимает один параметр. Если же аргумент *функция* равен `None`, то возвращаются все истинные элементы *итерируемого_объекта*, что равнозначно передаче *функции* встроенного типа `bool` в качестве параметра.

В результате вызова этой функции в версии 2.X возвращается *список*, а в версии Python 3.X — *итерируемый объект*, формирующий значения по требованию и допускающий обход только один раз. Для того чтобы получить нужные результаты, в оболочку этой функции следует заключить вызов функции `list()`.

`float([`*`X`*`])`

Преобразует число или символьную строку *X* в число с плавающей точкой (или число `0.0`, если аргумент не передается). Примеры применения этой функции см. ранее в разделе "Числа". Обозначает также имя класса подклассифицируемого типа.

format(*значение* [, *спецификация_формата*])

Преобразует *значение* объекта в форматированное представление, которое определяет *спецификация_формата*. Интерпретация аргумента *спецификация_формата* зависит от типа аргумента *значение*. В большинстве встроенных типов используется стандартный синтаксис форматирования, описанный ранее вместе с методом форматирования строк (см. ранее подраздел "Синтаксис метода форматирования"). В функции format(*значение*, *спецификация_формата*) делается вызов *значение*.__format__(*спецификация_формата*), и это основная операция метода str.format(). Например, вызов функции format(1.3333, '.2f') равнозначен операции '{0:.2f}'.format(1.3333).

frozenset([*итерируемый_объект*])

Возвращает объект *закрепленного множества*, элементы которого извлекаются из *итерируемого_объекта*. Закрепленные множества являются неизменяемыми, и поэтому у них отсутствуют методы обновления. В то же время они допускают вложение в другие множества.

getattr(*объект*, *имя* [, *по_умолчанию*])

Возвращает *имя* атрибута (символьную строку) из заданного *объекта*. Действует аналогично операции *объект*.*имя*, но *имя* вычисляется как символьная строка, а не взятое буквально имя переменной (например, вызов функции getattr(a, 'b') равнозначен операции a.b). Если именованный атрибут отсутствует, возвращается значение *по_умолчанию*, при условии, что оно предоставляется, а иначе генерируется исключение типа AttributeError.

globals()

Возвращает словарь, содержащий глобальные переменные в вызывающем коде (например, имена переменных в объемлющем модуле).

hasattr(*объект*, *имя*)

Возвращает логическое значение `True`, если *объект* содержит атрибут *имя* (в виде символьной строки), а иначе — логическое значение `False`.

hash(*объект*)

Возвращает хеш-значение *объекта*, если таковое в нем имеется. Хеш-значения служат как целочисленные значения для быстрого сравнения ключей во время поиска в словаре. Делает вызов *объект*`.__hash__()`.

help([*объект*])

Вызывает встроенную справочную систему. Эта функция предназначена для применения в диалоговом режиме. Если аргумент ей не предоставляется, то сеанс работы со справочной системой в диалоговом режиме начинается на консоли интерпретатора Python. А если аргумент предоставляется в виде символьной строки, то он служит в качестве имени для поиска модуля, функции, класса, метода, ключевого слова или раздела документации и последующего отображения текста справки. Если же аргумент оказывается объектом любого другого типа, то составляется справка по этому объекту. Например: `help(list.pop)`.

hex(*N*)

Преобразует целое число *N* в символьную строку шестнадцатеричных цифр (по основанию **16**). Если аргумент *N* не является объектом типа `int` в Python, то в версии Python 3.X должен быть определен метод `__index__()`, возвращающий целое значение. А в версии 2.X вместо этого вызывается метод `__hex__()`.

id(*объект*)

Возвращает целочисленное значение, однозначно определяющее идентичность объекта для вызывающего процесса среди всех существующих объектов (например, его адрес в оперативной памяти).

__import__(*имя*, [...*другие аргументы*...])

Импортирует и возвращает модуль, если его *имя* представлено символьной строкой во время выполнения, например `mod = __import__('mymod')`. Вызов этой функции, как правило, выполняется быстрее, чем построение и выполнение символьной строки для оператора `import` с помощью функции `exec()`. Эта функция вызывается в операторах `import` и `from` и может быть переопределена для специальной настройки операций импорта. Все ее аргументы, кроме первого, играют дополнительные роли (см. руководство по библиотеке Python). См. также описание модуля `imp` и вызова `importlib.import_module()` в стандартной библиотеке и ранее раздел "Оператор `import`".

input([*приглашение*])

Выводит *приглашение* в виде символьной строки, если таковое задано, а затем читает строку из стандартного потока ввода `sys.stdin` и возвращает ее в виде символьной строки. Завершающие символы `\n` в конце строки отбрасываются, а по достижении конца стандартного потока ввода генерируется исключение типа `EOFError`. На тех платформах, где поддерживается библиотека `readline` из проекта GNU, используется функция `input()`. В версии Python 2.X эта функция называется `raw_input()`.

int([*число* | *строка* [, *основание*]])

Преобразует число или символьную строку в обычное целое значение. При преобразовании чисел с плавающей точкой они округляются до нуля. Аргумент *основание* может быть передан только в том случае, если первый аргумент является символьной строкой, и по умолчанию он принимает значение **10**. Если же аргумент *основание* передается с нулевым значением, то основание системы счисления определяется из содержимого символьной строки (в виде кодового литерала), а иначе значение передаваемого аргумента *основание* используется для преобразования символьной строки в числовое значение по заданному основанию. Аргумент *основание* может принимать

нулевое значение или же значение от **2** до **36**. В отсутствие аргументов эта функция возвращает нулевое значение. Примеры ее применения см. ранее в разделе "Числа". Обозначает также имя класса подклассифицируемого типа.

isinstance(*объект, сведения_о_классе*)

Возвращает логическое значение `True`, если заданный *`объект`* является экземпляром класса, определяемого аргументом *`сведения_о_классе`*, или любого его подкласса. Аргумент *`сведения_о_классе`* может быть кортежем классов и/или типов. В версии Python 3.X типы представлены классами, и поэтому в ней не предусмотрены особые случаи для типов. А в версии Python 2.X второй аргумент этой функции может быть также объектом заданного типа, благодаря чему она оказывается удобной в обеих версиях Python в качестве альтернативы средствам проверки типов, например: `isinstance(`*`X, Type`*`)` вместо `type(`*`X`*`) is `*`Type`*.

issubclass(*класс1, класс2*)

Возвращает логическое значение `True`, если *`класс1`* является производным от *`класса2`*. Аргумент *`класс2`* может быть также кортежем классов.

iter(*объект* [, *сигнальная_метка*])

Возвращает объект итератора, который может быть использован для перебора всех элементов в итерируемом *`объекте`*. У возвращаемых объектов итераторов имеется метод `__next__()`, возвращающий следующий элемент или генерирующий исключение типа `StopIteration`, если продвигаться дальше некуда. Этот протокол используется во всех контекстах итерации в Python для продвижения к следующему элементу, если он поддерживается итерируемым *`объектом`*. Вызов метода *`I`*`.__next__()` делается автоматически и во встроенной функции `next(`*`I`*`)`. Если данной функции передается один аргумент, то *`объект`* должен предоставить свой итератор, или же он должен быть последовательностью. А если ей передаются два аргумента, то *`объект`* вызывается до тех пор, пока не возвратится

сигнальная_метка. Вызов функции `iter()` может быть перегружен в классах методом `__iter__()`.

В версии Python 2.X у объектов итераторов имеется метод `next()`, заменяющий метод `__next__()`. Для прямой и обратной совместимости в версии 2.X доступна также встроенная функция `next()`, начиная с выпуска 2.6. В этой функции делается вызов *I*`.next()` вместо вызова *I*`.__next__()`. А до выпуска 2.6 вызов *I*`.next()` можно было делать явным образом. См. также приведенное выше описание встроенной функции `next()` и подраздел "Протокол итерации".

`len(`*объект*`)`

Возвращает количество элементов (длину) коллекции, представленной аргументом *объект*, который может быть последовательностью, отображением, множеством или чем-то другим (например, определяемой пользователем коллекцией).

`list([`*итерируемый_объект*`])`

Возвращает новый список, содержащий все элементы в любом *итерируемом_объекте*. Если аргумент *итерируемый_объект* уже является списком, эта функция возвращает (неполную) копию данного объекта. В отсутствие аргументов возвращается новый пустой список. Обозначает также имя класса подклассифицируемого типа.

`locals()`

Возвращает словарь, содержащий локальные переменные в вызывающем коде (по одной записи *ключ*`:`*значение* на каждую локальную переменную).

`map(`*функция*`,` *итерируемый_объект*`[,`
** *итерируемый_объект*`]*)`**

Применяет *функцию* к каждому элементу любой последовательности или другого *итерируемого_объекта* и возвращает отдельные результаты. Например, в результате

вызова функции `map(abs, (1, -2))` возвращаются значения **1** и **2**. Если же передаются дополнительные аргументы *итерируемый_объект*, то *функция* должна принимать столько аргументов, сколько ей передается, и при каждом ее вызове ей передается один элемент из *итерируемого_объекта*. В этом режиме итерация прекращается по достижении конца самого короткого итерируемого объекта.

В версии Python 2.X эта функция возвращает *список* результатов отдельных вызовов. А в версии Python 3.X возвращается *итерируемый объект*, формирующий результаты по требованию и допускающий обход только один раз. Для того чтобы получить нужные результаты, в оболочку этой функции следует заключить вызов функции `list()`.

А если аргумент *функция* принимает значение `None` в версии Python 2.X (но не в версии Python 3.X), то функция `map()` накапливает все элементы *итерируемого_объекта* в списке результатов. При наличии нескольких итерируемых объектов результаты обхода их элементов объединяются в кортежи, а все итерируемые объекты заполняются значениями `None` по длине самого длинного объекта. Аналогичная возможность имеется и в модуле `itertools` из стандартной библиотеки в версии Python 3.X.

`max(`*`итерируемый_объект`*`[, arg] * [, key=`*`функция`*`])`

Если этой функции передается единственный аргумент *итерируемый_объект*, то она возвращает элемент с наибольшим значением из непустого итерируемого объекта (например, символьную строку, кортеж, список, множество). А если этой функции передается несколько аргументов, то она возвращает наибольшее значение среди всех аргументов. Дополнительный, но не обязательный и только именованный аргумент `key` обозначает функцию с единственным аргументом для преобразования значения, аналогичную той, что применяется в вызовах метода `list.sort()` и функции `sorted()` (см. ранее раздел "Списки" и далее описание функции `sorted()`).

memoryview(*объект*)

Возвращает объект представления памяти, создаваемый из заданного аргумента *объект*. Представления памяти дают возможность получать из кода Python доступ к внутренним данным объекта, поддерживающего протокол без копирования объекта. Память может быть интерпретирована как простая последовательность байтов или более сложная структура данных. К числу встроенных типов, поддерживающих протокол представления памяти, относятся типы `bytes` и `bytearray`. Подробнее об этом см. в документации на Python. Представления памяти почти полностью заменили протокол буферизации и встроенную функцию `buffer()` в версии Python 2.X, хотя функция `memoryview()` вставлена в качестве "заплаты" в выпуск 2.7 ради совместимости с версией 3.X.

min(*итерируемый_объект* [, *arg*] * [,key=*функция*])

Если этой функции передается единственный аргумент *итерируемый_объект*, то она возвращает элемент с наименьшим значением из непустого итерируемого объекта (например, символьную строку, кортеж, список, множество). А если этой функции передается несколько аргументов, то она возвращает наименьшее значение среди всех аргументов. Дополнительный аргумент `key` выполняет ту же самую роль, что и в описанной выше функции `max()`.

next(*итератор* [, *по_умолчанию*])

Извлекает следующий элемент из объекта *итератор*, вызывая его метод `__next__()` (в версии 3.X). Если *итератор* исчерпан, то возвращается значение *по_умолчанию*, при условии, что оно предоставляется, а иначе генерируется исключение типа `StopIteration`. Эта функция доступна также в версиях 2.6 и 2.7, где она делает вызов *итератор*`.next()` вместо вызова *итератор*`.__next__()`. Благодаря этому сохраняется прямая совместимость версии Python 2.X с версией 3.X, а также обратная совместимость версии Python 3.X с версией 2.X. До выпуска 2.6 такой вызов в версии Python 2.X отсутствовал. Вместо этого

для ручной итерации следует сделать вызов *итератор*.next(). См. выше описание функции iter() и подраздел "Протокол итерации".

object()

Возвращает новый бессодержательный объект. Имя object буквально обозначает суперкласс для всех классов нового стиля, включая и все классы в версии Python 3.X, а также классов, непосредственно производных от класса object в версии Python 2.X. У этого класса имеется ряд методов по умолчанию (см. выше описание функции dir(*объект*)).

oct(*N*)

Преобразует число *N* в символьную строку восьмеричных цифр (по основанию **8**). Если аргумент *N* не является объектом типа int, то в версии Python 3.X должен быть определен метод __index__(), возвращающий целое значение. А в версии Python 2.X вместо этого вызывается метод __oct__().

open(...)

```
open(файл
     [, mode='r'
     [, buffering=-1
     [, encoding=None          # Только в текстовом режиме
     [, errors=None            # Только в текстовом режиме
     [, newline=None           # Только в текстовом режиме
     [, closefd=True,          # Только для дескрипторов
    [, opener=None ]]]]]]])    # Специальное средство открытия
                               # файлов, начиная с версии 3.3
```

Возвращает новый *файловый объект*, связанный с внешним файлом, обозначаемым аргументом *файл*, а при неудачном исходе открытия файла генерирует исключение типа IOError (или производного от него типа OSError, начиная с выпуска 3.3). Здесь описывается функция open() для версии Python 3.X, а об особенностях ее применения в версии Python 2.X см. ниже, в разделе "Встроенные функции в версии Python 2.X".

Аргумент *файл* обычно является строковым объектом, содержащим текст или байты и предоставляющим имя открываемого файла, включая путь к нему, если он не находится в текущем рабочем каталоге. Аргумент *файл* может быть также целочисленным дескриптором открываемого файла. Если дескриптор файла задан, то он закрывается вместе с возвращаемым объектом потока ввода-вывода, при условии, что не установлено логическое значение `False` параметра `closefd`. Все описываемые далее параметры могут быть переданы как именуемые аргументы.

Параметр `mode` является необязательной символьной строкой, обозначающей режим открытия файла. По умолчанию это символьная строка `'r'`, обозначающая открытие файла для чтения в текстовом режиме. Параметр `mode` может принимать и другие строковые значения, в том числе `'w'` для записи данных (если файл уже существует, то он урезается), а также `'a'` для присоединения данных. Если параметр `encoding` не указан в текстовом режиме, то используется кодировка, зависящая от конкретной платформы, а символы новой строки по умолчанию преобразуются в символьные строки `'\n'`, и наоборот. Для чтения и записи исходных байтов в файл следует выбрать режим `'rb'`, `'wb'` или `'ab'`, не указывая параметр `encoding`.

Следующие доступные режимы открытия файла могут использоваться в определенном сочетании: `'r'` — для чтения данных (по умолчанию); `'w'` — для записи данных с предварительным урезанием файла; `'a'` — для записи данных, присоединяемых в конце файла, если он существует; `'b'` — для перехода в двоичный режим; `'t'` — для перехода в текстовый режим (по умолчанию); `'+'` — для открытия файла на диске с целью его обновления (чтения и записи); `'U'` — для перехода в режим универсального перевода строки (только ради обратной совместимости). Выбираемый по умолчанию режим `'r'` аналогичен режиму `'rt'` (открытия файла для чтения текста). Для произвольного доступа к двоичным данным в файле служит режим `'w+b'`, в

котором файл открывается и урезается до нуля байтов, тогда как в режиме `'r+b'` файл открывается без урезания.

В Python различаются файлы, открываемые в двоичном и текстовом режимах, даже если такая возможность отсутствует в базовой операционной системе. Ниже описываются особенности работы этих режимов.

- Для *ввода* файл открывается в двоичном режиме (путем присоединения строки `'b'` к параметру `mode`) и возвращается содержимое объектов типа `bytes` без декодирования данных в уникоде или преобразования символов конца строки. В текстовом режиме, выбираемом по умолчанию или в результате присоединения строки `'t'` к параметру `mode`, содержимое файла возвращается в виде символьных строк типа `str` после декодирования байтов с помощью кодировки в уникоде, явно указываемой в параметре `encoding` или же выбираемой по умолчанию на базовой платформе. При этом символы конца строки преобразуются в соответствии с установкой параметра `newline`.
- Для *вывода* в двоичном режиме предполагаются данные типа `bytes` или `bytearray`, которые записываются в файл без изменения. Для вывода в текстовом режиме предполагаются данные типа `str`, которые кодируются в уникоде в соответствии с установкой параметра `encoding`, а символы перевода строки преобразуются в соответствии с установкой параметра `newline` перед записью данных в файл.

Параметр `buffering` является необязательным целым значением, используемым для установки правил буферизации данных. По умолчанию, когда этот параметр не передается или принимает значение `-1`, данные буферизуются полностью. Для отключения буферизации следует передать нулевое значение данного параметра, хотя это разрешается только в двоичном режиме. Для буферизации строк (только в текстовом режиме) следует передать значение `1` данного параметра, а для полной буферизации — целое значение

больше 1, обозначающее размер буфера. Передача буферизованных данных не может быть выполнена немедленно, поэтому для принудительного опорожнения буфера следует сделать вызов *файл*.flush().

Параметр encoding задает наименование кодировки, используемой для кодирования или декодирования текстового содержимого выводимого в файл или вводимого из файла. Этот параметр следует указывать только в текстовом режиме. По умолчанию выбирается кодировка на базовой платформе (она получается в результате вызова функции locale.getpreferredencoding()). Но с помощью данного параметра может быть передана любая кодировка, которая поддерживается в Python. Перечень поддерживаемых кодировок см. в описании модуля codecs из стандартной библиотеки Python.

Параметр errors является необязательной символьной строкой, обозначающей порядок обработки ошибок кодирования. Этот параметр следует указывать только в текстовом режиме. Он может быть передан в виде символьной строки 'strict' (по умолчанию — значения None), чтобы при появлении ошибок кодирования генерировалось исключение типа ValueError; символьной строки 'ignore', чтобы игнорировать ошибки кодирования, хотя это может привести к потере данных; символьной строки 'replace', чтобы использовать маркер замены недостоверных данных, и т.д. О допустимых кодах ошибок и средствах их обработки см. в документации на Python, описание функции codecs.register_error() из стандартной библиотеки Python, а также приведенное выше описание функции str().

Параметр newline определяет режим работы универсального перевода строк и применяется только в текстовом режиме. Он может принимать значение None (по умолчанию) или одну из следующих символьных строк: '', '\n', '\r' и '\r\n'.

- Если при *вводе* параметр `newline` равен `None`, то режим универсального перевода строк активизируется, и строки могут оканчиваться последовательностью символов `''`, `'\n'`, `'\r'` или `'\r\n'`, причем все они преобразуются в последовательность символов `'\n'` перед возвратом в вызывающий код. Если же параметр `newline` равен `''`, то и в этом случае режим универсального перевода строк активизируется, но символы окончания строки возвращаются в вызывающий код без преобразования. А если параметр `newline` принимает любое другое допустимое значение, то вводимые строки завершаются только заданной последовательностью символов, причем символы окончания строки возвращаются в вызывающий код без преобразования.
- Если при *выводе* параметр `newline` равен `None`, то любая последовательность записываемых символов `'\n'` преобразуется в разделитель строк, используемый в системе по умолчанию (константа `os.linesep`). Если же параметр `newline` равен `''`, то никакого преобразования символов конца строки не происходит. А если параметр `newline` принимает любое другое допустимое значение, то любая последовательность записываемых символов `'\n'` преобразуется в заданную последовательность символов.

Если параметр `closefd` равен `False`, базовый дескриптор файла будет сохранен открытым при закрытии файла. Но такой способ оказывается неработоспособным, когда имя файла задается в виде символьной строки, и в этом случае параметр `closefd` должен быть равен `True` (по умолчанию). Если же параметру `opener` передается вызываемый объект (начиная с выпуска 3.3), то дескриптор файла получается в результате вызова функции *`opener(файл, признаки)`* с такими же аргументами, как и для вызова функции `os.open()` (см. далее раздел "Системный модуль `os`").

Подробнее об интерфейсе объектов, возвращаемых функцией `open()`, см. ранее в разделе "Файлы". *Совет:* любой объект, поддерживающий интерфейс метода файлового объекта, может быть использован в тех контекстах, в которых предполагается файл (см., например, описание функции *`socketobj`*`.makefile()`, функций `io.StringIO(`*`str`*`)` и `io.BytesIO(`*`bytes`*`)` в версии Python 3.X, функции `StringIO.stringIO(`*`str`*`)` в версии Python 2.X, а также всей стандартной библиотеки Python).

НА ЗАМЕТКУ

Файловый режим подразумевает оба варианта конфигурации и строковые типы данных в версии 3.X, поэтому функцию `open()` следует рассматривать с точки зрения двух разных разновидностей этого режима: текстовой и двоичной, как пояснялось выше при описании параметра `mode` данной функции. Разработчики Python выбрали перегрузку единственной функции для поддержки двух типов файлов, задавая режим работы с файлами с помощью аргументов и изменяя тип его содержимого, вместо того чтобы предоставить две разные функции `open()`. А в базовой библиотеке классов модуля `io`, для которой функция `open()` служит внешним интерфейсом в версии 3.X, типы файловых объектов *различаются* по отдельным режимам. Подробнее о модуле `io` см. в документации на Python. Этот модуль доступен и в версии 2.X, начиная с выпуска 2.6, в качестве альтернативы встроенному типу `file`, но в то же время он является обычным файловым интерфейсом для функции `open()` в версии 3.X.

`ord(`*`C`*`)`

Возвращает целочисленное значение кодовой точки единственного символа в строке *C*. Для символов в коде ASCII это 7-разрядный код символа *C*, а в общем, — кодовая

точка в уникоде единственного символа в строке *C*. См. также приведенное выше описание функции `chr()`, обратной данной функции.

`pow(X, Y[, Z])`

Возвращает значение *X* в степени *Y* (дополнительно по модулю *Z*). Действует аналогично оператору выражения **.

`print(...)`

```
print([объект [, объект]*]
      [, sep=' '] [, end='\n']
      [, файл=sys.stdout] [, flush=False])
```

Выводит необязательный *объект* (или несколько таких объектов) в поток *файл*, разграничивая их разделителем `sep`, завершая последовательностью символов `end` и дополнительно очищая поток после вывода, при условии, что установлено соответствующее значение аргумента `flush`. Если указываются четыре последних аргумента, они должны быть именованными, а их значения по умолчанию приведены выше. Аргумент `flush` стал доступен в выпуске Python 3.3.

Значения всех неименованных аргументов сначала преобразуются в символьные строки с помощью аналога функции `str()`, а затем выводятся в поток. Оба аргумента, `sep` и `end`, должны быть символьными строками или равными `None` (в этом случае используются их значения по умолчанию). Если ни один из аргументов *объект* не задан, то выводится последовательность символов `end`. В качестве аргумента *файл* может быть задан объект с методом `write(`*строка*`)`, но совсем не обязательно конкретный файл. Если этот аргумент не передается данной функции или принимает значение `None`, то вывод выполняется в стандартный поток `sys.stdout`.

В версии Python 2.X функциональные возможности вывода доступны в форме оператора. Подробнее об этом см. ранее в разделе "Оператор `print`".

property([*fget*[, *fset*[, *fdel*[, *doc*]]]])

Возвращает атрибут свойства для классов нового стиля (производных от класса `object` и всех классов в версии 3.X). Аргумент *fget* обозначает функцию для получения значения атрибута, аргумент *fset* — функцию для установки значения атрибута, тогда как аргумент *fdel* — функцию для удаления этого значения. Вызов этой функции можно использовать как ее декоратор (`@property`). В этом случае она возвращает объект с методами *getter*(), *setter*() и *deleter*(), которые также могут служить декораторами (см. ранее раздел "Оператор `def`"). Эта функция реализуется вместе с декораторами (см. ранее раздел "Методы для операций с дескрипторами").

range([*начало*,] *конец* [, *шаг*])

Возвращает последовательный ряд чисел в пределах, задаваемых аргументами *начало* и *конец*. Если этой функции передается один аргумент, она возвращает целые значения в пределах от **0** до ***конец*-1**. Если же этой функции передаются два аргумента, она возвращает целые значения в пределах от ***начало*** до ***конец*-1**. А если ей передаются три аргумента, то она возвращает целые значения в пределах от ***начало*** до ***конец*-1**, но добавляя к каждому предыдущему значению заданный *шаг*. По умолчанию аргумент *начало* равен **0**, тогда как аргумент *конец* — **1**.

Аргумент *шаг* может быть больше **1** для пропуска значений. Например, в результате вызова функции `range(0, 20, 2)` получается ряд четных чисел в пределах от **0** до **18**. Этот аргумент может быть и отрицательным для обратного отсчета от наибольшего значения, определяемого аргументом *начало*. Например, в результате вызова функции `range(5, -5, -1)` получается ряд чисел в пределах от **5** до **-4**. С помощью данной функции нередко формируются смещенные списки или подсчеты повторов в циклах `for` и других контекстах итерации.

В версии Python 2.X эта функция возвращает *список*, а в версии Python 3.X — *итерируемый объект*, формирующий значения по требованию и допускающий многократный обход своих элементов. Для того чтобы получить нужные результаты, в оболочку этой функции следует заключить вызов функции `list()`.

`repr(`*`объект`*`)`

Возвращает строковое представление *`объекта`* на низком уровне восприятия кода. Как правило, возвращаемая строка принимает форму, которая может быть подвергнута синтаксическому анализу в функции `eval()`, или предоставляет больше подробностей, чем описанная ранее функция `str()`. Эта функция равнозначна выражению *`'объект'`*, но только в версии Python 2.X, поскольку выражения, заключаемые в обратные кавычки, исключены из версии Python 3.X. См. описание метода `__repr__()` выше, в разделе "Методы перегрузки операторов".

`reversed(`*`последовательность`*`)`

Возвращает обратный итерируемый объект. Аргумент *`последовательность`* должен быть объектом, имеющим метод `__reversed__()` и поддерживающим протокол последовательности (методы `__len__()` и `__getitem__()`, вызываемые с целочисленными аргументами, начиная с нуля).

`round(`*`X`*`[, `*`N`*`])`

Возвращает числовое значение *X* с плавающей точкой, округленное до *N* цифр после десятичной точки. По умолчанию аргумент *N* равен нулю, но он может быть и отрицательным для обозначения цифр слева от десятичной точки. Возвращаемое значение является целым, если эта функция вызывается с одним аргументом, а иначе оно такого же типа, как и значение *X*. Результат вызова данной функции всегда является числом с плавающей точкой, но только в версии Python 2.X. А в версии Python 3.X (и только в ней) в этой функции делается вызов *X*`.__round__()`.

set([*итерируемый_объект*])

Возвращает множество, элементы которого извлекаются из заданного *итерируемого_объекта*. Эти элементы должны быть изменяемыми. Для представления множества множеств вложенные множества должны быть объектами типа `frozenset`. Если аргумент *итерируемый_объект* не указан, то возвращается новое пустое множество. Эта функция стала доступной в версии 2.4. См. описание множеств и их литералов `{...}` ранее в разделе "Множества".

setattr(*объект*, *имя*, *значение*)

Присваивает *значение* атрибуту *имя* (в виде символьной строки) в заданном *объекте*. Действует аналогично выражению *объект*`.`*имя* `=` *значение*, но *имя* вычисляется как символьная строка, а не взятое буквально имя переменной. Например, вызов функции `setattr(a, 'b', c)` равнозначен выражению `a.b = c`.

slice([*начало*,] *конец* [, *шаг*])

Возвращает объект нарезки, представляющий заданные пределы с доступными только для чтения атрибутами *start* (аргумент *начало*), *stop* (аргумент *конец*) и *step* (аргумент *шаг*), любой из которых может принимать значение `None`. Аргументы этой функции интерпретируются таким же образом, как и аргументы функции `range()`. Объекты нарезки могут быть использованы вместо обозначения нарезки *i*`:`*j*`:`*k*. Например, выражение *X*`[`*i*`:`*j*`]` равнозначно выражению *X*`[slice(`*i*`,` *j*`)]`.

sorted(*итерируемый_объект*, key=None, reverse=False)

Возвращает новый отсортированный список, состоящий из элементов заданного *итерируемого_объекта*. Дополнительные, но не обязательные именованные аргументы `key` и `reverse` имеют такое же назначение, как и аналогичные аргументы метода `list.sort()`, описанного ранее в разделе "Списки". В частности, аргумент `key` обозначает функцию преобразования с одним аргументом.

Эта функция подходит для любого итерируемого объекта и возвращает новый объект вместо непосредственного изменения списка, а следовательно, ее удобно применять в циклах `for` и прочих контекстах итерации, чтобы исключить выделение вызовов средств сортировки в отдельные операторы вследствие возврата значений `None`. Стала доступной в версии 2.4.

В версии Python 2.X эта функция имеет сигнатуру вызова `sorted(`*`итерируемый_объект`*`, cmp=None, key=None, reverse=False)`, где дополнительные, но не обязательные аргументы `cmp`, `key` и `reverse` имеют такое же назначение, как и аналогичные аргументы в методе `list.sort()` для версии Python 2.X, описанном ранее в разделе "Списки".

`staticmethod(`*`функция`*`)`

Возвращает статический метод для заданной *функции*. Статический метод не получает экземпляр в виде подразумеваемого первого аргумента, и поэтому он удобен для обработки атрибутов класса, включая его экземпляры. Начиная с версии 2.4 можно пользоваться декоратором функции `@staticmethod` (см. ранее раздел "Оператор `def`"). В версии Python 3.X (и только в ней) эта встроенная функция не требуется для простых функций, вызываемых в классах только через их объекты, а не экземпляры.

`str([`*`объект`*`[, `*`кодировка`*`[, `*`ошибки`*`]])`

Обозначает также имя класса подклассифицируемого типа и действует в одном из двух режимов, определяемых в версии Python 3.X по следующим образцам вызова.

- *Вывод символьной строки на печать.* Когда задается только *`объект`*, эта функция возвращает печатаемое строковое представление *`объекта`* на высоком уровне, удобном для восприятия пользователем. Что касается символьных строк, то это сама символьная строка. В отличие от функции `repr(`*`X`*`)`, функция `str(`*`X`*`)` не всегда пытается возвратить символьную строку, воспринимаемую функцией `eval()`. Ее назначение — возвратить

удобочитаемую и печатаемую символьную строку. В отсутствие аргументов эта функция возвращает пустую символьную строку. См. также описание метода `__str__()`, вызываемого в данном режиме, выше, в разделе "Методы перегрузки операторов".

- *Декодирование в уникоде.* Если данной функции передаются аргументы *кодировка* и/или *ошибки*, то объект, представляющий символьную строку, состоящую из байтов, или буфер символов будет соответственно декодирован с помощью кодера-декодера, определяемого аргументом *кодировка*. Аргумент *кодировка* является символьной строкой, предоставляющей название кодировки в уникоде. Если же кодировка неизвестна, то возникает исключение типа `LookupError`. Обработка ошибок выполняется в соответствии со значением аргумента *ошибки*, которое может быть равным `'strict'` (по умолчанию), чтобы генерировать исключение типа `ValueError` при появлении ошибок кодирования; `'ignore'`, чтобы молча игнорировать ошибки и, возможно, потерять данные; или же `'replace'`, чтобы заменить вводимые символы, которые не могут быть декодированы, официально заменяющим символом в уникоде (**U+FFFD**). См. также описание модуля `codecs` из стандартной библиотеки Python, а также аналогичного метода `bytes.decode()`. Например, вызов `b'a\xe4'.decode('latin-1')` равнозначен вызову `str(b'a\xe4', 'latin-1')`.

В версии Python 2.X у этой функции имеется более простая сигнатура вызова `str([`*объект*`])`, в результате которого возвращается символьная строка, содержащая печатаемое представление *объекта* на высоком уровне восприятия, что равнозначно упомянутому выше первому образцу вызова в версии Python 3.X. Декодирование в уникоде реализуется в версии 2.X с помощью строковых методов или функции `unicode()`, которая, по существу, аналогична функции `str()` в версии 3.X.

sum(*итерируемый_объект* [, *начало*])

Суммирует *начало* и все элементы любого *итерируемого_объекта* и возвращает итоговую сумму. По умолчанию аргумент *начало* равен нулю. Как правило, элементы итерируемого объекта являются числами и могут быть символьными строками. *Совет:* для сцепления символьных строк итерируемого объекта следует воспользоваться формой вызова `''.join(`*итерируемый_объект*`)`.

super([*тип* [, *объект*]])

Возвращает суперкласс заданного *типа*. Если второй аргумент опущен, возвращаемый объект суперкласса оказывается непривязанным. Если же второй аргумент является объектом, то в результате вызова `isinstance(`*объект*`, `*тип*`)` должно быть возвращено истинное значение. Эта функция пригодна для всех классов в версии 3.X, но только для классов нового стиля в версии Python 2.X, где *тип* также не является необязательным.

Вызов функции `super()` (только в версии 3.X) без аргументов в методе класса неявно равнозначен вызову `super(`*содержащий_класс*`, `*аргумент_метода_экземпляра_self*`)`. Какой бы ни была эта форма (явной или неявной), она приводит к созданию привязанного объекта-заместителя, составляющего пару экземпляру *self* с доступом к местоположению вызывающего класса в порядке разрешения методов (по правилу MRO) из класса экземпляра *self*. Этот объект-заместитель может быть использован в последующих ссылках на атрибуты суперкласса и вызовах методов. Подробнее о порядке разрешения методов класса см. выше, в подразделе "Классы нового стиля: правило MRO".

Функция `super()` всегда выбирает *следующий* класс по правилу MRO, т.е. первый класс, следующий после вызывающего класса, имеющего запрашиваемый атрибут, будь то подлинный суперкласс или нет. Поэтому она может быть использована для переадресации вызовов методов.

В иерархическом дереве классов с единичным наследованием вызов данной функции может быть использован для обращения к родительским суперклассам вообще, не именуя их явно. А в иерархических деревьях классов с множественным наследованием вызов данной функции может быть использован для реализации совместной диспетчеризации вызовов методов, распространяющей вызовы по иерархическому дереву.

В последнем случае, т.е. при совместной диспетчеризации вызовов методов, эта функция может быть использована в ромбовидных иерархических деревьях, позволяя обращаться к каждому суперклассу лишь один раз по цепочке вызовов соответствующих методов. Но в то же время функция `super()` может проявлять слишком неявное поведение, из-за чего в некоторых программах обращение к суперклассам может осуществляться не так, как предполагается или требуется. Методика диспетчеризации вызовов методов, применяемая в функции `super()`, обычно налагает следующие ограничения.

- *Средства привязки.* Метод, вызываемый функцией `super()`, должен существовать, для чего требуется дополнительный код, если отсутствует привязка к цепочке вызовов.
- *Аргументы.* Метод, вызываемый функцией `super()`, должен иметь одну и ту же сигнатуру аргументов по всему иерархическому дереву классов, что нарушает гибкость этого дерева, особенно для таких методов на уровне реализации, как конструкторы.
- *Развертывание.* В каждом вхождении метода, вызываемого функцией `super()`, кроме последнего, должна использоваться сама функция `super()`, что может затруднить применение существующего кода, изменение порядка вызовов, перегрузку методов и программирование автономных классов.

В силу перечисленных выше ограничений вызов методов из суперкласса с явным указанием его имени вместо

обращения к функции super() в некоторых случаях может оказаться более простым, предсказуемым или даже необходимым способом. Так, традиционная форма *S.метод(self)* обращения к суперклассу *S* равнозначна неявной форме super().*метод*(). Подробнее об особом случае поиска атрибутов с помощью функции super() см. выше, в подразделе "Алгоритм наследования классов нового стиля". Вместо выполнения полного наследования получающиеся в итоге объекты просматривают зависящую от контекста завершающую часть списка по правилу MRO в иерархическом дереве классов, выбирая первый совпадающий дескриптор или значение.

tuple([*итерируемый_объект*])

Возвращает новый кортеж с теми же самыми элементами, что и у любого передаваемого *итерируемого_объекта*. Если аргумент *итерируемый_объект* уже является кортежем, то возвращается именно он, а не его копия. Этого оказывается достаточно, поскольку кортежи неизменяемы. Если же аргумент не указан, то возвращается новый пустой кортеж. Обозначает также имя класса подклассифицируемого типа.

type(*объект* | (*имя, базы, словарь*))

Обозначает также имя класса подклассифицируемого типа и применяется в двух разных режимах, определяемых по следующим образцам вызова.

- *С одним аргументом.* Возвращает объект, представляющий тип заданного *объекта*. Это удобно для проверки типов в условных операторах if, например type(*X*) ==type([]). См. описание объектов предустановленных типов, не являющихся встроенными именами, в документации на модуль types из стандартной библиотеки, а также приведенное выше описание функции isinstance(). В классах нового стиля вызов функции type(*объект*) обычно равнозначен выражению *объект*.__class__. В состав модуля types включены также синонимы большинства имен встроенных типов, но только в версии Python 2.X.

- *С тремя аргументами.* Служит в качестве конструктора, возвращающего объект нового типа. Это динамическая форма оператора `class`. Символьная строка *имя* обозначает имя класса и становится атрибутом `__name__`. Кортеж *базы* перечисляет базовые (супер) классы и становится атрибутом `__bases__`. И наконец, аргумент *словарь* представляет пространство имен, содержащее определения атрибутов для тела класса и становится атрибутом `__dict__`. Например, следующие выражения равнозначны:

```
class X(object): a = 1
X = type('X', (object,), dict(a=1))
```

- Такое преобразование употребляется для построения *метаклассов*, при котором подобные вызовы функции `type()` делаются автоматически и, как правило, приводят к вызову метода `__new__()` из метакласса и/или метода `__init__()` с аргументами для создания подклассов, производных от класса указанного *типа*.
- Дополнительно см. выше подразделы "Метаклассы", "Декораторы классов в версиях Python 2.6, 2.7 и 3.0" и раздел "Методы перегрузки операторов".

`vars([`*`объект`*`])`

Если аргумент не указан, эта функция возвращает словарь, содержащий имена переменных из текущей локальной области действия. Если же в качестве аргумента *объект* указан модуль, класс или экземпляр класса, то возвращается словарь, соответствующий пространству имен атрибутов заданного *объекта* (например, его атрибут `__dict__`). Полученный результат не должен быть видоизменен. *Совет:* эта функция удобна для обращения к переменным при форматировании символьных строк.

`zip([`*`итерируемый_объект`*`[, `*`итерируемый_объект`*`]*])`

Возвращает ряд кортежей, где каждый *i*-й кортеж содержит *i*-й элемент из каждого аргумента *итерируемый_объект*. Например, в результате вызова функции `zip('ab', 'cd')` возвращаются кортежи `('a', 'c')` и `('b', 'd')`.

Для вызова этой функции требуется указать хотя бы один итерируемый объект, иначе будет получен пустой кортеж. Получаемый в итоге ряд кортежей урезается по длине самого короткого аргумента *итерируемый_объект*. Если же указать единственный аргумент *итерируемый_объект*, то возвращается ряд одноэлементных кортежей. Эту функцию можно также использовать для разархивирования архивированных кортежей следующим образом: `X, Y = zip(*zip(T1, T2))`.

В версии Python 2.X эта функция возвращает *список*, а в версии Python 3.X возвращает *итерируемый объект*, формирующий значения по требованию и допускающий обход только один раз. Для того чтобы получить нужные результаты, в оболочку этой функции следует заключить вызов функции `list()`. Если указано несколько аргументов *итерируемый_объект* одинаковой длины, то в версии Python 2.X (но не 3.X) функция `zip()` действует аналогично функции `map()` с первым аргументом, равным `None`.

Встроенные функции в версии Python 2.X

В предыдущем разделе упоминались семантические отличия встроенных функций, доступных в *обеих* версиях Python, 3.X и 2.X. А в этом разделе отмечаются отличия в их содержании, наблюдаемые в обеих версиях Python.

Встроенные функции из версии Python 3.X, не поддерживаемые в версии Python 2.X

В версии Python 2.X недоступны следующие встроенные функции из версии Python 3.X.

`ascii()`

Действует аналогично функции `repr()` в версии Python 2.X.

`exec()`

У этой функции имеется равнозначная форма оператора со сходной семантикой в версии Python 2.X.

memoryview()

Эта функция все же доступна, начиная с выпуска 2.7, ради совместимости с версией Python 3.X.

print()

Присутствует в модуле `__builtin__` версии Python 2.X, но не может быть использована непосредственно без импорта модуля `__future__`, поскольку в синтаксисе версии Python 2.X для вывода на печать имеется соответствующая форма оператора и зарезервированное слово (см. выше раздел "Оператор `print`").

Встроенные функции из версии Python 2.X, не поддерживаемые в версии Python 3.X

В версии Python 2.X имеются также следующие встроенные функции, хотя некоторые из них доступны в других формах в версии Python 3.X:

apply(*функция*, *pargs* [, *kargs*])

Вызывает любой вызываемый объект *функция* (метод, функцию, класс и т.д.), передавая ему позиционные аргументы в кортеже *pargs*, а именованные аргументы — в словаре *kargs*. Возвращает результат вызова объекта *функция*.

Эта функция исключена из версии Python 3.X. Вместо нее рекомендуется использовать следующий синтаксис вызова с распаковкой аргументов: *функция*(*pargs*, ***kargs*). Такая форма со знаками звездочки предпочтительна и в версии Python 2.X, поскольку это более общая форма, симметричная членам, обозначенным звездочкой в определениях функций (см. выше раздел "Оператор выражения").

basestring()

Возвращает базовый (супер) класс для символьных строк (как обычных, так и в уникоде). Эта функция удобна для проверки типов с помощью функции `isinstance()`.

В версии Python 3 единственный тип `str` представляет весь текст как в 8-разрядном коде, так и в расширенном уникоде.

buffer(*объект* [, *смещение* [, *размер*]])

Возвращает новый объект буфера, удовлетворяющий требованиям заданного *объекта*. (Подробнее об этом см. в руководстве по библиотеке в версии Python 2.X.)

Эта функция исключена из версии Python 3.X. Новая встроенная функция `memoryview()` предоставляет аналогичные функциональные возможности в версии Python 3.X и также доступна в версии 2.7 ради прямой совместимости.

cmp(*X*, *Y*)

Возвращает отрицательное, нулевое или положительное целочисленное значение для отображения операций сравнения *X* < *Y*, *X* == *Y* или *X* > *Y* соответственно.

Эта функция исключена из версии Python 3.X, хотя и может быть сымитирована в виде следующего выражения: `(X > Y) - (X < Y)`. Но большинство общих примеров применения функции `cmp()` (функций сравнения при сортировке и методов `__cmp__()` из класса) исключены из версии Python 3.X.

coerce(*X*, *Y*)

Возвращает кортеж, содержащий два числовых аргумента, *X* и *Y*, преобразованных к общему типу данных. Эта функция исключена из версии Python 3.X. Она служила главным образом для классических классов в версии Python 2.X.

execfile(*имя_файла* [, *глобальные* [, *локальные*]])

Действует аналогично функции `eval()`, но выполняет весь код в файле, имя которого передается в виде символьной строки в аргументе *имя_файла* (вместо выражения). В отличие от операций импорта, эта функция не создает новый объект модуля для указанного файла. Она возвращает значение `None`. Пространства имен для кода из файла, представленного аргументом *имя_файла*, служат той же цели, что и для функции `eval()`.

В версии Python 3.X эта функция может быть сымитирована следующим образом: `exec(open(`*имя_файла*`).read())`.

file(*имя_файла* [, *режим*[, *размер_буфера*]])

Является псевдонимом встроенной функции open() и имеет имя класса подклассифицируемого встроенного типа файлов. Имя файла file исключено из версии Python 3.X, поэтому для доступа к файлам и модулю io из стандартной библиотеки с целью их специальной настройки следует использовать функцию open(). В версии 3.X модуль io применяется в функции open(), но такая возможность имеется и в версии 2.X (с выпуска 2.6).

input([*приглашение*]) (исходная форма в версии 2.X)

Выводит сначала *приглашение*, если таковое задано, а затем читает введенную строку из стандартного потока ввода sys.stdin, вычисляет ее как код Python и возвращает результат. В версии 2.X эта функция действует аналогично вызову функции eval(raw_input(*приглашение*)). *Совет:* пользоваться этой функцией для вычисления символьных строк ненадежного кода не рекомендуется, поскольку они могут содержать зловредный исполняемый код.

В версии Python 3.X функция raw_input() была переименована в input(), поэтому исходная функция input() в версии Python 2.X больше не доступна, хотя и может быть сымитирована следующим образом: eval(input(*приглашение*)).

intern(*символьная_строка*)

Вводит *символьную_строку* в таблицу так называемых *интернированных* строк и возвращает интернированную строку. Интернированные строки "вечны" и служат для целей оптимизации производительности. Их можно сравнивать с помощью оператора is быстрого определения идентичности, а не оператора сравнения ==.

Эта функция была перенесена в вызов функции sys.intern() в версии Python 3.X. Для этой цели следует импортировать модуль sys (подробнее об этом см. ниже, в разделе "Модуль sys").

long(*X* [, *основание*])

Преобразует число или символьную строку *X* в длинное целое число. Аргумент *основание* может быть передан только в том случае, если аргумент *X* является символьной строкой. Если же аргумент *основание* равен нулю, то основание определяется из содержимого символьной строки. В противном случае значение этого аргумента используется в качестве основания для преобразования. Обозначает также имя класса подклассифицируемого типа.

В версии Python 3.X целочисленный тип `int` поддерживает операции с длинными целыми числами произвольной точности, а следовательно, к его категории относится тип `long` из версии Python 2.X. Поэтому в версии Python 3.X следует использовать функцию `int()`.

raw_input([*приглашение*])

Этой функции в версии Python 3.X соответствует описанная ранее функция `input()`. Она выводит *приглашение*, читает и возвращает очередную введенную строку, но не вычисляет ее. В версии Python 3.X вместо нее следует использовать функцию `input()`.

reduce(*функция*, *итерируемый_объект* [, *начальное_значение*])

Применяет задаваемую *функцию* с двумя аргументами по очереди к элементам указанного *итерируемого_объекта*, чтобы свести коллекцию к единственному значению. Если задан аргумент *начальное_значение*, он добавляется в начале *итерируемого_объекта*.

Эта встроенная функция по-прежнему доступна в версии Python 3.X в виде вызова `functools.reduce()`. Для этого следует дополнительно импортировать модуль `functools`.

reload(*модуль*)

Перезагружает, повторно осуществляет синтаксический анализ и еще раз выполняет импортированный ранее *модуль* в его текущем пространстве имен. В результате

повторного выполнения происходит непосредственная замена прежних значений атрибутов модуля. Аргумент *модуль* должен ссылаться на объект существующего модуля, а не обозначать новое имя или символьную строку. Этой функцией удобно пользоваться в диалоговом режиме, если модуль требуется перезагрузить после его исправления, не перезапуская интерпретатор Python. Она возвращает объект *модуль*. См. также таблицу `sys.modules`, где сохраняются импортируемые модули (их можно удалить, чтобы принудительно повторить импорт). В версии Python 3.X эта встроенная функция по-прежнему доступна в виде вызова `imp.reload()`. Для этого следует импортировать модуль `imp`.

`unichr(`*`I`*`)`

Возвращает строку с одним символом в уникоде, кодовая точка которого обозначается аргументом *I*. Например, в результате вызова функции `unichr(97)` возвращается символьная строка `u'a'`. Эта функция является обратной функции `ord()` для обработки символьных строк в уникоде, а также версией функции `chr()` для обработки символов в уникоде. В качестве аргумента *I* может быть указано числовое значение в пределах от **0** до **65535** включительно, а иначе возникает исключение типа `ValueError`.

В версии Python 3.X обычные символьные строки представлены символами в уникоде, поэтому вместо данной функции следует вызывать функцию `chr()`. Например, в результате вызова функции `ord('\xe4')` возвращается числовое значение **228**, тогда как в результате обоих вызовов функции `chr(228)` и `chr(0xe4)` — символьная строка `'ä'`.

`unicode([`*`объект`*`[, `*`кодировка`*`[, `*`ошибки`*`]]])`

Действует аналогично функции `str()` в версии 3.X (подробнее см. выше описание функции `str()`). Если данной функции передается только *один* аргумент, то она возвращает печатаемое строковое представление *объекта* на высоком уровне восприятия, но в виде символьной строки

в уникоде (для версии 2.X), а не объекта типа `str`. Если же данной функции передается больше одного аргумента, то она выполняет декодирование в уникоде строкового представления *объекта*, используя кодер-декодер, задаваемый аргументом *кодировка*, а также выполняя обработку ошибок в режиме, задаваемом аргументом *ошибки*. По умолчанию обработка ошибок происходит в строгом режиме, когда исключение типа `ValueError` генерируется при появлении любых ошибок кодирования.

См. также описание файлов, поддерживающих в модуле `codecs` разные кодировки, в руководстве по библиотеке Python. В версии 2.X объекты могут предоставлять метод `__unicode__()`, возвращающий их строковое представление в уникоде для функции `unicode(X)`.

В версии Python 3.X отсутствует отдельный тип для данных в уникоде. Вместо этого тип `str` представляет весь текст (как в 8-разрядном коде, так и в расширенном уникоде), а тип `bytes` — байты 8-разрядных двоичных данных. Поэтому для представления обычного текста следует использовать символьные строки типа `str` в уникоде; функции `bytes.decode()` или `str()` — для декодирования исходных байтов в символы уникода по заданной кодировке; а обычные файловые объекты, получаемый при вызове функции `open()`, — для обработки текстовых файлов в уникоде.

`xrange([`*`начало`*`,] `*`конец`*`[, `*`шаг`*`])`

Действует аналогично функции `range()`, но не сохраняет сразу весь ряд целочисленных значений, а формирует их по очереди. Этой функцией удобно пользоваться в циклах `for`, когда имеется большой ряд целочисленных значений, а оперативной памяти недостаточно. Она оптимизирует доступное пространство в памяти, но, как правило, не дает никаких преимуществ в быстродействии. В версии Python 3.X исходная функция `range()` изменена, чтобы возвращать итерируемый объект, вместо того чтобы формировать ряд целочисленных значений в оперативной

памяти. Следовательно, к ее категории относится и функция `xrange()` из версии Python 2.X, которая исключена из версии 3.X.

Кроме того, функция `open()` была настолько радикально изменена в версии Python 3.X, что здесь следует отдельно упомянуть ее вариант в версии Python 2.X. Функция `codecs.open()` в версии Python 2.X обладает многими возможностями функции `open()` в версии Python 3.X, включая поддержку кодирования и декодирования текстовых данных в уникоде при их передаче в файлы и обратно.

`open(`*`имя_файла`*`[, `*`режим`*`, [`*`размер_буфера`*`]])`

Возвращает новый файловый объект, связанный с внешним файлом, обозначаемым аргументом *`имя_файла`* (в виде символьной строки), а при неудачном исходе открытия файла генерирует исключение типа `IOError`. Имя файла отображается на текущий рабочий каталог, если только оно не включает в себя префикс пути к каталогу. Два первых аргумента этой функции, как правило, аналогичны аргументам функции `fopen()` в C, а управление файлом осуществляется системой `stdio`. Данные из файла функция `open()` обычно представляет в виде обычной символьной строки типа `str`, состоящей из байтов, извлеченных из файла. А функция `codecs.open()` интерпретирует содержимое файла в виде текста, представленного в уникоде объектами типа `unicode`.

Если аргумент *`режим`* опускается, то по умолчанию он принимает значение `'r'`. Но этот аргумент может также принимать значение `'r'` для ввода данных из файла, значение `'w'` для вывода данных в файл (с предварительным урезанием файла), значение `'a'` для присоединения данных к файлу, а также значения `'rb'`, `'wb'` или `'ab'` для обработки двоичных файлов с подавлением взаимного преобразования символов конца строки в последовательность символов `\n`. В большинстве систем указываемые режимы могут также дополняться знаком **+**, чтобы открывать файлы в режиме ввода-вывода обновлений (например, режим `'r+'`

служит для чтения и записи, а режим `'w+'` — для чтения и записи, но с инициализацией пустого файла).

По умолчанию аргумент *размер_буфера* принимает значение, зависящее от конкретной реализации. Но этот аргумент может также принимать нулевое значение, если данные не буферизуются; значение **1** для линейной буферизации; отрицательное значение для стандартной буферизации на уровне системы или же конкретный размер буфера. Передача буферизованных данных не может быть выполнена немедленно, поэтому для принудительного опорожнения буфера следует вызвать метод `flush()` файлового объекта. См. также описание модуля `io` в стандартной библиотеке Python, а именно: альтернативы встроенному типу `file` в версии 2.X и обычного файлового интерфейса для функции `open()` в версии 3.X.

Встроенные исключения

В этом разделе описываются исключения, которые предопределены в языке Python и могут генерироваться его интерпретатором или прикладным кодом во время выполнения программы. Основное внимание в этом разделе уделяется состоянию встроенных исключений в версии Python 3.3, где внедрены новые классы для обработки системных ошибок. К их категории теперь относятся прежние обобщенные классы, предоставляющие сведения о состоянии, которые дополняются подробностями, общими для большинства версий Python. О характерных отличиях встроенных исключений в версиях Python 3.X и 2.X речь пойдет в конце этого раздела.

Начиная с версии Python 1.5 все встроенные исключения являются *объектами классов*, тогда как до версии 1.5 они были представлены символьными строками. Встроенные исключения предоставляются в пространстве имен встроенной области действия (см. ранее раздел "Правила обозначения пространств имен и областей действия"), и со многими встроенными исключениями связаны сведения о состоянии, предоставляющие подробности об исключении. А определяемые пользователем исключения, как

правило, являются производными от встроенных исключений (см. выше раздел "Оператор `raise`").

Суперклассы категорий исключений

Приведенные ниже исключения служат только в качестве суперклассов для других исключений.

`BaseException`

Это корневой суперкласс для всех встроенных исключений. Он не предназначен для непосредственного наследования определяемых пользователем классов, поэтому для этой цели следует использовать класс `Exception`. Так, если функция `str()` вызывается для экземпляра данного класса, то возвращается представление аргументов конструктора, передаваемых при создании экземпляра, а если такие аргументы отсутствуют, — пустая символьная строка. Аргументы конструктора экземпляра сохраняются и становятся доступными в виде кортежа в атрибуте `args` этого экземпляра. Такой протокол наследуется подклассами.

`Exception`

Это корневой суперкласс для всех встроенных исключений и тех, что не приводят к выходу из системы. Это подкласс, непосредственно производный от класса `BaseExceptio`. Все определяемые пользователем исключения должны быть производными (наследовать) от этого класса. Такое наследование требуется для определяемых пользователем исключений в версии Python 3.X. А в версиях Python 2.7 и 2.6 наследование требуется для классов нового стиля, но в то же время оно допускает создание автономных классов исключений.

Операторы `try`, перехватывающие данное исключение, будут перехватывать все, кроме событий выхода их системы, поскольку этот класс служит суперклассом для классов всех исключений, кроме `SystemExit`, `KeyboardInterrupt` и `GeneratorExit`, так как эти три класса исключений являются производными непосредственно от класса `BaseException`.

ArithmeticError

Это категория исключений, связанных с ошибками арифметических операций и суперкласс для классов исключений `ZeroDivisionError` и `FloatingPointError`, а также подкласса, производного от класса `Exception`.

BufferError

Это исключение возникает, когда не может быть выполнена операция, связанная с буфером данных. Его класс является подклассом, производным от класса `Exception`.

LookupError

Это исключение возникает в связи с ошибками индексирования последовательностей и отображений, а также некоторыми ошибками поиска кодировки в уникоде. Его класс служит суперклассом для классов исключений `IndexError` и `KeyError`. В то же время класс этого исключения является подклассом, производным от класса `Exception`.

OSError (начиная с версии Python 3.3)

Это исключение возникает, когда системная функция выдает системную ошибку, включая сбои в операциях файлового ввода-вывода. В версии Python 3.3 класс этого исключения является корневым для нового ряда исключений, описывающих сбои в системе и перечисляемых далее, в разделе "Конкретные исключения типа `OSError`". К их категории относятся также обобщенные исключения со сведениями о состоянии, описываемые далее, в разделе "Встроенные исключения в версии Python 3.2".

В версии 3.3 класс `OSError` является подклассом, производным от класса `Exception`, и включает в себя следующие общие информационные атрибуты, предоставляющие подробности о системных ошибках: `errno` (числовой код); `strerror` (строка сообщения); `winerror` (ошибка в Windows); а также `filename` (для исключений, включающих в себя пути к файлам). В версии 3.3 этот класс включает в себя прежние типы исключений `EnvironmentError`, `IOError`, `WindowsError`, `VMSError`, `socket.error`,

select.error, mmap.error и является синонимом класса os.error. Подробнее об атрибутах системных исключений см. ниже, в разделе "Системный модуль os".

Конкретные исключения

Ниже описываются классы более конкретных, фактически возникающих исключений. Кроме того, исключения типа NameError, RuntimeError, SyntaxError, ValueError и Warning являются одновременно конкретными исключениями и суперклассами категорий других встроенных исключений.

AssertionError

Возникает, когда проверка в операторе assert дает ложный результат, т.е. не проходит.

AttributeError

Возникает при неудачном исходе обращения к атрибуту или его присваивания.

EOFError

Возникает, когда в функции input() (или raw_input() в версии Python 2.X) сразу же обнаруживается конец файла. По достижении конца файла методы чтения файловых объектов возвращают пустой объект, вместо того чтобы генерировать это исключение.

FloatingPointError

Возникает при неудачном исходе операции над числами с плавающей точкой.

GeneratorExit

Возникает, когда в генераторе вызывается метод close(). Класс этого исключения наследует непосредственно от класса BaseException, а не Exception, поскольку данное исключение не является ошибкой.

ImportError

Возникает, когда в операторе import или from не удается найти модуль или атрибут. В версии Python 3.3 экземпляры

классов включают в себя атрибуты `name` и `path`, обозначающие модуль, в котором возникает ошибка, и передаваемые в качестве именованных аргументов конструктору.

`IndentationError`

Возникает, когда в исходном коде обнаруживается неверный отступ. Класс этого исключения является производным от класса `SyntaxError`.

`IndexError`

Возникает при выходе индекса за пределы последовательности (в операциях извлечения или присваивания). Индексы нарезок настраиваются таким образом, чтобы попасть в допустимые пределы. Если индекс не является целочисленным значением, возникает исключение типа `TypeError`.

`KeyError`

Возникает по ссылкам на несуществующие ключи отображения (в операциях извлечения). Если выполняется присваивание несуществующему ключу, этот ключ создается.

`KeyboardInterrupt`

Возникает после нажатия пользователем клавиши прерывания (как правило, комбинации клавиш <Ctrl+C> или <Delete>). Во время исключения регулярно выполняется проверка на прерывание. Класс этого исключения наследует непосредственно от класса `BaseException`, чтобы избежать его случайного перехвата в прикладном коде, перехватывающем исключение типа `Exception`, а следовательно, предотвратить выход из интерпретатора Python.

`MemoryError`

Возникает при исчерпании восстанавливаемой оперативной памяти. В итоге отображаются результаты трассировки стека, если причиной тому стала неуправляемая программа.

`NameError`

Возникает, когда не удается обнаружить неуточненное локальное или глобальное имя.

NotImplementedError

Возникает, когда не удается определить предполагаемые протоколы. Это исключение может быть сгенерировано в методах абстрактных классов, когда в них требуется переопределить метод. Класс этого исключения является производным от класса `RuntimeError`. (Класс этого исключения не следует путать со специальным встроенным классом `NotImplemented`, объект которого возвращается некоторыми методами перегрузки операторов, когда типы операндов не поддерживаются; см. ранее раздел "Методы перегрузки операторов").

OverflowError

Возникает, когда арифметические операции дают слишком крупные результаты. Ничего подобного не может произойти в операциях с целочисленными значениями, поскольку в них поддерживается произвольная точность. В силу ограничений, присущих базовому языку C, проверка на переполнение в большинстве операций над числовыми значениями с плавающей точкой также не производится.

ReferenceError

Возникает вместе со *слабыми ссылками* — средствами для поддержания ссылок на объекты, которые не препятствуют их освобождению из оперативной памяти (например, из кешей). См. также описание модуля `weakref` из стандартной библиотеки Python.

RuntimeError

Редко используемое универсальное исключение.

StopIteration

Возникает по окончании продвижения значений в объектах итераторов. Оно генерируется встроенной функцией `next(`*I*`)` и методом *I*`.__next__()` (именуемым *I*`.next()` в версии Python 2.X).

В версии Python 3.3 экземпляры классов включают в себя атрибут `value`, который отражает позиционный аргумент явного конструктора или же в нем автоматически

устанавливается возвращаемое значение, задаваемое в операторе `return` функции-генератора, которым завершается итерация. По умолчанию значение этого атрибута равно `None` и также доступно в обычном для исключения кортеже `args`, но не используется при автоматической итерации. До версии 3.3 функции-генераторы не должны были возвращать никаких значений, иначе возникали бы синтаксические ошибки, поэтому применение данного исключения не совместимо с прежними версиями 2.X и 3.X. См. выше раздел "Оператор `yield`".

`SyntaxError`

Возникает, когда появляется синтаксическая ошибка. Нечто подобное может произойти во время операций импорта, вызова функций `eval()` и `exec()`, а также при чтении кода из файла сценария высокого уровня или стандартного потока ввода. У экземпляров класса этого исключения имеются атрибуты `filename`, `lineno`, `offset` и `text` для доступа к подробностям данного исключения. А в результате вызова функции `str()` для экземпляра данного исключения возвращается только самое элементарное сообщение.

`SystemError`

Возникает, когда появляются внутренние ошибки интерпретатора Python, хотя они и не настолько серьезные, чтобы прекратить его работу. О таких ошибках следует непременно сообщать.

`SystemExit`

Возникает при вызове функции `sys.exit(`*N*`)`. Если не обработать это исключение, интерпретатор Python прекратит свою работу без вывода результатов обратной трассировки стека. Если этой функции передается целочисленное значение аргумента *N*, оно обозначает состояние выхода из программы в систему, передаваемое функции выхода, написанной на C. Если же аргумент *N* принимает значение `None` или вообще опускается, то состояние выхода равно нулю (что означает успешное завершение). А если этот аргумент принимает значение другого типа, то выводится значение

объекта и состояние выхода равно **1** (неудачное завершение). Класс данного исключения является производным от класса `BaseException`, чтобы избежать его случайного перехвата в прикладном коде, перехватывающем исключение типа `Exception`, а следовательно, предотвратить выход из интерпретатора Python. Дополнительно см. далее раздел "Модуль `sys`".

Это исключение генерируется функцией `sys.exit()` таким образом, чтобы выполнить обработчики, возвращающие ресурсы (операторы в блоке `finally` оператора `try`), и чтобы отладчик мог выполнить сценарий, не теряя управление. Функция `os._exit()` осуществляет выход, как только в этом возникает потребность (например, в порожденном процессе после вызова функции `fork()`). О функции выхода см. также в описании модуля `atexit` из стандартной библиотеки Python.

`TabError`

Генерируется, когда в исходном коде обнаруживается неверное сочетание пробелов и символов табуляции. Класс этого исключения является производным от класса `IndentationError`.

`TypeError`

Генерируется, когда операция или функция применяется к объекту неподходящего типа.

`UnboundLocalError`

Генерируется по ссылкам на имена локальных переменных, которым еще не присвоено значение. Класс этого исключения является производным от класса `NameError`.

`UnicodeError`

Генерируется при появлении ошибок кодирования и декодирования в уникоде. Его класс относится к категории суперклассов и в то же время является подклассом, производным от класса `ValueError`. *Совет:* некоторые средства кодирования и декодирования в уникоде могут также генерировать исключение типа `LookupError`.

UnicodeEncodeError
UnicodeDecodeError
UnicodeTranslateError

Генерируются при обработке ошибок кодирования и декодирования в уникоде. Их классы являются подклассами, производными от класса `UnicodeError`.

ValueError

Генерируется, когда встроенная функция или операция получает аргумент, имеющий правильный тип, но неверное значение, и эта ситуация не описывается более конкретным исключением вроде `IndexError`.

ZeroDivisionError

Генерируется в операциях деления или получения модуля с нулевым значением в качестве правого операнда.

Конкретные исключения типа `OSError`

Перечисленные ниже подклассы исключений являются производными от класса `OSError`, начиная с версии Python 3.3. Они обозначают системные ошибки и соответствуют кодам системных ошибок, которые были доступны в классе `EnvironmentError` в прежних версиях Python (см. далее раздел "Встроенные исключения в версии Python 3.2"). Описание информационных атрибутов, общих для всех перечисленных ниже подклассов, см. выше, в разделе "Суперклассы категорий исключений".

BlockingIOError

Генерируется, когда операция блокируется для объекта, установленного для выполнения неблокирующей операции. Класс этого исключения имеет дополнительный атрибут `characters_written`, обозначающий количество символов, выведенных в поток до блокировки.

ChildProcessError

Генерируется при неудачном исходе операции над порожденным процессом.

ConnectionError

Класс этого исключения является суперклассом для следующих исключений, связанных с установлением соединения: `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError` и `ConnectionResetError`.

BrokenPipeError

Генерируется при попытке вывести данные в конвейер, в то время как соединение на другом конце разрывается, или при попытке вывести данные в сокет, закрытый для вывода.

ConnectionAbortedError

Генерируется, когда одноранговый узел прерывает попытку установить соединение.

ConnectionRefusedError

Генерируется, когда одноранговый узел отказывается от попытки установить соединение.

ConnectionResetError

Генерируется, когда одноранговый узел сбрасывает соединение.

FileExistsError

Генерируется, когда создается файл или каталог, который уже существует.

FileNotFoundError

Генерируется, когда запрашиваемый файл или каталог не существует.

InterruptedError

Генерируется, когда системный вызов прерывается входящим сигналом.

IsADirectoryError

Генерируется, когда такая операция с файлами, как `os.remove()`, запрашивается в каталоге.

NotADirectoryError

Генерируется, когда такая операция с каталогами, как `os.listdir()`, запрашивается не в каталоге.

PermissionError

Генерируется при попытке выполнить операции без надлежащих прав доступа (например, полномочий на пользование файловой системой).

ProcessLookupError

Генерируется, когда процесс не существует.

TimeoutError

Генерируется, когда истекает время ожидания системной функции на уровне системы.

Исключения категории предупреждений

Перечисленные ниже исключения относятся к категории предупреждений.

Warning

Класс этого исключения является суперклассом для приведенных ниже предупреждений. В то же время он является подклассом, производным от класса `Exception`.

UserWarning

Предупреждения, генерируемые в прикладном коде.

DeprecationWarning

Предупреждения о средствах, не рекомендованных к применению.

PendingDeprecationWarning

Предупреждения о средствах, применение которых будет не рекомендовано в будущем.

SyntaxWarning

Предупреждения о сомнительном синтаксисе.

RuntimeWarning

Предупреждения о сомнительном поведении во время выполнения.

FutureWarning

Предупреждения о конструкциях, которые будут семантически изменены в будущем.

ImportWarning

Предупреждения о возможных ошибках в операциях импорта модулей.

UnicodeWarning

Предупреждения, связанные с кодированием и декодированием в уникоде.

BytesWarning

Предупреждения, связанные с объектами типа `bytes` и объектами буферов (представлений памяти).

ResourceWarning

Этот суперкласс был внедрен в версии Python 3.2 для предупреждений, связанных с использованием ресурсов.

Каркас предупреждений

Предупреждения выдаются, когда предстоящие изменения в языке программирования могут нарушить существующий код в будущих выпусках Python и других контекстах. Предупреждения могут быть настроены на вывод сообщений, генерирование исключений или полное их игнорирование. Для выдачи предупреждений можно воспользоваться каркасом предупреждений, вызывая функцию `warnings.warn()` следующим образом:

```
warnings.warn("usage obsolete", DeprecationWarning)
```

Для отмены некоторых предупреждений могут быть введены специальные фильтры. А для подавления предупреждений с переменной степенью обобщенности можно применить к сообщению или имени модуля шаблон регулярного выражения. Например,

предупреждение о не рекомендованном к употреблению модуле regex можно подавить, сделав следующий вызов:

```
import warnings
warnings.filterwarnings(action = 'ignore',
                        message='.*regex module*',
                        category=DeprecationWarning,
                        module = '__main__')
```

В этом фрагменте кода вводится фильтр, оказывающий влияние только на предупреждения из класса DeprecationWarning, инициируемые в модуле __main__; применяется регулярное выражение для совпадения только с тем сообщением, которое именует не рекомендованный к употреблению модуль regex; а также задаются условия для игнорирования подобного предупреждения. Кроме того, предупреждения могут выводиться только один раз, каждый раз, когда выполняется ошибочный код, или же превращаться в исключения, приводящие к прекращению работы программы, если только эти исключения не перехвачены. Подробнее об этом см. описание модуля warnings в руководстве по Python, начиная с версии 2.1. См. также описание аргумента **-W** выше, в разделе "Параметры командной строки в Python".

Встроенные исключения в версии Python 3.2

Перечисленные ниже исключения доступны в версии Python 3.2 и до нее. А в версии Python 3.3 все они были объединены в один класс исключения OSError. Тем не менее они сохранены в версии 3.3 ради обратной совместимости, хотя и могут быть исключены из последующий выпусков Python.

EnvironmentError

Относится к категории исключений, происходящих вне среды Python. Класс этого исключения служит суперклассом для классов исключений IOError и OSError, но в то же время он является подклассом, производным от класса Exception. Экземпляр генерируемого исключения содержит информационные атрибуты errno и strerror, а возможно, и атрибут filename для исключений, включающих в себя пути к файлам. Эти атрибуты присутствуют также в

аргументах args и предоставляют коды системных ошибок и подробные сведения для сообщений.

IOError

Генерируется при неудачном исходе операции ввода-вывода или обращения к файлу. Класс этого исключения является производным от класса EnvironmentError с упоминавшимися ранее сведениями о состоянии.

OSError (в версии Python 3.2)

Генерируется при появлении ошибок в модуле os (его исключении os.error). Класс этого исключения является производным от класса EnvironmentError с упоминавшимися ранее сведениями о состоянии.

VMSError

Генерируется при появлении ошибок, характерных для системы виртуальной памяти (VMS). Класс этого исключения является подклассом, производным от класса OSError.

WindowsError

Генерируется при появлении ошибок, характерных для Windows. Класс этого исключения является подклассом, производным от класса OSError.

Встроенные исключения в версии Python 2.X

Ряд доступных исключений, а также иерархическая структура классов исключений несколько отличаются в версии Python 2.X от всего, что было описано ранее для версии 3.X. Эти отличия в версии Python 2.X заключаются, в частности, в следующем.

- Класс Exception, а не класс BaseException, отсутствующий в версии Python 2.X, является корневым и находится на самой вершине иерархии классов исключений.
- Класс StandardError является дополнительным подклассом, производным от класса Exception, а также корневым классом для всех встроенных исключений, кроме SystemExit.

Подробнее об исключениях в версии Python 2.X см. в документации на соответствующую библиотеку конкретной версии.

Встроенные атрибуты

Некоторые объекты экспортируют специальные атрибуты, предопределенные в Python. Ниже приведен неполный перечень встроенных атрибутов, поскольку многие типы данных имеют свои особые атрибуты. Подробнее об атрибутах отдельных типов данных см. в руководстве по библиотеке Python.[5]

`X.__dict__`

Словарь, используемый для сохранения изменяемых (записываемых) атрибутов объекта *X*.

`I.__class__`

Объект класса, из которого сформирован экземпляр *I*. Начиная с версии 2.2 этот атрибут применяется также к типам данных. Атрибут `__class__` имеется у большинства объектов. Например: `[].__class__ == list == type([])`.

`C.__bases__`

Кортеж базовых классов, перечисляемых в заголовке класса *C*.

`C.__mro__`

Путь, вычисляемый по правилу MRO при обходе иерархического дерева класса *C* нового стиля (см. выше, подраздел "Классы нового стиля: правило MRO").

---

[5] Начиная с версии Python 2.1 произвольные определяемые пользователем атрибуты можно присоединять к *функциональным* объектам, просто присваивая им значения (см. ранее подраздел "Атрибуты функций и значения аргументов по умолчанию"). В версии Python 2.X поддерживаются также специальные атрибуты *I.*`__methods__` и *I.*`__members__`, перечисляющие имена методов и членов данных для экземпляров некоторых встроенных типов. Эти атрибуты исключены из версии Python 3.X, а вместо них следует пользоваться встроенной функцией `dir()`.

X.__name__

Имя объекта *X* в виде символьной строки. Это имя указывается в заголовке класса, используется в операциях импорта модулей или обозначает модуль как "__main__" в самом начале программы (например, главный файл для запуска программы).

Стандартные библиотечные модули

Стандартные библиотечные модули доступны всегда, тем не менее их нужно импортировать, чтобы применять в прикладных модулях. Для доступа к ним служит один из следующих форматов.

- `import` *модуль*, а также *модуль*`.`*имя* для извлечения имени атрибутов.
- `import` *модуль* `import` *имя*, а также *имя* для применения неуточтенных имен модулей.
- `import` *модуль* `import *`, а также *имя* для применения неуточтенных имен модулей.

Например, для того чтобы воспользоваться именем `argv` из модуля `sys`, следует указать оператор `import sys` и имя `sys.argv` или же оператор `from sys import argv` и имя `argv`. Первая полная форма *модуль*`.`*имя* применяется в заголовках со списками содержимого только для предоставления контекста в многостраничных листингах, тогда как в описаниях чаще всего применяется более краткая форма *имя*.

Имеются буквально сотни стандартных библиотечных модулей, и все они подвержены изменениям вместе с самим языком Python. Поэтому в последующих разделах описывается далеко *не* полный перечень наиболее общеупотребительных модулей. Более подробное описание стандартных библиотечных модулей см. в руководстве по библиотеке Python.

Все упоминаемые далее модули описываются с учетом следующего.

- Перечисляемые имена экспортируемых компонентов, после которых следуют круглые скобки, являются *функциями*, которые непременно должны быть вызваны. А все остальные компоненты являются простыми атрибутами (т.е. именами переменных, извлекаемых, но не вызываемых из модулей).
- Содержимое модулей описывает состояние модулей в версии *Python 3.X*, но это описание, как правило, распространяется на обе версии, 2.X и 3.X, если не указано иное. Подробнее об отличиях в обеих версиях см. в руководствах по Python.

Модуль sys

Модуль `sys` содержит средства, *связанные* с интерпретатором Python. Это элементы, связанные с интерпретатором Python и его процессом в обеих версиях Python, 2.X и 3.X. В нем также предоставляется доступ к некоторым компонентам окружения, в том числе к командной строке, стандартным потокам ввода-вывода и т.д. О других средствах, связанных с процессами, см. далее, в разделе "Системный модуль `os`".

sys.argv

Представляет следующий список символьных строк с аргументами командной строки: [*имя_сценария, аргументы*...], аналогичный массиву аргументов `argv` в C. В частности, элемент списка `argv[0]` содержит имя файла сценария (возможно, с указанием полного пути); символьную строку `'-c'` в качестве параметра **`-c`** командной строки; путь к модулю в качестве параметра **`-m`** командной строки; символьную строку `'-'` в качестве параметра `-` командной строки или же пустую строку, если имя сценария или параметр командной строки не передается. Дополнительно см. выше раздел "Указание программ в командной строке".

sys.byteorder

Обозначает порядок следования байтов, характерный для конкретной платформы (например, `'big'` — обратный

порядок следования байтов, а `'little'` — прямой порядок следования байтов).

sys.builtin_module_names

Представляет кортеж имен модулей C в виде символьных строк, скомпилированных в данном интерпретаторе Python.

sys.copyright

Представляет символьную строку, содержащую авторские права на интерпретатор Python.

sys.displayhook(*значение*)

Вызывается интерпретатором Python для отображения результирующих значений в диалоговом режиме работы. Для специальной настройки вывода функцию `sys.displayhook()` следует присвоить функции с одним аргументом.

sys.dont_write_bytecode

Если эта переменная принимает истинное значение, то интерпретатор Python вообще не будет пытаться выводить данные в файлы с расширением **.pyc** или **.pyo** при импорте исходных модулей (см. также описание параметра командной строки **-B** выше, в разделе "Запуск программ на Python из командной строки").

sys.excepthook(*тип*, *значение*, *объект обратной трассировки стека*)

Вызывается интерпретатором Python для вывода подробностей неперехваченного исключения в стандартный поток `sys.stderr`. Для специальной настройки вывода исключений функции `sys.excepthook()` следует присвоить функцию с тремя аргументами.

sys.exc_info()

Возвращает следующий кортеж из трех значений, описывающих исключение, обрабатываемое в настоящий момент: (*тип*, *значение*, *объект обратной трассировки стека*), где *тип* — класс исключения; *значение* —

экземпляр класса сгенерированного исключения; *объект обратной трассировки стека* — объект, предоставляющий доступ к стеку вызовов во время выполнения в том состоянии, в каком он находился на тот момент, когда возникло исключение. Получаемый результат характерен для текущего потока исполнения. К этой категории относятся атрибуты `exc_type`, `exc_value` и `exc_traceback`, начиная с версии Python 1.5 (эти три атрибута присутствуют и в прежних выпусках Python 2.X, но полностью исключены из версии Python 3.X). Подробнее об обработке объектов обратной трассировки стека см. описание модуля `traceback` в руководстве по библиотеке Python, а об обработке исключений — ранее в разделе "Оператор `try`".

`sys.exec_prefix`

Этой переменной следует присвоить символьную строку, предоставляющую префикс того каталога, где установлены файлы Python, не зависящие от платформы. По умолчанию в этой переменной хранится префикс каталога ***/usr/local*** или аргумент времени компоновки. Используется для обнаружения местоположения общих библиотечных модулей (в каталоге ***<exec_prefix>/lib/python<версия>/lib-dynload***) и файлов конфигурации.

`sys.executable`

Представляет символьную строку, предоставляющую полный путь к файлу с интерпретатором Python, выполняющим вызывающий код.

`sys.exit([N])`

Осуществляет выход из процесса Python с состоянием *N* (по умолчанию равным нулю), генерируя встроенное исключение типа `SystemExit`, которое может быть перехвачено в операторе `try` и, если требуется, проигнорировано. Подробнее об исключении `SystemExit` см. выше, в разделе "Встроенные исключения", а о применении функции `os._exit()` и связанных с ней средств немедленного выхода без обработки исключений, что удобно в порожденных процессах после вызова функции `os.fork()`, — ниже,

в разделе "Системный модуль os". Что же касается определения общей функции выхода, то см. описание модуля atexit из стандартной библиотеки Python.

sys.flags

Предоставляет значения параметров командной строки в Python, по одному атрибуту на каждый такой параметр (подробнее об этом см. в руководстве по Python).

sys.float_info

Предоставляет подробности о реализации в Python операций над числами с плавающей точкой посредством атрибутов (подробнее об этом см. в руководстве по Python).

sys.getcheckinterval()

Возвращает "интервал проверки", выполняемой интерпретатором, в версии Python 3.1 и до нее. В версии Python 3.2 заменена описываемой далее функцией setcheckinterval().

sys.getdefaultencoding()

Возвращает наименование текущей стандартной кодировки символьных строк, применяемой в реализации уникода.

sys.getfilesystemencoding()

Возвращает наименование кодировки, применяемой для преобразования имен файлов из уникода в их системное представление, или же значение None, если используется кодировка, выбираемая в системе по умолчанию.

sys._getframe([*глубина*])

Возвращает объект фрейма из стека вызовов в Python (подробнее об этом см. в руководстве по библиотеке Python).

sys.getrefcount(*объект*)

Возвращает текущее значение подсчета ссылок на *объект* (+1 для самого аргумента вызываемой функции).

sys.getrecursionlimit()

Возвращает максимальный предел глубины стека вызовов в Python (дополнительно см. далее описание функции `setrecursionlimit()`).

sys.getsizeof(*объект* [, *по_умолчанию*])

Возвращает размер задаваемого *объекта* в байтах. Это может быть *объект* любого типа. Все встроенные объекты возвращают правильные результаты, тогда как результаты сторонних расширений зависят от конкретной реализации. Аргумент *по_умолчанию* предоставляет значение, которое возвращается, если объект относится к типу, не реализующему интерфейс для извлечения размера объекта.

sys.getswitchinterval()

Возвращает текущую установку интервала для переключения потоков исполнения в интерпретаторе Python, начиная с версии Python 3.2. А в версии Python 3.1 и до нее следует использовать функцию `getcheckinterval()`. Дополнительно см. далее описание функции `setswitchinterval()`.

sys.getwindowsversion()

Возвращает объект, описывающий текущую выполняющуюся версию Windows (подробнее об этом см. в руководстве по Python).

sys.hexversion

Предоставляет номер версии Python в виде одного целочисленного значения, которое, возможно, лучше просматривать с помощью встроенной функции `hex()`. Это целочисленное значение увеличивается с каждым новым выпуском Python.

sys.implementation

В версии Python 3.3 предоставляет объект, снабжающий сведениями о реализации выполняющегося в настоящий момент интерпретатора Python, включая наименование, версию и пр. Подробнее об этом см. в руководстве по Python.

sys.int_info

Предоставляет подробные сведения о реализации целочисленных операций в Python. Подробнее об этом см. в руководстве по Python.

sys.intern(*символьная_строка*)

Вводит *символьную_строку* в таблицу интернированных строк и возвращает интернированную строку, т.е. саму символьную строку или ее копию. Интернирование символьных строк обеспечивает незначительное улучшение производительности для словарного поиска. Если ключи в словаре и ключи поиска интернированы, то сравнение ключей (после хеширования) может быть произведено путем сравнения указателей, а не символьных строк. Как правило, имена, используемые в программах на Python, интернируются автоматически, а словари, применяемые для хранения атрибутов модулей, классов и экземпляров, имеют интернированные ключи.

sys.last_type, **sys.last_value**, **sys.last_traceback**

Предоставляют соответственно тип, значение и объекты обратной трассировки стека последнего перехваченного исключения (главным образом, для отладки после аварийного сбоя).

sys.maxsize

Предоставляет целое число, обозначающее максимальное значение, которое может принимать переменная типа `Py_ssize_t`. Как правило, это целое число **`2**31 - 1`** на 32-разрядной платформе или **`2**63 - 1`** на 64-разрядной.

sys.maxunicode

Предоставляет целочисленное значение, обозначающее наибольшую поддерживаемую кодовую точку для символа в уникоде. Начиная с версии Python 3.3 это всегда значение **`1114111`** (или **`0x10FFFF`** в шестнадцатеричной форме) благодаря внедренной в данной версии гибкой системе хранения символьных строк переменной длины. До версии 3.3 это значение зависело от параметра конфигурации,

определявшего, будут ли символы в уникоде сохраняться в кодировке UCS-2 или UCS-4, и поэтому оно могло быть **0xFFFF** или **0x10FFFF**.

sys.modules

Предоставляет словарь уже загруженных модулей, каждому из которых соответствует отдельная запись *имя*:*объект* в словаре. *Совет:* содержимое этого словаря может изменяться, отражая результаты последующих операций импорта. Например, в результате операции `del sys.modules['`*имя*`']` указанный модуль перезагружается при последующем импорте.

sys.path

Предоставляет список символьных строк, обозначающих путь для поиска импортируемых модулей. Инициализируется из переменной окружения PYTHONPATH командной оболочки любыми файлами путей с расширением **.pth** и настройками по умолчанию, зависящими от конкретной установки. *Совет:* этот путь является абсолютным, а его список может быть изменен, чтобы отражать последующие операции импорта. Например, в результате операции `sys.path.append('C:\\dir')` указанный каталог динамически вводится в путь для поиска модулей.

Первый элемент `path[0]` представляемого списка содержит сценарий, использовавшийся для вызова интерпретатора Python. Если каталог этого сценария недоступен (например, интерпретатор вызывался в диалоговом режиме или сценарий вводился из стандартного потока), то элемент списка `path[0]` содержит пустую строку, а следовательно, поиск модулей осуществляется сначала в рабочем каталоге. Каталог сценария вводится перед любыми записями из переменной окружения PYTHONPATH. Дополнительно см. выше раздел "Оператор `import`".

sys.platform

Предоставляет символьную строку, обозначающую систему, в которой выполняется интерпретатор Python: `'win32'`,

'darwin', 'linux2', 'cygwin', 'os2', 'freebsd8', 'sunos5', 'PalmOS3' и т.д. Этой переменной удобно пользоваться для организации проверок в платформенно-зависимом коде.

Для всех текущих версий Windows эта переменная принимает строковое значение 'win32', но проверка типа платформы sys.platform[:3]=='win' или sys.platform.startswith ('win') допускается ради обобщенности. Начиная с версии Python 3.3 для всех платформа Linux данная переменная принимает строковое значение 'linux', но в сценариях следует организовать аналогичные проверки типа платформы с помощью функции *str*.startswith('linux'), как это делалось ранее с помощью строкового значения 'linux2' или 'linux3'.

sys.prefix

Этой переменной присваивается символьная строка, предоставляющая префикс того каталога, где установлены независящие от платформы файлы Python. По умолчанию в этой переменной хранится префикс каталога ***/usr/local*** или аргумент времени компоновки. Библиотечные модули Python установлены в каталоге ***<prefix>/lib/python<версия>***, а независящие от платформы заголовочные файлы хранятся в каталоге ***<prefix>/include/python<версия>***.

sys.ps1

Предоставляет символьную строку, обозначающую первичное приглашение в диалоговом режиме. По умолчанию это приглашение **>>>**, если не назначено иное.

sys.ps2

Предоставляет символьную строку, обозначающую вторичное приглашение для продолжения ввода составных операторов в диалоговом режиме. По умолчанию это приглашение . . ., если не назначено иное.

sys.setcheckinterval(*частота_повторов*)

В версии Python 3.2 эта функция заменена описываемой далее функцией `setswitchinterval()`, тем не менее она по-прежнему присутствует, хотя и не имеет особого значения, поскольку был полностью переделан механизм, реализующий переключение потоков исполнения и асинхронное выполнение задач.

До версии Python 3.2 эта функция вызывалась с целью установить *частоту_повторов* периодически выполняемых задач (например, переключателей потоков исполнения, обработчиков сигналов), измеряемую в инструкциях виртуальной машины (по умолчанию **100** таких инструкций). В общем, оператор Python преобразуется во многие инструкции виртуальной машины. И чем меньше величина *частоты_повторов*, тем оперативнее ответная реакция потоков исполнения, но больше издержки на их переключение.

sys.setdefaultencoding(*имя*)

Эта функция исключена из версии Python 3.2. Она вызывается с целью установить текущую стандартную кодировку, применяемую в реализации уникода, в соответствии со значением аргумента *имя*. Предназначается для применения в модуле `site` и доступна только во время запуска.

sys.setprofile(*функция*)

Вызывается с целью установить *функцию* системного профиля, т.е. "перехватчик" для профилировщика, но выполняется не для каждой строки кода. Подробнее об этом см. в руководстве по библиотеке Python.

sys.setrecursionlimit(*глубина*)

Вызывается с целью установить *глубину* стека вызовов в Python. Этот предел препятствует возникновению бесконечной рекурсии, приводящей к переполнению стека C и аварийному завершению интерпретатора Python. По умолчанию аргумент *глубина* равен **1000** в Windows, но

он может быть изменен. Большие значения аргумента *глубина* требуется устанавливать для глубоко рекурсивных функций.

sys.setswitchinterval(*интервал*)

Начиная с версии Python 3.2 эта функция устанавливает заданный *интервал* переключения потоков исполнения в секундах. Значение аргумента *интервал* является числовым с плавающей точкой (например, значение **0,005** соответствует 5 миллисекундам) и определяет идеальную продолжительность интервала времени, выделяемого для параллельно исполняющихся потоков в Python. Конкретное значение аргумента *интервал* может быть большим, особенно если применяются долго выполняющиеся функции или методы, а выбор потока исполнения, запланированного в конце интервала, делается операционной системой. (У интерпретатора Python отсутствует свой планировщик.) До версии Python 3.2 вместо данной функции использовалась описанная выше функция `setcheckinterval()`.

sys.settrace(*функция*)

Вызывается с целью установить системную *функцию* трассировки. Эта функция используется отладчиками и прочими инструментальными средствами в качестве "перехватчика" обратных вызовов, возникающих при изменении местоположения или состояния программы.

sys.stdin

Предоставляет предварительно открываемый файловый объект, первоначально связанный со стандартным потоком ввода `sys.stdin`. С помощью методов типа `read` этой переменной может быть присвоен любой объект для установки в исходное состояние потока ввода в сценарии (например, `sys.stdin = MyObj()`). Используется для ввода данных в интерпретаторе Python, включая встроенную функцию `input()`, а также функцию `raw_input()` в версии Python 2.X.

sys.stdout

Предоставляет предварительно открываемый файловый объект, первоначально связанный со стандартным потоком вывода `sys.stdout`. С помощью методов типа `write` этой переменной может быть присвоен любой объект для установки в исходное состояние потока вывода в сценарии (например, `sys.stdout=open('log', 'a')`). Используется для вывода некоторых приглашений и во встроенной функции `print()`, а в версии Python 2.X — в операторе `print`. Если же требуется переопределить зависящую от платформы кодировку, то лучше воспользоваться переменной окружения `PYTHONIOENCODING` (см. выше раздел "Переменные окружения Python"), а для организации небуферизованных потоков ввода-вывода — параметром командной строки **-u** (см. выше раздел "Параметры командной строки в версии Python 2.X").

sys.stderr

Предоставляет предварительно открываемый файловый объект, первоначально связанный со стандартным потоком вывода ошибок `sys.stderr`. С помощью методов типа `write` этой переменной может быть присвоен любой объект для установки в исходное состояние потока вывода ошибок в сценарии (например, `sys.stderr=`*`заключенный_в_оболочку_сокет`*). Используется для вывода сообщений об ошибках в интерпретаторе Python.

sys.__stdin__, sys.__stdout__, sys.__stderr__

Предоставляют исходные значения стандартных потоков ввода-вывода `sys.stdin`, `sys.stdout` и `sys.stderr` в начале работы программы (например, для восстановления в качестве последнего средства; как правило, служат для сохранения прежнего значения перед его присваиванием переменной `sys.stdout` и т.д. и последующего восстановления в операторе `finally`). *Примечание:* эти атрибуты могут принимать значение `None` для приложений Windows, работающих в режиме ГПИ, а не консоли.

sys.thread_info

Предоставляет подробные сведения о реализации потоков исполнения в интерпретаторе Python через атрибуты, начиная с версии Python 3.3 (подробнее об этом см. в руководстве по Python).

sys.tracebacklimit

Предоставляет максимальное количество уровней обратной трассировки стека для вывода неперехваченных исключений. По умолчанию это количество равно **1000**, если не указано иное.

sys.sys.version

Предоставляет символьную строку, содержащую номер версии интерпретатора Python.

sys.version_info

Предоставляет кортеж, содержащий составляющие, обозначающие версию интерпретатора Python: основной, дополнительный, вспомогательный и серийный номер версии. Так, для версии Python 3.0.1 эта переменная предоставляет следующий кортеж: `(3, 0, 1, 'final', 0)`. В последних выпусках Python (и только в них) эта переменная предоставляет именованный кортеж, составляющие которого могут быть доступны в виде отдельных элементов кортежа или имен атрибутов. Так, для версии Python 3.3.0 выводится следующий результат: `sys.version_info(major=3, minor=3, micro =0, releaselevel='final', serial=0)`. Подробнее об этом см. в руководстве по библиотеке Python.

sys.winver

Предоставляет номер версии, используемый для формирования ключей системного реестра на платформе Windows. Доступна только в Windows; подробнее об этом см. в руководстве по библиотеке Python.

Модуль string

В модуле string определяются константы и переменные для обработки *строковых объектов*. Дополнительно о средствах типа Template и Formatter, определенных в этом модуле для подстановки шаблонов и форматирования строк, см. выше, в разделе "Символьные строки".

Функции и классы

В версии Python 2.0 все функции из рассматриваемого здесь модуля стали доступны также в виде *методов* строковых объектов. Их вызовы в виде методов более эффективны и предпочтительны в 2.X, тогда как в версии 3.X они сохранились как единственная возможность. Подробнее обо всех строковых методах см. выше, в разделе "Символьные строки". А в этом разделе рассматриваются только те методы, которые характерны для модуля string.

string.capwords(*s*, sep=None)

Разделяет аргумент *s* на отдельные слова с помощью строкового метода *s*.split(), начиная каждое слово с прописной буквы с помощью строкового метода *s*.capitalize() и затем соединяя начинающиеся с прописной буквы слова с помощью строкового метода *s*.join(). Если дополнительный аргумент sep отсутствует или равен None, то следующие подряд пробельные символы заменяются единственным пробелом, а начальные и конечные пробелы удаляются. В противном случае аргумент sep используется для разделения и соединения слов.

string.maketrans(*from*, *to*)

Возвращает таблицу преобразования, пригодную для передачи строковому методу bytes.translate(), которая преобразует каждый символ из исходной строки *from* в символ на той же самой позиции в целевой строке *to*. Обе символьные строки, *from* и *to*, должны быть одинаковой длины.

string.Formatter

Этот класс служит для создания специальных средств форматирования символьных строк с помощью того же самого механизма, что и у метода `str.format()`, как пояснялось ранее, в подразделе "Метод форматирования символьных строк".

string.Template

Этот класс служит для подстановки шаблонов символьных строк, как пояснялось ранее, в подразделе "Подстановка шаблонных символьных строк".

Константы

string.ascii_letters

Содержит символьную строку, состоящую из констант `ascii_lowercase` и `ascii_uppercase`.

string.ascii_lowercase

Содержит символьную строку `'abcdefghijklmnopqrstuvwxyz'`. Не зависит от региональных настроек и не подлежит изменению.

string.ascii_uppercase

Содержит символьную строку `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Не зависит от региональных настроек и не подлежит изменению.

string.digits

Содержит символьную строку `'0123456789'`.

string.hexdigits

Содержит символьную строку `'0123456789abcdefABCDEF'`.

string.octdigits

Содержит символьную строку `'01234567'`.

string.printable

Содержит сочетание констант digits, ascii_letters, punctuation и whitespace.

string.punctuation

Содержит строку, состоящую из символов, которые считаются знаками препинания в региональных настройках.

string.whitespace

Содержит строку, состоящую из последовательностей символов, обозначающих пробел, табуляцию, перевод строки, вертикальную табуляцию и перевод формата: **' \t\n\r \v\f'**.

Модуль os

Модуль os служит основным интерфейсом для сопряжения со службами *операционной системы* (ОС) в обеих версиях Python 2.X и 3.X. Он обеспечивает общую поддержку ОС и доступ к стандартным, не зависящим от конкретной платформы системным утилитам. В состав модуля os входят инструментальные средства для обращения к среде исполнения, процессам, файлам, командной оболочке и прочим системным ресурсам. В его состав входит также вложенный подмодуль os.path, предоставляющий переносимый интерфейс для сопряжения с инструментальными средствами обращения к каталогам.

Сценарии, в которых модули os и os.path используются для целей системного программирования, как правило, оказываются переносимыми на большинстве платформ Python. Тем не менее некоторые операции экспорта из модуля os доступны не на всех платформах (например, функция os.fork() доступна на платформах Unix и Cygwin, но не является стандартной в версии Python для Windows). Переносимость таких вызовов может со временем изменяться, поэтому за дополнительной справкой рекомендуется обращаться к руководству по библиотеке Python для конкретной платформы.

В последующих разделах рассматриваются лишь наиболее употребительные инструментальные средства из модуля `os`, а полный их перечень (около 200), а также их отличия на разных платформах см. в руководстве по стандартной библиотеке Python на некоторых платформах. В приведенном далее описании отражены лишь основные функциональные возможности этого крупного модуля, в том числе следующие.

- *Административные средства.* Операции экспорта модулей по заданному пути.
- *Константы переносимости.* Константы поиска файлов и каталогов модулей по заданному пути.
- *Средства командной оболочки.* Для выполнения команд и файлов из командной строки.
- *Средства среды исполнения.* Среда и контекст выполнения.
- *Средства дескрипторов файлов.* Обработка файлов по их дескрипторам.
- *Средства имен путей к файлам.* Обработка файлов по именам путей к ним.
- *Управление процессами.* Создание процессов и управление ими.
- *Модуль* **`os.path`**. Службы, связанные с именами путей к каталогам.

Дополнительно о системных модулях см. в руководстве по стандартной библиотеке Python, если не указано иное, а также в следующих разделах этой книги.

- Модуль `sys` — инструментальные средства процессов интерпретатора Python — раздел "Модуль `sys`".
- Модуль `subprocess` — команды управления порожденными процессами — раздел "Модуль `subprocess`".
- Модуль `queue` — инструментальные средства многопоточной обработки — раздел "Модули многопоточной обработки".
- Модуль `socket` — работа в сети и доступ к Интернету — раздел "Модули и средства доступа к Интернету".

- Модуль `glob` — расширение имен файлов (например, `glob.glob('*.py')`).
- Модуль `tempfile` — обращение с временными файлами.
- Модуль `signal` — обработка сигналов.
- Модуль `multiprocessing` — потокообразный прикладной программный интерфейс API для процессов.
- Модули `getopt`, `optparse` и `argparse`, начиная с версии 3.2, — для работы в режиме командной строки.

Административные средства

Ниже перечислены различные административные средства для операций экспорта модулей.

`os.error`

Предоставляет псевдоним для встроенного исключения типа `OSError` (см. выше раздел "Встроенные исключения"). Это исключение возникает при появлении всех ошибок, связанных с модулями. У этого исключения имеются следующие два атрибута: `errno` — числовой код ошибки по стандарту POSIX (например, значение переменной `errno` в языке C); а также `strerror` — соответствующее сообщение об ошибке, предоставляемое операционной системой и форматируемое базовыми функциями на C (например, функцией `perror()` при вызове `os.strerror()`). Что же касается исключений, связанных с именами путей к файлам (например, при вызове функций `chdir()` и `unlink()`), то экземпляр исключения также содержит атрибут `filename` с именем передаваемого файла. Подробнее об наименованиях кодов ошибок в базовой ОС см. описание модуля `errno` в руководстве по библиотеке Python.

`os.name`

Предоставляет имя отдельного модуля ОС, имена из которого (например, `posix`, `nt`, `mac`, `os2`, `ce` или `java`) копируются на верхний уровень модуля `os`. Дополнительно см. описание переменной `sys.platform` выше, в разделе "Модуль `sys`".

os.path

Вложенный модуль для переносимых утилит, связанных с именами путей к файлам и каталогам. Например, платформенно-независимая функция `s.path.split()` выполняет действие над именем пути к каталогу на конкретной платформе.

Константы переносимости

В этом разделе описываются *константы переносимости* для путей поиска файлов и каталогов, символов перевода строки и прочего. В них автоматически устанавливаются значения, характерные для тех платформ, на которых выполняется сценарий. Они полезны как для синтаксического анализа, так и для составления символьных строк, не зависящих от конкретной платформы. Дополнительно см. далее раздел "Модуль `os.path`".

os.curdir

Содержит символьную строку, представляющую текущий каталог (например, `'.'` для Windows и POSIX или `':'` для Mac OS).

os.pardir

Содержит символьную строку, представляющую родительский каталог (например, `'..'` для POSIX или `':::'` для Mac OS).

os.sep

Содержит символьную строку, используемую для разделения каталогов (например, `'/'` для Unix, `'\'` для Windows или `':'` для Mac OS).

os.altsep

Содержит альтернативную символьную строку для разделения каталогов или значение `None` (например, `'/'` для Windows).

os.extsep

Содержит символ, отделяющий базовое имя файла от его расширения (например, символ `'.'`).

os.pathsep

Содержит символ, используемый для разделения составляющих пути поиска файлов и каталогов, как в переменных окружения PATH и PYTHONPATH (например, '; ' для Windows или ':' для Unix).

os.defpath

По умолчанию содержит путь к файлам и каталогам, используемый при вызовах функций типа os.exec*p* в отсутствие переменной окружения PATH в командной оболочке.

os.linesep

Содержит символьную строку, используемую для окончания строк на текущей платформе (например, '\n' для POSIX, символьную строку '\r' для Mac OS или '\r\n' для Windows). Не требуется при записи строк в файлы, находящиеся в текстовом режиме, если в файловом объекте выполняется автоматическое преобразование последовательности символов '\n' (дополнительно см. описание функции open() выше, в разделе "Встроенные функции").

os.devnull

Содержит путь к файлу фиктивного устройства (для отвергаемого текста). Для POSIX это символьная строка '/dev/null', а для Windows — 'null'. Эта константа доступна также в подмодуле os.path.

Средства командной оболочки

Функции данной категории выполняют команды или файлы из *командной строки* базовой ОС. В версии Python 3.X обращения к разновидностям функции os.popen2/3/4() из версии Python 2.X были заменены на класс subprocess.Popen, как правило, обеспечивающий более точное управление порожденными процессами (см. ниже раздел "Модуль subprocess"). *Совет:* эти средства не следует применять для запуска команд оболочки из ненадежных символьных строк, поскольку они позволяют выполнить любую команду, разрешенную для процесса Python.

os.system(*команда*)

Выполняет содержимое символьной строки *команда* из командной строки подчиненного процесса командной оболочки. Возвращает состояние порожденного процесса. В отличие от функции `popen()`, не устанавливает связь символьной строки *команда* со стандартными потоками ввода-вывода через конвейеры. *Совет:* для выполнения команды в фоновом режиме работы Unix следует добавить знак **&** в конце символьной строки *команда* (например, `os.system('python main.py &')`), а для запуска программ в Windows проще воспользоваться командой `start` в режиме DOS работы Windows (например, `os.system ('startfile.html')`).

os.startfile(*имя_пути_к_файлу*)

Запускает файл вместе с тем приложением, с которым он сопоставляется. Действует подобно двойной проверке файла в Проводнике Windows или предоставлению имени файла в качестве аргумента команды `start` в Windows (например, `os.system('start` *путь* `')`). Файл открывается в том приложении, с которым сопоставляется его расширение. Вызов происходит без ожидания и, как правило, без открытия окна консоли Windows с *приглашением командной строки*. Эта функция стала поддерживаться только с версии Windows 2.0.

os.popen(*команда*, mode='r', buffering=None)

Открывает конвейер, направляющий к символьной строке *команда* для передачи или приема данных в режиме командной строки. Возвращает объект открытого файла, который может быть использован для чтения из стандартного потока вывода (по умолчанию `mode='r'`) или записи в стандартный поток ввода (`mode='w'`) символьной строки *команда*. Например, в выражении `dirlist = os.popen('ls -l *.py').read()` читается результат выполнения команды `ls` в Unix.

Первый аргумент, *команда*, является символьной строкой с любой командой, которую можно ввести из системной

консоли или по приглашению командной строки. Второй аргумент, mode, может быть равен 'w' или 'r' (по умолчанию). А третий аргумент, buffering, такой же, как и у встроенной функции open(). Выполнение *команды* происходит независимо, а код ее завершения возвращается методом close() результирующего файлового объекта, за исключением того, что значение None возвращается в том случае, если код завершения равен нулю (т.е. без ошибок). Для чтения результата построчно следует вызвать функцию readline() или организовать итерацию файлового объекта, а возможно, и чередовать обе операции для большей полноты.

В версии Python 2.X имеются также варианты popen2(), popen3() и popen4() данной функции для установления связи с другими потоками ввода-вывода порождаемого процесса (например, функция popen2() возвращает кортеж (*порожденный_стандартный_поток_ввода*, *порожденный_стандартный_поток_вывода*)). Эти функции исключены из версии Python 3.X, а вместо них применяется класс subprocess.Popen. Начиная с версии 2.4 модуль subprocess позволяет порождать новые процессы в сценариях, связывать их с потоками ввода-вывода и получать из них коды завершения (см. далее раздел "Модуль subprocess").

os.spawn*(*args*...)

Это семейство функций, предназначенных для порождения процессов, в которых выполняются программы и команды. Подробнее об этом см. ниже, в разделе "Управление процессами", а также в руководстве по библиотеке Python. В качестве альтернативы этим функциям может служить модуль subprocess (см. далее раздел "Модуль subprocess").

Средства среды исполнения

Перечисленные ниже атрибуты и функции служат для экспорта *контекста исполнения*: окружения командной оболочки, текущего каталога и пр.

os.environ

Предоставляет словарноподобный объект для переменных окружения командной оболочки. В частности, os.environ['USER'] — это значение переменной USER окружения командной оболочки, равнозначное значению переменной окружения $USER в Unix или переменной окружения %USER% в Windows. Этот атрибут инициализируется при запуске программы. Изменения, происходящие в атрибуте os.environ в результате присваивания ключей, экспортируются за пределы среды исполнения Python благодаря вызову функции putenv() в языке C и наследуются любыми процессами, которые так или иначе порождаются в дальнейшем, а также любым связанным кодом на C. Подробнее об интерфейсе типа bytes для сопряжения с окружением, обозначаемым атрибутом os.environb, начиная с версии 3.2, см. в рукододстве по Python.

os.putenv(*имя_переменной*, *значение*)

Устанавливает символьную строку *значение* в переменной окружения *имя_переменной*. Воздействует на подчиненные процессы, запускаемые функциями system(), popen(), spawnv(), fork() и execv() и т.д. В результате присваивания ключей атрибуту os.environ автоматически вызывается функция os.putenv(), но при этом атрибут os.environ не обновляется, поэтому лучше пользоваться атрибутом os.environ.

os.getenv(*имя_переменной*, default=None)

Возвращает значение переменной окружения *имя_переменной*, если таковая существует, а иначе значение аргумента default. В настоящее время просто индексирует словарь переменных окружения, предварительно загружаемый в результате вызова функции os.environ.get(*имя_переменной*, default). Аргументы *имя_переменной*, default и результат относятся к типу str. Описание эквивалентной функции os.getenvb() для типа bytes, начиная с версии Python 3.2, а также правила кодирования в уникоде см. в руководстве по Python.

os.getcwd()

Возвращает имя текущего рабочего каталога в виде символьной строки.

os.chdir(*путь*)

Изменяет текущий рабочий каталог на указанный *путь* для данного процесса. Последующие операции с файлами выполняются относительно нового рабочего каталога, который становится текущим. *Совет:* эта функция не обновляет переменную `sys.path`, используемую для импорта модулей, хотя первая запись в ней может служить общим обозначением для текущего рабочего каталога.

os.strerror(*код*)

Возвращает сообщение об ошибке, соответствующее заданному *коду*.

os.times()

Возвращает пятиэлементный кортеж, содержащий сведения о времени ЦП, которое истекло для вызывающего процесса и выражается в числовых значениях секунд с плавающей точкой: (*время_пользователя*, *системное_время*, *время_порожденного_пользователя*, *время_порожденной_системы*, *истекшее_реальное_время*). Дополнительно см. далее раздел "Модуль `time`".

os.umask(*маска*)

Устанавливает значение аргумента *маска* в числовой переменной `umask` и возвращает предыдущее значение.

os.uname()

Возвращает кортеж символьных строк со сведениями о системе: (*имя_системы*, *имя_узла*, *выпуск*, *версия*, *машина*).

Средства дескрипторов файлов

Перечисленные ниже функции служат для обработки *файлов* по их *дескрипторам*, где аргумент *fd* — целочисленное значение дескриптора файла. В модуле `os` файлы, доступные по

дескрипторам, предназначаются для решения низкоуровневых задач и не похожи на файловые объекты, возвращаемые встроенной функцией `open()` из модуля `stdio`. Для обработки файлов, как правило, должны использоваться файловые *объекты*, а не *дескрипторы* файлов. Если требуется, взаимное преобразование обеих форм выполняется функцией `os.fdopen()` и методом `fileno()` файлового объекта, тогда как встроенная функция `open()` принимает дескриптор файла в версии 3.X. Подробнее о функции `open()` см. выше, в разделе "Встроенные функции".

НА ЗАМЕТКУ

Рассматриваемые здесь *дескрипторы файлов* отличаются от *дескрипторов классов* (см. выше раздел "Методы для операций с дескрипторами"). Следует также иметь в виду, что в версию Python 3.4 внесено изменение, чтобы дескрипторы файлов *не* наследовались подчиненными процессами по умолчанию. И для этой цели в модуле `os` предоставляются новые функции `get_inheritable(fd)` и `set_inheritable(fd, boolean)`.

`os.close(fd)`

Закрывает дескриптор файла `fd`, но не файловый объект.

`os.dup(fd)`

Возвращает дубликат дескриптора файла `fd`.

`os.dup2(fd, fd2)`

Копирует дескриптор файла `fd` в дескриптор файла `fd2`, закрывая сначала дескриптор файла `fd2`, если он открыт.

`os.fdopen(fd, *args, **kwargs)`

Возвращает файловый *объект*, встроенный в модуль `stdio` и связанный с дескриптором файла `fd` (целочисленным значением). Эта функция является псевдонимом встроенной функции `open()` и принимает те же самые аргументы, за исключением того, что первый аргумент функции `fdopen()` должен всегда быть целочисленным

дескриптором файла (дополнительно см. описание функции open() выше, в разделе "Встроенные функции"). Файловые объекты, как правило, создаются путем автоматического преобразования из файлов, доступных по дескрипторам, во встроенной функции open(). *Совет:* для преобразования файлового объекта в дескриптор файлов следует сделать вызов *файловый_объект*.fileno().

os.fstat(*fd*)

Возвращает состояние дескриптора файла *fd* (аналогично функции os.stat()).

os.ftruncate(*fd*, *длина*)

Урезает файл, доступный по дескриптору fd, чтобы его размер в байтах был не больше заданной *длины*.

os.isatty(*fd*)

Возвращает логическое значение True, если дескриптор файла fd открыт и связан с устройством терминального типа, а иначе — логическое значение False. В прежних версия Python может возвращать значения **1** и **0** соответственно.

os.lseek(*fd*, *pos*, *how*)

Устанавливает текущую позицию дескриптора файла *fd* в соответствии со значением аргумента pos для произвольного доступа к данным. Если аргумент *how* равен нулю, то новая позиция дескриптора файла устанавливается относительно начала файла; если этот аргумент равен **1** — относительно текущей позиции, если он равен **2** — относительно конца файла.

os.open(*имя_файла*, *признаки*[, mode=0o777], [dir_fd=None])

Открывает файл, доступный по дескриптору, и возвращает его дескриптор — целочисленное значение, которое может быть передано другой функции обработки файлов из модуля os, но не файловому объекту из модуля stdio. Предназначена для решения только низкоуровневых задач, но не равнозначна встроенной функции open(), которой

следует отдавать предпочтение в большинстве операций обработки файлов (см. выше раздел "Встроенные функции").

Аргумент *имя_файла* обозначает имя пути (возможно, относительного) к файлу, тогда как аргумент *признаки* — битовую маску. Для объединения зависящих и не зависящих от конкретной платформы констант признаков, определенных в модуле `os` (табл. 18), следует использовать логический оператор `|`. По умолчанию аргумент `mode` принимает восьмеричное значение **`0o777`**, чтобы маскировать текущее значение переменной `umask`. Аргумент `dir_fd` появился в версии Python 3.3 и поддерживает пути относительно дескрипторов файлов в каталоге (подробнее об этом см. в руководстве по Python). *Совет:* функцию `os.open()` следует вызывать с признаком `os.O_EXCL`, чтобы обеспечить переносимость блокировки файлов при параллельных обновлениях и прочих видах синхронизации процессов.

`os.pipe()`

Создает анонимный конвейер. Дополнительно см. далее раздел "Управление процессами".

`os.read(`*`fd, n`*`)`

Читает не меньше *n* байтов из файла по его дескриптору *fd* и возвращает их в виде символьной строки.

`os.write(`*`fd, строка`*`)`

Записывает все байты из символьной *строки* в файл по его дескриптору *fd*.

Таблица 18. Признаки, определенные для функции `os.open()` в модуле `os`

`O_APPEND`	`O_APPEND`	`O_RDONLY`	`O_TRUNC`
`O_BINARY`	`O_NDELAY`	`O_RDWR`	`O_WRONLY`
`O_CREAT`	`O_NOCTTY`	`O_RSYNC`	
`O_DSYNC`	`O_NONBLOCK`	`O_SYNC`	

Средства имен путей к файлам

Перечисленные ниже функции служат для *обработки* файлов по именам путей к ним, где *путь* — символьная строка с именем пути к файлу. Дополнительно см. далее раздел "Модуль `os.path`". В версии Python 2.X в этот модуль включены также средства обработки временных файлов, замененные модулем `tempfile` в версии Python 3.X. В версии Python 3.3 некоторые из этих средств были дополнены необязательным, и поэтому не упоминаемым здесь аргументом `dir_fd` для поддержки путей относительно дескрипторов файлов в каталоге. Подробнее об этом см. в руководстве по Python.

`os.chdir(путь)`
`os.getcwd()`

Это средства для текущих рабочих каталогов. Дополнительно см. выше раздел "Средства среды исполнения".

`os.chmod(путь, режим)`

Изменяет режим *пути* к файлу в соответствии с числовым значением аргумента *режим*.

`os.chown(путь, uid, gid)`

Изменяет идентификаторы владельца и группы *пути* в соответствии с числовыми значениями аргументов *uid* и *gid*.

`os.link(исходный_путь, целевой_путь)`

Создает жесткую связь, называемую *целевым_путем*, с *исходным_путем* к файлу.

`os.listdir(путь)`

Возвращает список имен всех элементов в *пути* к каталогу. Является быстрой и переносимой альтернативой вызову функции `glob.glob(`*шаблон*`)` и выполнению команд перечисления из командной оболочки в функции `os.popen()`. Подробнее о расширении шаблонов для имен файлов см. описание модуля `glob` в руководстве по Python, а о полном обходе дерева каталогов — приведенное далее описание функции `os.walk()`.

В версии Python 3.X этой функции передается и ею возвращается значение типа bytes, а не str, чтобы подавить декодирование имени файла в уникоде, выполняемое на каждой платформе по умолчанию (такое поведение распространяется и на функции glob.glob() и os.walk()). А начиная с версии Python 3.2 аргумент *путь* по умолчанию равен ".". Если он опускается, то выбирается текущий рабочий каталог.

os.lstat(*путь*)

Действует аналогично функции os.stat(), но не следует символическим ссылкам.

os.mkfifo(*путь* [, mode=0o666])

Создает именованный конвейер обратного магазинного типа (FIFO), обозначаемый строковым аргументом *путь* с правами доступа, задаваемыми числовым значением аргумента mode, но не открывает его. По умолчанию аргумент mode принимает восьмеричное значение **0o666**, которым сначала маскируется текущее значение переменной umask. В версии 3.3 у этой функции появился дополнительный, но *не* обязательный именованный аргумент dir_fd.

Конвейеры обратного магазинного типа, доступные в файловой системе, могут открываться и обрабатываться как и обычные файлы, но они поддерживают синхронизированный доступ к файлам по имени среди независимо запускаемых клиентов и серверов. Конвейеры обратного магазинного типа существуют до тех пор, пока их не удалить. В настоящее время данная функция доступна на Unix-подобных платформах, включая Cygwin под Windows, но не в стандартной версии Python для Windows. Аналогичных целей зачастую позволяют добиться и сокеты (см. описание модуля socket ниже, в разделе "Модули и средства доступа к Интернету", а также в руководстве по Python).

os.mkdir(*путь* [, *режим*])

Создает каталог по указанному *пути* в заданном *режиме*. По умолчанию аргумент *режим* принимает восьмеричное значение **0o777**.

os.makedirs(*путь* [, *режим*])

Эта функция создает каталоги рекурсивно. Она действует аналогично функции `mkdir()`, но создает все каталоги промежуточного уровня, необходимые для хранения листового каталога. Генерирует исключение, если листовой каталог уже существует или не может быть создан. По умолчанию аргумент *режим* принимает восьмеричное значение **0o777**. В версии 3.2 у этой функции появился дополнительный, но *не* обязательный аргумент `exists_ok`. Подробнее об этом см. в руководстве по Python.

os.readlink(*путь*)

Возвращает путь по символической ссылке, передаваемой в качестве аргумента *путь*.

os.remove(*путь*)
os.unlink(*путь*)

Удаляют файл, обозначаемый по заданному *пути*. Функция `remove()` действует аналогично функции `unlink()`. См. также приведенное ниже описание функций `rmdir()` и `removedirs()` для удаления каталогов.

os.removedirs(*путь*)

Эта функция удаляет каталоги рекурсивно. Она действует аналогично функции `rmdir()`, но если листовой каталог удален успешно, то каталоги, соответствующие крайним справа элементам пути, отсекаются до тех пор, пока не будет употреблен весь путь, а иначе возникает ошибка. Генерирует исключение, если нельзя удалить листовой каталог.

os.rename(*исходный_путь*, *целевой_путь*)

Переименовывает (перемещает) файл по *исходному_пути* в файл по *целевому_пути*. См. также описание функции `os.replace()`, доступной с версии 3.3 в руководстве по Python.

os.renames(*прежний_путь*, *новый_путь*)

Эта функция рекурсивно выполняет переименование каталога или файла. Она действует аналогично функции

rename(), но сначала создает любые промежуточные каталоги, необходимые для того, чтобы сделать действительным имя нового пути. После переименования каталоги, соответствующие крайним справа составляющим в имени прежнего пути, отсекаются с помощью функции removedirs().

os.rmdir(*путь*)

Удаляет каталог по заданному *пути*.

os.stat(*путь*)

Делает вызов системного модуля stat по заданному *пути*. Возвращает кортеж целочисленных значений с низкоуровневой информацией о файле. Отдельные элементы этой информации определяются и обрабатываются средствами стандартного библиотечного модуля stat.

os.symlink(*исходный_путь*, *целевой_путь*)

Создает символическую ссылку по *целевому_пути* на файл по *исходному_пути*.

os.utime(*путь*, (*atime*, *mtime*))

Устанавливает время доступа (*atime*) и время модификации (*mtime*) файла по заданному *пути*.

os.access(*путь*, *режим*)

Обращается за подробными сведениями к руководству по библиотеке Python или справочным страницам Unix.

os.walk(...)

```
os.walk(вершина
  [, topdown=True
  [, onerror=None]
  [, followlinks=False]]])
```

Формирует имена файлов в дереве каталогов, обходя его сверху вниз или снизу вверх. Выдает для каждого каталога, расположенного в этом дереве относительно корневого каталога по пути, определяемому символьной строкой *вершина*, включая и саму *вершину*, трехэлементный (*тройной*) кортеж (*путь_к_каталогу*, *имена_каталогов*, *имена_файлов*), где

- *путь_к_каталогу* — символьная строка, обозначающая путь к каталогу;
- *имена_каталогов* — список имен подкаталогов в каталоге по *пути_к_каталогу*, исключая . и ..;
- *имена_файлов* — список имен файлов в каталоге по *пути_к_каталогу*.

Следует иметь в виду, что имена в списках не содержат составляющие пути. Для того чтобы получить полный путь к файлу или каталогу, который начинается с *вершины* дерева каталогов в *пути_к_каталогу*, нужно сделать вызов `os.path.join(`*путь_к_каталогу*`,` *имя*`)`.

Если дополнительный аргумент `topdown` принимает логическое значение `True` или вообще не указан, то тройной кортеж для каталога формируется *перед* тройными кортежами для каждого из его подкаталогов (в этом случае каталоги формируются сверху вниз). А если дополнительный аргумент `topdown` принимает логическое значение `False`, то тройной кортеж для каталога формируется *после* тройных кортежей для всех его подкаталогов (в этом случае каталоги формируются снизу вверх). Если же указан еще один дополнительный аргумент `onerror`, он должен обозначать функцию, которая вызывается с единственным аргументом — экземпляром `os.error`. По умолчанию функция `os.walk()` не выполняет обход по символическим ссылкам, разрешающим каталоги, поэтому для обращения к каталогам, указываемым по таким ссылкам в тех системах, где они поддерживаются, следует задать дополнительный аргумент `followlinks` с логическим значением `True`.

Если аргумент `topdown` принимает логическое значение `True`, то список *имена_каталогов* может быть видоизменен непосредственно для управления поиском каталогов. Ведь функция `os.walk()` рекурсивно обходит только те подкаталоги, имена которых сохраняются в списке *имена_каталогов*. Это удобно для укорочения поиска, задания конкретного порядка обхода дерева каталогов и т.д.

В версии Python 2.X представляется также функция `os.path.walk()` с аналогичными возможностями для обхода дерева

каталогов, но с помощью обратного вызова вместо генератора. Эта функция исключена из версии Python 3.X в силу ее избыточности, поэтому вместо нее следует использовать функцию `os.walk()`. О связанных с этим расширением именах файлов (например, `glob.glob(r'*\*\*.py')`) см. также описание модуля `glob` в руководстве по Python.

Управление процессами

Перечисленные ниже функции служат для создания и управления *процессами* и *программами*. О других способах запуска программ и файлов см. выше, в разделе "Средства командной оболочки". *Совет:* эти средства не следует применять для запуска команд оболочки из ненадежных символьных строк, поскольку они позволяют выполнить любую команду, разрешенную для процесса Python.

`os.abort()`

Посылает сигнал `SIGABRT` текущему процессу. В Unix по умолчанию производится разгрузка (так называемый *дамп*) оперативной памяти, а в Windows процесс немедленно возвращает код завершения 3.

`os.execl(`*`путь`*`, `*`arg0`*`, `*`arg1`*`, ...)`

Равнозначна функции `execv(`*`путь`*`, (`*`arg0`*`, `*`arg1`*`, ...))`.

`os.execle(`*`путь`*`, `*`arg0`*`, `*`arg1`*`, ..., `*`env`*`)`

Равнозначна функции `execv(`*`путь`*`, (`*`arg0`*`, `*`arg1`*`, ...), `*`env`*`))`.

`os.execlp(`*`путь`*`, `*`arg0`*`, `*`arg1`*`, ...)`

Равнозначна функции `execv(`*`путь`*`, (`*`arg0`*`, `*`arg1`*`, ...))`.

`os.execve(`*`путь`*`, `*`args`*`, `*`env`*`)`

Действует аналогично функции `execv()`, но словарь *`env`* заменяет переменную окружения командной оболочки. Словарь *`env`* должен отображать одни символьные строки на другие.

os.execvp(*путь, args*)

Действует аналогично функции execv(*путь, args*), но дублирует действия командной оболочки при поиске исполняемого файла в списке каталогов. Список каталогов получается из элемента списка os.environ['PATH'].

os.execvpe(*путь, args, env*)

Эта функция представляет собой нечто среднее между функциями execve() и execvp(). Список каталогов получается из элемента списка os.environ['PATH'].

os.execv(*путь, args*)

Исполняет файл по заданному *пути* с аргументами командной строки *args*, заменяя текущую программу в данном процессе интерпретатора Python. В качестве аргументов командной строки *args* может быть указан кортеж или список символьных строк, который принято начинать с имени исполняемого файла (в элементе списка argv[0]). Эта функция вообще ничего не возвращает, если только при запуске новой программы не возникнет ошибка.

os._exit(*n*)

Выполняет немедленный выход из процесса с кодом состояния *n*, не совершая обычные шаги по нормальному завершению программы. Как правило, используется только в порожденном процессе после вызова функции fork(). Стандартным способом выхода из процесса является вызов функции sys.exit(*n*).

os.fork()

Порождает дочерний процесс (виртуальную копию вызывающего процесса, выполняющегося параллельно). Возвращает нулевое значение в дочернем процессе и его новый идентификатор в родительском процессе. Эта функция недоступна в стандартной версии Python для Windows, но доступна в версии Python для платформы Cygwin под Windows (функции popen(), system(), spawnv(), а также модуль subprocess, как правило, более переносимы).

os.getpid()
os.getppid()

Возвращают идентификатор текущего (вызывающего) процесса. В частности, функция `getppid()` возвращает идентификатор родительского процесса.

os.getuid()
os.geteuid()

Возвращают идентификатор пользователя процесса. В частности, функция `geteuid()` возвращает действующий идентификатор пользователя.

os.kill(*pid*, *sig*)

Посылает сигнал *sig* процессу с идентификатором *pid*, потенциально уничтожая его (для некоторых видов сигналов). См. также описание в руководстве по Python констант сигналов и регистрации их обработчиков в модуле `signal`.

os.mkfifo(*путь* [, *режим*])

См. выше раздел "Средства имен путей к файлам". Имена файлов используются для синхронизации процессов.

os.nice(*приращение*)

Вводит *приращение* в "точность" процесса, т.е. понижает его приоритет для ЦП.

os.pipe()

Возвращает кортеж из дескрипторов файлов (*readfd*, *writefd*) для чтения и записи нового анонимного (безымянного) конвейера. Служит для обмена данными между связанными вместе процессами.

os.plock(*op*)

Блокирует сегменты программы в оперативной памяти. Аргумент *op*, значение которого определяется в заголовочном файле <sys./lock.h>, обозначает блокируемые сегменты.

os.spawnv(*режим*, *путь*, *args*)

Выполняет программу по заданному *пути*, передавая аргументы *args*, указанные в командной строке. В качестве аргументов *args* может быть указан список или кортеж. Необязательный аргумент *режим* обозначает операционную константу, состоящую из следующих имен, также определенных в модуле os: P_WAIT, P_NOWAIT, P_NOWAITO, P_OVERLAY и P_DETACH. В Windows эта функция приблизительно равнозначна сочетанию функций fork() и execv(). (Функция fork() недоступна в стандартной версии Python для Windows, тогда как функции popen() и system() доступны.) Об альтернативных вариантах этой функции с более богатыми возможностями см. описание стандартного библиотечного модуля subprocess и далее раздел "Модуль subprocess".

os.spawnve(*режим*, *путь*, *args*, *env*)

Действует аналогично функции spawnv(), но передает содержимое отображения *env* в виде окружения командной оболочки порожденной программы, а иначе оно наследовалось бы из родительского процесса.

os.wait()

Ожидает завершения порожденного процесса. Возвращает кортеж с идентификатором порожденного процесса и код завершения.

os.waitpid(*pid*, *options*)

Ожидает завершения порожденного процесса с идентификатором *pid*. Аргумент *options* принимает нулевое значение для обычного применения или же значение os.WNOHANG с целью избежать зависания, если состояние порожденного процесса недоступно. Если же аргумент *pid* равен нулю, то запрос распространяется на любой порожденный процесс в группе процессов, связанных с текущим процессом. См. также описание функций, выполняющих проверку состояния завершения (например, функции WEXITSTATUS(*состояние*) для извлечения кода завершения), в руководстве по библиотеке Python.

Модуль os.path

В модуле os.path предоставляются службы и средства переносимости дополнительных файлов по *именам путей к каталогам*. Это вложенный модуль, причем имена его составляющих вложены в модуль os и находятся в подмодуле os.path. Например, для доступа к функции exists() достаточно импортировать модуль os и воспользоваться именем os.path.exists.

Большинство функций в этом модуле принимают в качестве аргумента *путь* символьную строку с именем пути к каталогу, в котором находится файл (например, 'C:\dir1\spam.txt'). Пути к каталогам, как правило, обозначаются по принятым на каждой платформе правилам и отображаются на текущий рабочий каталог в отсутствие префикса каталога. *Совет:* знаки косой черты обычно действуют в качестве разделителей каталогов на всех платформах. В состав рассматриваемого здесь модуля в версии Python 2.X входит функция os.path.walk(), которая была заменена функцией os.walk() в версии Python 3.X (см. выше раздел "Средства имен путей к файлам").

os.path.abspath(*путь*)

Возвращает нормализованный абсолютный вариант заданного *пути*. На большинстве платформ вызов этой функции равнозначен вызову normpath(join(os.getcwd(), *путь*)).

os.path.basename(*путь*)

Действует аналогично второй половине пары, возвращаемой функцией split(*путь*).

os.path.commonprefix(*список*)

Возвращает (посимвольно) самый длинный префикс пути, т.е. префикс всех путей, указанных в *списке*.

os.path.dirname(*путь*)

Действует аналогично первой половине пары, возвращаемой функцией split(*путь*).

os.path.exists(*путь*)

Возвращает логическое значение `True`, если символьная строка *путь* обозначает имя пути к существующему файлу.

os.path.expanduser(*путь*)

Возвращает символьную строку, обозначающую заданный *путь* со встроенным расширением имени пользователя `~`.

os.path.expandvars(*путь*)

Возвращает символьную строку, обозначающую заданный *путь* со встроенным расширением переменных окружения `$`.

os.path.getatime(*путь*)

Возвращает время (в секундах с момента начала эпохи) последнего доступа по заданному *пути*.

os.path.getmtime(*путь*)

Возвращает время (в секундах с момента начала эпохи) последней модификации заданного *пути*.

os.path.getsize(*путь*)

Возвращает размер (в байтах) файла по заданному *пути*.

os.path.isabs(*путь*)

Возвращает логическое значение `True`, если заданный *путь* является абсолютным.

os.path.isfile(*путь*)

Возвращает логическое значение `True`, если символьная строка *путь* обозначает обычный файл.

os.path.isdir(*путь*)

Возвращает логическое значение `True`, если символьная строка *путь* обозначает каталог.

os.path.islink(*путь*)

Возвращает логическое значение `True`, если символьная строка *путь* обозначает символическую ссылку.

os.path.ismount(*путь*)

Возвращает логическое значение `True`, если символьная строка *путь* обозначает точки монтирования тома в файловой системе.

os.path.join(*путь1* [, *путь2* [, ...]])

Оптимально соединяет несколько составляющих пути по правилам разделения, принятым на конкретной платформе.

os.path.normcase(*путь*)

Нормализует регистр в имени пути. Не оказывает никакого влияния в Unix. В других файловых системах, где регистр не учитывается, преобразует имя пути в нижний регистр, а в Windows — заменяет также знаки / на \.

os.path.normpath(*путь*)

Нормализует имя пути, сворачивая лишние разделители и ссылки верхнего уровня. В Windows заменяет также знаки / на \.

os.path.realpath(*путь*)

Возвращает в канонической форме путь к файлу по указанному его имени, исключая любые символические ссылки, обнаруживаемые в самом пути.

os.path.samefile(*путь1*, *путь2*)

Возвращает логическое значение `True`, если оба аргумента обозначают путь к одному и тому же файлу или каталогу.

os.path.sameopenfile(*fp1*, *fp2*)

Возвращает логическое значение `True`, если оба файловых объекта, представленных аргументами *fp1* и *fp2*, обозначают один и тот же файл.

os.path.samestat(*stat1*, *stat2*)

Возвращает логическое значение `True`, если оба кортежа, представленных аргументами *stat1* и *stat2*, обозначают один и тот же файл.

os.path.split(*путь*)

Разделяет *путь* на кортеж (*начало*, *конец*), где *конец* — последняя составляющая в имени пути, а *начало* — все, что следует перед составляющей *конец*. Этот кортеж аналогичен кортежу (dirname(*путь*), basename(*путь*)).

os.path.splitdrive(*путь*)

Разделяет *путь* на кортеж ('*диск*:', *конец*) (в Windows).

os.path.splitext(*путь*)

Разделяет *путь* на кортеж (*корень*, *расширение*), где последняя составляющая в *корне* не содержит ., а *расширение* оказывается пустым или содержит ..

os.path.walk(*путь*, *посетитель*, данные)

Служит альтернативой функции os.walk() в версии Python 2.X (и только в ней) и основывается на функции обратного вызова, определяемой аргументом *посетитель* с состоянием *данные* для обработки каталогов вместо генератора каталогов. Исключена из версии Python 3.X, где вместо нее следует вызывать функцию os.walk().

Модуль re сопоставления по шаблонам

Модуль re служит стандартным интерфейсом *для сопоставления с шаблонами* регулярных выражений в обеих версиях Python, 2.X и 3.X. Шаблоны регулярных выражений и сопоставляемый с ними текст указываются в виде символьных строк. Чтобы воспользоваться этим модулем, его необходимо импортировать.

Функции из модуля re

В состав интерфейса верхнего уровня рассматриваемого здесь модуля входят перечисленные ниже функции для сопоставления с непосредственно предоставляемыми или предварительно

скомпилированными шаблонами. При этом создаются шаблонные (*pobj*) и совпадающие (*mobj*) объекты, определяемые в последующих разделах.

re.compile(*шаблон* [, *признаки*])

Компилирует *шаблон* регулярного выражения в шаблонный объект (*pobj*) для последующего сопоставления. Аргумент *признаки* может объединять с помощью логического оператора | следующие признаки, доступные на верхнем уровне модуля `re`:

re.A или **re.ASCII** или **(?a)**

Выполняют сопоставление только в коде ASCII, а не полное сопоставление в уникоде по шаблонам `\w`, `\W`, `\b`, `\B`, `\s` и `\S`. Это имеет значение только для шаблонов в уникоде и ничего не значит для байтовых шаблонов. Однако признак `re.U`, а также его синоним `re.UNICODE` и встроенный аналог `?u` оставлены ради обратной совместимости. Но в версии Python 3.X они излишни, поскольку сопоставление в уникоде по умолчанию выполняется для символьных строк и не предусматривается для байтов.

re.I или **re.IGNORECASE** или **(?i)**

Выполняют сопоставление с учетом регистра.

re.L или **re.LOCALE** или **(?L)**

Выполняют сопоставление по шаблонам `\w`, `\W`, `\b`, `\B`, `\s`, `\S`, `\d` и `\D` в зависимости от текущих региональных настроек (по умолчанию в уникоде для версии Python 3.X).

re.M или **re.MULTILINE** или **(?m)**

Выполняют сопоставление с каждым символом новой строки, но не со всей символьной строкой в целом.

re.S или **re.DOTALL** или **(?s)**

Выполняют сопоставление по шаблону `.` со всеми символами, включая и символы новой строки.

re.U или **re.UNICODE** или **(?u)**

Выполняют сопоставление по шаблонам \w, \W, \b, \B, \s, \S, \d и \D в зависимости от свойств символов в уникоде. Этот признак появился в версии 2.0, но стал излишним в версии 3.X.

re.X или **re.VERBOSE** или **(?x)**

Игнорируют в шаблоне пробелы, выходящие за пределы наборов символов.

re.match(*шаблон*, *строка* [, *признаки*])

Если в начале *строки* отсутствуют или имеются символы, совпадающие со строкой *шаблон*, эта функция возвращает совпадающий объект (*mobj*), а в отсутствие совпадения — значение None. Аргумент *признаки* принимает те же значения, что и в функции compile().

re.search(*шаблон*, *строка* [, *признаки*])

Просматривается *строку* на совпадение с заданным *шаблоном*. Возвращает совпадающий объект (*mobj*), а в отсутствие совпадения — значение None. Аргумент *признаки* принимает те же значения, что и в функции compile().

re.split(*шаблон*, *строка* [, maxsplit=0])

Разделяет *строку* по совпадениям с *шаблоном*. Если для фиксации в *шаблоне* употребляются круглые скобки **()**, то возвращаются также совпадения с основными и подчиненными шаблонами.

re.sub(*шаблон*, *замена*, *строка*[, count=0])

Возвращает символьную строку, получаемую путем *замены* (при первом подсчете) крайних слева неперекрывающихся совпадений с *шаблоном* (символьной строкой или шаблонным объектом регулярного выражения) в заданной *строке*. Аргумент *замена* может быть символьной строкой или функцией, вызываемой с совпадающим объектом (*mobj*) в качестве единственного аргумента и возвращающей замещающую строку. Этот аргумент может также

включать в себя управляющие последовательности \1, \2 и так далее, которые можно использовать в качестве подстроки для сопоставления с группами символов, тогда как управляющую последовательность \0 — для сопоставления со всеми символами.

re.subn(*шаблон*, *замена*, *строка*[, count=0])

Действует аналогично функции sub(), но возвращает кортеж (*новая_строка*, *количество_сделанных_подстановок*).

re.findall(*шаблон*, *строка*[, *признаки*])

Возвращает список символьных строк со всеми неперекрывающимися совпадениями с *шаблоном* в заданной *строке*. Если в шаблоне присутствует одна группа или больше, то возвращается список групп.

re.finditer(*шаблон*, *строка* [, *признаки*])

Возвращает итерируемый объект по всем неперекрывающимся совпадениям с шаблоном регулярного выражения в заданной *строке* (совпадающим объектам).

re.escape(*строка*)

Возвращает *строку* со всеми символами, не являющимися буквенно-цифровыми, но снабженными знаками обратной косой черты, чтобы их можно было скомпилировать как строковый литерал.

Шаблонные объекты регулярных выражений

Шаблонные объекты регулярных выражений (*pobj*) возвращаются функцией re.compile() и имеют перечисленные ниже атрибуты и методы. Некоторые из этих атрибутов служат для создания совпадающих объектов (*mobj*).

***pobj*.flags**

Применяется, когда скомпилирован шаблонный объект регулярного выражения.

***pobj*.groupindex**

Предоставляет словарь элементов {*имя_группы*: *номер_группы*} в шаблоне.

***pobj*.pattern**

Предоставляет символьную строку шаблона, из которой скомпилирован шаблонный объект регулярного выражения.

***pobj*.match(*строка* [, *позиция* [, *конечная_позиция*]])**
***pobj*.search(*строка*[, *позиция* [, *конечная_позиция*]])**
***pobj*.split(*строка*[, maxsplit=0])**
***pobj*.sub(*замена*, *строка* [, count=0])**
***pobj*.subn(*замена*, *строка* [, count=0])**
***pobj*.findall(*строка* [, *позиция* [, *конечная_позиция*]])**
***pobj*.finditer(*строка* [, *позиция* [, *конечная_позиция*]])**

Все эти методы служат тем же самым целям, что и аналогичные описанные ранее функции из модуля re. Но в них *шаблон* подразумевается, тогда как аргументы *позиция* и *конечная_позиция* предоставляют индексы начала и конца символьной строки для сопоставления. В двух первых из этих методов могут быть созданы совпадающие объекты (*mobj*).

Совпадающие объекты

Совпадающие объекты (*mobj*) возвращаются в результате успешного выполнения методов match() и search() шаблонных объектов. У них имеются перечисленные ниже атрибуты и методы (подробнее о реже употребляемых и поэтому не упоминаемых здесь атрибутах и методах совпадающих объектов см. в руководстве по библиотеке Python).

***mobj*.pos, *mobj*.endpos**

Предоставляют значения аргументов *позиции* и *конечной_позиции* методам match() и search().

***mobj*.re**

Предоставляет шаблонный объект регулярного выражения, методы match() и search() которого возвращают

данный совпадающий объект (см. ниже атрибут *mobj*.pattern).

***mobj*.pattern**

Предоставляет символьную строку, передаваемую в качестве аргумента методам match() и search() шаблонного объекта.

***mobj*.group([*g* [, *g*]*])**

Возвращает подстроки, совпавшие с группами, указанными в круглых скобках в шаблоне. Принимает определенное (от нуля и больше) количество идентификаторов *g номеров* или *имен* групп, подразумеваемых шаблонами (R) и (?P<name>R) соответственно. Если передается один аргумент, то в результате возвращается подстрока, совпавшая с группой, идентификатор которой передается. Если же передается несколько аргументов, то в результате возвращается кортеж с одной совпавшей подстрокой на каждый аргумент. А если аргументы вообще не передаются, то возвращается полностью совпавшая подстрока. И если номер любой группы равен нулю, то и в этом случае возвращается полностью совпавшая подстрока. Группы в шаблоне нумеруются слева направо от **1** до **N**.

***mobj*.groups()**

Возвращает кортеж из всех сопоставляемых групп. А группы, не участвующие в сопоставлении, имеют значение None.

***mobj*.groupdict()**

Возвращает словарь, содержащий все именованные подгруппы в сопоставлении, доступные по имени подгруппы.

***mobj*.start([*g*]), *mobj*.end([*g*])**

Индексируют начало и конец подстроки, совпавшей с группой *g* (или всей совпавшей строки, если группа не указана). Так, для совпадающего объекта *M* можно составить следующее выражение: *M*.string[*M*.start(g):*M*.end(g)] == *M*.group(g).

***mobj*.span([*g*])**

Возвращает кортеж (*mobj*.start(*g*), *mobj*.end(*g*)).

***mobj*.expand(*шаблон*)**

Возвращает символьную строку, получаемую путем подстановки знака обратной косой черты в символьной строке *шаблон*, как это делается в методе sub(). Такие управляющие последовательности символов, как \n, преобразуются в соответствующие символы, а числовые обратные ссылки наподобие \1 и \2, а также именованные обратные ссылки (например, \g<1>, \g<name>) заменяются соответствующей группой.

Синтаксис шаблонов

Символьные строки шаблонов составляются с помощью форм сцепления (табл. 19), а также управляющих последовательностей из классов символов (табл. 20). В них могут присутствовать и обычные для Python управляющие последовательности символов (например, \t). Символьные строки шаблонов сопоставляются с текстовыми строками, давая логический результат совпадения, а сгруппированные подстроки указываются в круглых скобках для сопоставления с подшаблонами, как показано в приведенных ниже примерах.

```
>>> import re
>>> pobj = re.compile('hello[ \t]*(.*)')
>>> mobj = pobj.match('hello world!')
>>> mobj.group(1)
'world!'
```

В табл. 19 *C* обозначает любой символ; *R* — любую форму регулярного выражения в первом столбце таблицы; *m* и *n* — целочисленные значения. В каждой форме обычно употребляется как можно бо́льшая часть сопоставляемой символьной строки, за исключением непоглощающих форм, в которых употребляется как можно меньшая часть сопоставляемой символьной строки, при условии, что весь шаблон сопоставляется с целевой строкой.

Таблица 19. Синтаксис шаблонов регулярных выражений

Форма	Описание
`.`	Обозначает совпадение с любым символом, включая символ начала строки, если указан признак **DOTALL**
`^`	Обозначает совпадение с началом символьной строки (для каждой строки в режиме **MULTILINE**)
`$`	Обозначает совпадение с концом символьной строки (для каждой строки в режиме **MULTILINE**)
C	Обозначает совпадение с любым неспециальным символом
*R**	Обозначает отсутствие или как можно больше вхождений предыдущего регулярного выражения *R*
R+	Обозначает от одного и до как можно больше вхождений предыдущего регулярного выражения *R*
R?	Обозначает отсутствие или одно вхождение предыдущего регулярного выражения *R*
R{*m*}	Обозначает совпадение точно *m* раз с предыдущим регулярным выражением *R*
R{*m*,*n*}	Обозначает совпадение от *m* до *n* раз с предыдущим регулярным выражением *R*
*R**?, *R*+?, *R*??, *R*{*m*,*n*}?	То же, что и *, + и ?, но обозначает совпадение с как можно меньшим количеством символов или как можно меньше раз (*непоглощающая* форма)
[...]	Определяет набор символов; например, шаблон [a-zA-Z] обозначает совпадение со всеми символами (см. также табл. 20)
[^...]	Определяет дополняемый набор символов; обозначает совпадение, если символ отсутствует в наборе
\	Экранирует специальные символы (например, *, ?, +, |, ()) и вводит специальные последовательности символов (см. табл. 20). По принятым в Python правилам записывается как \\ или r'\\'
\\	Обозначает совпадение с литералом \; записывается как \\\\ в шаблоне или r'\\'
номер	Обозначает совпадение с содержимым группы одного и того же номера; так, шаблон r'(.+)\1' совпадет со строкой '42 42'
R|*R*	Обозначает совпадение с левым или с правым регулярным выражением
RR	Обозначает совпадение с обоими регулярными выражениями
(*R*)	Обозначает совпадение с любым регулярным выражением в круглых скобках, а также устанавливает границы *группы* (сохраняет совпавшую подстроку)
(?:*R*)	То же, что и форма (*R*), но не устанавливает границы *группы*
(?=*R*)	Обозначает упреждающее утверждение о совпадении, если происходит дальнейшее совпадение с регулярным выражением *R*, но символьная строка больше не употребляется (например, шаблон 'X(?=Y)' совпадает с X, если далее следует Y)

Форма	Описание
`(?!R)`	Обозначает отрицательное упреждающее утверждение о совпадении, если дальнейшее совпадение с регулярным выражением ***R не*** происходит. Противоположна форме `(?=R)`
`(?P<`***имя***`>R)`	Обозначает совпадение с любым регулярным выражением в круглых скобках и устанавливает границы группы (например, в шаблоне `r'(?P<id>[a-zA-Z_]\w*)'` определяется группа `id`
`(?P=`***имя***`)`	Обозначает совпадение с любым текстом, совпавшим с предыдущей группой ***имя***
`(?#...)`	Комментарий, который игнорируется
`(?`***буква***`)`	Устанавливает признак для всего регулярного выражения, где ***буква*** — одна из букв `a`, `i`, `L`, `m`, `s`, `x` или `u` (например, `re.A`, `re.I`, `re.L` и т.д.)
`(?<=R)`	Обозначает положительное упреждающее утверждение о совпадении, если имело место предыдущее совпадение с регулярным выражением ***R*** фиксированной длины
`(?<!R)`	Обозначает отрицательное упреждающее утверждение о совпадении, если ***не*** имело место предыдущее совпадение с регулярным выражением ***R*** фиксированной длины
`(?(`***id/имя***`)`***yespatt***`|`***nopatt***`)`	Предпринимает попытку сопоставления с шаблоном ***yespatt***, если группа ***id*** или ***имя*** существует, а иначе — с дополнительным шаблоном ***nopatt***

Специальные последовательности символов `\b`, `\B`, `\d`, `\D`, `\s`, `\S`, `\w` и `\W`, приведенные в табл. 20, действуют по-разному в зависимости от установленных признаков, и в Python 3.X они по умолчанию интерпретируются в уникоде, если только не установлен признак `ASCII` (т.е. `?a`). *Совет:* для буквальной интерпретации знаков обратной косой черты в последовательностях из классов символов, приведенных в табл. 20, следует использовать неформатированные символьные строки (например, `r'\n'`).

Таблица 20. Специальные последовательности для шаблонов регулярных выражений

Последовательность	Описание
`\`***номер***	Обозначает совпадение с текстом группы указанного ***номера***, начиная с `1`
`\A`	Обозначает совпадение только в начале символьной строки
`\b`	Обозначает пустую строку на границах слов

Последовательность	Описание
`\B`	Обозначает непустую строку на границах слов
`\d`	Обозначает любую десятичную цифру подобно шаблону `[0-9]`
`\D`	Обозначает любой недесятичный знак подобно шаблону `[^0-9]`
`\s`	Обозначает любой пробельный символ подобно шаблону `[\t\n\r\f\v]`
`\S`	Обозначает любой непробельный символ подобно шаблону `[^\t\n\r\f\v]`
`\w`	Обозначает любой буквенно-цифровой символ
`\W`	Обозначает любой небуквенно-цифровой символ
`\Z`	Обозначает совпадение только в конце символьной строки

Модули сохраняемости объектов

В стандартной библиотеке систему *сохраняемости объектов* образуют следующие три модуля:

`dbm` (или **`anydbm`** в версии Python 2.X)

Сохраняет в файлах только символьные строки по ключам.

`pickle` (и **`cPickle`** в версии Python 2.X)

Сериализирует объект, находящийся в оперативной памяти, когда он направляется в потоки файлового ввода-вывода.

`shelve`

Сохраняет объекты по ключам в файлах хранилищ типа `dbm` в форме, характерной для модуля `pickle`.

В модуле `shelve` реализуется механизм сохраняемости объектов в хранилищах. В свою очередь, модуль `pickle` используется в модуле `shelve` для преобразования (сериализации) объектов Python, находящихся в оперативной памяти, в линейные строки, а модуль `dbm` — для хранения этих строк в файлах с доступом по ключам. Все три модуля могут быть использованы непосредственно.

НА ЗАМЕТКУ

В версии Python 2.X модуль dbm называется anydbm, а модуль cPickle является оптимизированным вариантом модуля pickle, который может быть импортирован непосредственно и использован автоматически в модуле shelve, если таковой существует. А в версии Python 3.X модуль cPickle переименован в _pickle и автоматически используется в модуле pickle, если таковой существует. Его не нужно импортировать непосредственно, как это требуется в модуле shelve.

Следует также иметь в виду, что интерфейс Berkeley DB (т.е. bsddb) для модуля dbm больше не входит в состав самой версии Python 3.X, но является сторонним расширением с открытым кодом, которое должно быть установлено отдельно, если в этом возникнет необходимость. Подробнее об этом см. в руководстве по библиотеке для версии Python 3.X.

Модули shelve и dbm

Модуль dbm представляет собой простой *текстовый файл с доступом по ключам*. Символьные строки сохраняются и извлекаются из него по строковым ключам. Модуль dbm выбирает реализацию этого файла с доступом по ключам в интерпретаторе Python и предоставляет для сценариев прикладной программный интерфейс API словарного типа.

Модуль shelve представляет собой *объектный файл с доступом по ключам*. Он используется таким же образом, как и простой файл dbm, но только под другим именем модуля, а кроме того, в нем можно сохранять объекты любого типа, доступного в Python, хотя ключи доступа к нему по-прежнему остаются символьными строками. Во многих отношениях файлы dbm и shelve действуют подобно *словарям*, которые нужно открывать перед употреблением и закрывать после внесения изменений. К ним применимы все операции отображения и некоторые словарные методы, а сами они автоматически отображаются на внешние файлы.

Открытие файлов

Для того чтобы создать новый файл shelve или открыть уже существующий, следует импортировать соответствующую библиотеку и вызвать из нее функцию open(), как показано ниже.

```
import shelve
файл = shelve.open(имя_файла
          [, flag='c'
          [, protocol=None
          [, writeback=False]]])
```

А для того чтобы создать новый файл dbm или открыть уже существующий, следует импортировать соответствующую библиотеку и вызвать из нее файл open(), как показано ниже. Для этой цели служит любая библиотека, поддерживающая модуль dbm, в том числе dbm.bsd, dbm.gnu, dbm.ndbm или dbm.dumb (последняя используется по умолчанию в качестве резервной и всегда присутствует).

```
import dbm
файл = dbm.open(имя_файла
          [, flag='r'
          [, mode]])
```

В обоих модулях, shelve и dbm, символьная строка *имя_файла* обозначает относительное или абсолютное имя внешнего файла, в котором хранятся данные. Аргумент flag также одинаков для обоих модулей, причем модуль shelve передает его модулю dbm. Он может принимать строковое значение 'r' для открытия существующего файла базы данных только для чтения (по умолчанию в модуле dbm); строковое значение 'w' — для чтения и записи; строковое значение 'c' — для создания файла базы данных, если таковой отсутствует (по умолчанию в модуле shelve); а также строковое значение 'n' — для создания нового файла пустой базы данных. В модуле dbm.dumb, используемом в версии 3.X по умолчанию, если не установлена другая библиотека, аргумент flag игнорируется, а следовательно, файл базы данных всегда открывается для обновления или создается, если он отсутствует.

Дополнительный аргумент mode для модуля dbm обозначает режим доступа к файлу в Unix. Он указывается лишь в том случае, если база данных должна быть создана. По умолчанию он принимает восьмеричное значение **0o666**.

Аргумент protocol передается из модуля shelve в модуль pickle. Он предоставляет номер протокола сериализации для сохранения откладываемых на хранение объектов, как поясняется далее, в разделе "Модуль pickle". Этот аргумент по умолчанию равен **0** в версии Python 2.X и **3** в версии Python 3.X.

По умолчанию изменения в объектах, извлекаемых из хранилищ (так называемых "полок"), не записываются автоматически на диск. Если дополнительный аргумент writeback указан и принимает логическое значение True, то все доступные записи сначала кешируются в оперативной памяти, а затем записываются обратно на диск в момент закрытия хранилища. Благодаря этому упрощается внесение изменений в записи, изменяемые в хранилище, хотя и требует дополнительного объема оперативной памяти для кеширования, замедляя тем самым операцию закрытия хранилища, поскольку на диск записываются все доступные записи. Для обновления хранилищ вручную следует повторно присвоить объекты их ключам.

Операции с файлами

Созданные однажды файлы shelve и dbm имеют практически одинаковые интерфейсы словарного типа, как показано ниже.

```
файл[ключ] = значение
```

В ходе операции сохранения создается или изменяется запись, доступная по символьной строке *ключ*. А *значение* представлено символьной строкой для модуля dbm или произвольным объектом для модуля shelve.

```
значение = файл[ключ]
```

В ходе операции извлечения из записи, доступной по заданному *ключу*, загружается *значение*. Для модуля shelve исходный объект восстанавливается в оперативной памяти.

```
подсчет= len(файл)
```

В ходе операции определения размера возвращается количество хранящихся записей.

```
индекс = файл.keys()
```

В ходе операции индексирования извлекается итерируемый объект с хранящимися ключами (список в версии 2.X).

```
for ключ in файл: ...
```

В ходе итерации используется итератор ключей, подходящий для любого контекста итерации.

```
found = ключ in файл # а также метод has_key() только в
                     # версии 2.X
```

По запросу возвращается логическое значение `True`, если запись по указанному *ключу* отсутствует.

```
del файл[ключ]
```

В ходе операции удаления по заданному *ключу* удаляется запись.

```
файл.close()
```

В ходе операции закрытия файл закрывается вручную. Эта операция требуется для выгрузки обновлений на диск в некоторых базовых интерфейсах модуля `dbm`.

Модуль `pickle`

Модуль `pickle` служит для *сериализации объектов*, преобразуя практически каждый объект произвольного типа Python, находящийся в оперативной памяти и направляемый в байтовые потоки ввода-ввода. В эти байтовые потоки ввода-ввода может быть направлен любой файлоподобный объект, в котором предполагаются методы чтения и записи данных и которые используются в модуле `shelve` для внутреннего представления данных. В ходе десериализации исходный объект воссоздается в оперативной памяти с тем же самым значением, но в то же время новой идентичностью (адресом в оперативной памяти).

См. ранее примечание к модулю `cPickle` в версии Python 2.X и оптимизированному модулю `_pickle` в версии Python 3.X. См. также описание метода `makefile()`, доступного в объектах

из модуля socket для передачи сериализированных объектов по сети, не обращаясь к сокетам вручную, ниже, в разделе "Модули и средства доступа к Интернету" и в руководстве по Python.

Интерфейсы сериализации

В модуле pickle поддерживаются вызовы следующих интерфейсов:

```
P = pickle.Pickler(файловый_объект [, protocol=None])
```

Создает новый сериализатор, предназначенный для сохранения данных в объекте выходного файла.

```
P.dump(объект)
```

Выводит объект в файл или поток сериализатора.

```
pickle.dump(объект, файловый_объект [, protocol=None])
```

Представляет собой определенное сочетание двух предыдущих интерфейсов, сериализируя объект, выводимый в файл.

```
строка = pickle.dumps(объект [, protocol=None])
```

Возвращает сериализированное представление объекта в виде символьной строки (символьную строку типа bytes в версии Python 3.X).

Интерфейсы десериализации

В модуле pickle поддерживаются вызовы следующих интерфейсов:

```
U = pickle.Unpickler(файловый_объект,
          encoding="ASCII", errors="strict")
```

Создает десериализатор для загрузки данных из объекта входного файла.

```
объект = U.load()
```

Вводит объект из файла или потока десериализатора.

```
объект = pickle.load(файловый_объект,
                encoding="ASCII", errors="strict")
```

Представляет собой определенное сочетание двух предыдущих интерфейсов, десериализируя объект, вводимый из файла.

```
объект = pickle.loads(строка, encoding="ASCII",
                      errors="strict")
```

Вводит объект из символьной строки (типа `bytes` или совместимой строки в версии Python 3.X).

Примечания к применению модуля `pickle`

- В версии Python 3.X файлы, используемые для сохранения сериализированных объектов, всегда открываются в двоичном режиме работы всех протоколов, поскольку сериализатор производит символьные строки типа `bytes`, а файлы, работающие в текстовом режиме, не поддерживают запись символьных строк типа `bytes`, потому что в версии 3.X происходит кодирование и декодирование текста в уникоде.
- В версии Python 2.X файлы, используемые для сохранения сериализированных объектов, должны открываться в двоичном режиме для номеров всех протоколов сериализации, равных или больше **1**, чтобы подавлять преобразование символов конца строки в сериализированных двоичных данных. А протокол **0** основывается на коде ASCII, и поэтому его файлы могут открываться в текстовом или двоичном режиме, при условии, что это делается согласованно.
- Параметр *файловый_объект* обозначает открытый файловый или любой другой объект, реализующий атрибуты файлового объекта, обращение к которым происходит из интерфейса. Так, в классе `Pickler` вызывается метод `write()` с единственным аргументом для вывода данных в файл, а в классе `Unpickler` — метод `read()` с аргументом, обозначающим количество выводимых в файл байтов, а также метод `readline()` без аргументов.
- Аргумент `protocol` является необязательным и обозначает формат, выбираемый для сериализированных данных. Он доступен как в конструкторе класса `Pickler`, так в служебных функциях `dump()` и `dumps()` данного модуля. Этот аргумент принимает значение от **0** до **3**, обозначающее номер протокола (чем он больше, тем эффективнее

протокол), но этот номер может оказаться несовместимым с десериализаторами в прежних версиях Python. По умолчанию в версии Python 3 выбирается номер протокола **3**, не допускающий десериализацию в версии Python 2.X, где по умолчанию выбирается номер протокола **0**, который менее эффективен, но в основном переносим. Если же указан номер протокола **-1**, то автоматически выбирается протокол с наибольшим номером среди всех поддерживаемых. При *десериализации* протокол неявно выбирается исходя из содержимого сериализированных данных.

- Необязательные аргументы `encoding` и `errors` доступны как именованные для десериализации только в версии Python 3.X. Они служат для декодирования 8-разрядных строковых экземпляров, сериализируемых в версии Python 2.X. По умолчанию эти аргументы принимают строковые значения `'ASCII'` и `'strict'` соответственно. Дополнительно об аналогичных средствах см. описание функции `str()` выше, в разделе "Встроенные функции".
- Классы `Pickler` и `Unpickler` являются экспортируемыми и могут быть специально настроены путем подклассификации. О доступных в них методах см. в руководстве по библиотеке Python.

НА ЗАМЕТКУ

На момент написания этой книги существовало предложение (PEP) оптимизировать протокол десериализации номер **4** в версии Python 3.4, но оно все еще находилось на стадии чернового проекта, и поэтому было неясно, когда оно появится. Если это произойдет, то данное усовершенствование будет доступно в версии 3.X по умолчанию, но окажется несовместимым обратно и вообще не распознаваемым в прежних версиях Python. И как стало известно совсем недавно, это изменение официально вошло в версию 3.4, начиная с ее бета-выпуска. Подробнее об этом см. в разделе "What's New in Python" (Что нового в Python) документации на Python.

Модуль tkinter для построения ГПИ

Модуль tkinter, называемый Tkinter в версии Python 2.X и пакетом модулей в версии Python 3.X, представляет собой переносимую библиотеку, предназначенную для построения графического пользовательского интерфейса (ГПИ) и входящую в стандартный библиотечный модуль Python. Модуль tkinter предоставляет объектный интерфейс для библиотеки Tk с открытым кодом и реализует средствами Python внешний вид ГПИ, характерный для конкретной платформы: Windows, X-Window и Mac OS. Этот переносимый и вполне зрелый модуль прост в употреблении, неплохо документирован, широко применяется и хорошо поддерживается. К другим средствам построения ГПИ в Python относятся сторонние расширения *wxPython* и *PyQT* с более богатыми наборами виджетов, но, как правило, более сложными требованиями к программированию.

Пример применения модуля tkinter

В сценариях из модуля tkinter *виджеты* являются специализированными классами (например, Button, Frame), *параметры* — именованными аргументами (например, text= ="press"), а *композиция* — встраивание объектов, но не имен путей (например, Label(top, ...)), как показано в приведенном ниже примере.

```
from tkinter import *          # Виджеты и константы

def msg():                     # Обработчик обратных вызовов
    print('hello stdout...')

top = Frame()                  # Создание контейнера
top.pack()
Label(top, text='Hello world').pack(side=TOP)
widget = Button(top, text='press', command=msg)
widget.pack(side=BOTTOM)
top.mainloop()
```

Базовые виджеты в модуле tkinter

В табл. 21 перечислены основные классы виджетов в модуле tkinter. Эти классы Python подлежат подклассификации и встраиванию в другие объекты. Так, для создания устройства вывода на экран достаточно получить экземпляр соответствующего класса, настроить его и скомпоновать с помощью одного из методов из интерфейса диспетчера геометрических форм (например, метода `Button(text='hello').pack()`). Помимо классов, приведенных в табл. 21, в модуле tkinter предоставляется немало предопределенных имен (т.е. констант), предназначенных для настройки виджетов (например, RIGHT, BOTH, YES). Эти константы автоматически загружаются из модуля tkinter.constants (Tkconstants в версии Python 2.X).

Таблица 21. Классы базовых виджетов в модуле tkinter

Класс виджета	Описание
Label	Область вывода простых сообщений
Button	Простой виджет нажимаемой кнопки с меткой
Frame	Контейнер для присоединения и расположения объектов других виджетов
Toplevel, Tk	Окна верхнего уровня, управляемые диспетчером окон
Message	Многострочное поле вывода текста (метки)
Entry	Простое однострочное поле ввода текста
Checkbutton	Виджет кнопки с двумя состояниями, предназначенный для многовариантного выбора
Radiobutton	Виджет кнопки с двумя состояниями, предназначенный для одновариантного выбора
Scale	Виджет ползунка с масштабируемыми позициями
PhotoImage	Объект изображения, предназначенный для расположения многоцветных изображений в других виджетах
BitmapImage	Объект изображения, предназначенный для расположения растровых изображений в других виджетах
Menu	Пункты меню, связанные с виджетом **Menubutton** и окном верхнего уровня
Menubutton	Кнопка, раскрывающая виджет **Menu** с пунктами основного меню и подменю
Scrollbar	Полоса прокрутки других виджетов (например, **Listbox**, **Canvas**, **Text**)

Класс виджета	Описание
`Listbox`	Список выбираемых имен
`Text`	Многострочный виджет для ввода и правки текста с поддержкой его форматирования, включая шрифты и прочее
`Canvas`	Область рисования графики: линий, фигур, текста, изображений и прочего
`OptionMenu`	Раскрывающийся список
`PanedWindow`	Оконный интерфейс со многими панелями
`LabelFrame`	Виджет именованного фрейма
`Spinbox`	Виджет для многократного выбора
`ScrolledText`	Область текста с полосой прокрутки; под этим именем виджет доступен в версии Python 2.X, а под именем `tkinter.scrolledtext` — в версии Python 3.X
`Dialog`	Прежний типичный виджет для создания диалоговых окон, а новые виджеты данного типа приведены в следующем разделе; под этим именем виджет доступен в версии Python 2.X, а под именем `tkinter.dialog` — в версии Python 3.X

Типичные средства создания диалоговых окон

Модуль `tkinter.messagebox` (`tkMessageBox` в версии Python 2.X)

```
showinfo(title=None, message=None, **параметры)
showwarning(title=None, message=None, **параметры)
showerror(title=None, message=None, **параметры)
askquestion(title=None, message=None, **параметры)
askokcancel(title=None, message=None, **параметры)
askyesno(title=None, message=None, **параметры)
askretrycancel(title=None, message=None, **параметры)
```

Модуль `tkinter.simpledialog` (`tkSimpleDialog` в версии Python 2.X)

```
askinteger(заглавие, приглашение, **kw)
askfloat(заглавие, приглашение, **kw)
askstring(заглавие, приглашение, **kw)
```

Модуль `tkinter.colorchooser` (`tkColorChooser` в версии Python 2.X)

```
askcolor(color=None, **параметры)
```

Модуль `tkinter.filedialog` (`tkFileDialog` в версии Python 2.X)

```
class Open
class SaveAs
class Directory
askopenfilename(**параметры)
asksaveasfilename(**параметры)
askopenfile(mode="r", **параметры)
asksaveasfile(mode="w", **параметры)
askdirectory(**параметры)
```

В качества аргумента *параметры* в приведенных выше функциях указывает следующее: `defaultextension` (добавляется к имени файла, если указано явно), `filetypes` (последовательность кортежей `(label, pattern)`), `initialdir` (начальный каталог, запоминаемый классами), `initialfile` (начальный файл), `parent` (окно, в котором размещается диалоговое окно), а также `title` (заглавие диалогового окна).

Дополнительные классы и средства в модуле `tkinter`

В табл. 22 перечислены некоторые из дополнительных классов и средств, доступных в модуле `tkinter`, помимо классов основных виджетов и стандартных средств создания диалоговых окон. Все эти средства относятся к стандартной библиотеке Python, кроме тех, которые указаны в последней строке табл. 22. Подробнее о них можно узнать из документации на Python.

Таблица 22. Дополнительные средства в модуле `tkinter`

Категория	Доступные средства
Классы связанных переменных	Классы **`StringVar`**, **`IntVar`**, **`DoubleVar`**, **`BooleanVar`**
Методы управления геометрическими формами	Методы **`pack()`**, **`grid()`**, **`place()`** для объектов виджетов, а также средства настройки в модуле
Планируемые обратные вызовы	Методы **`after()`**, **`wait()`** и **`update()`** для виджетов, а также обратные вызовы в операциях файлового ввода-вывода

Категория	Доступные средства
Другие средства в модуле **tkinter**	Методы доступа к буферу обмена; методы обработки низкоуровневых событий для объектов виджетов; средства настройки виджетов вроде `config()`; средства поддержки модальных окон
Расширения модуля **tkinter**	*PMW*: дополнительные виджеты; *PIL* (или *Pillow*): изображения; древовидные виджеты; средства поддержки шрифтов и перетаскивания; виджеты **tix**, тематические виджеты **ttk** и т.д.

Сопоставление модуля tkinter с библиотекой Tk на языке Tcl

В табл. 23 прикладной программный интерфейс API модуля `tkinter` сравнивается с базовой библиотекой Tk, доступной на языке Tcl, на котором эта библиотека была первоначально написана. В общем, строки команд языка Tcl сопоставимы с объектами в языке Python. А в частности, ГПИ библиотеки Tk в модуле отличается от Tcl следующим.

- **Создание.** Виджеты создаются в виде экземпляров классов при обращении к классу виджета.
- **Родительские объекты.** Это созданные ранее объекты, переданные конструкторам классов виджетов.
- **Параметры виджетов.** Это именованные аргументы конструктора или функции `config()` или же индексированные ключи.
- **Операции.** Это действия над виджетами, ставшие методами объектов классов виджетов в модуле `tkinter`.
- **Обратные вызовы.** Обработчиком обратных вызовов может быть любой вызываемый объект: функция, метод, лямбда-выражение, класс с методом `__call__` и т.д.
- **Расширение.** Виджеты расширяются с помощью механизмов наследования классов в Python.
- **Композиция.** Построение интерфейсов происходит путем присоединения объектов, а не сцепления имен.
- **Связанные переменные.** Это переменные, связанные с виджетами и являющиеся объектами классов с соответствующими методами в модуле `tkinter`.

Таблица 23. Сопоставление модуля `tkinter` с библиотекой Tk

Операция	Tcl/Tk	Python/*tkinter*
Создание	`frame .panel`	`panel = Frame()`
Родительские объекты	`button .panel.quit`	`quit = Button(panel)`
Параметры	`button .panel.go -fg black`	`go = Button(panel, fg='black')`
Настройка	`.panel.go config -bg red`	`go.config(bg='red')` `go['bg'] = 'red'`
Действия	`.popup invoke`	`popup.invoke()`
Упаковка	`pack .panel -side left -fill x`	`panel.pack(side=LEFT, fill=X)`

Модули и средства доступа к Интернету

В этом разделе вкратце рассматриваются средства, поддерживающие *написание сценариев* доступа к Интернету в версиях Python 2.X и 3.X, в том числе наиболее употребительные модули из набора стандартных библиотечных модулей доступа к Интернету. Приведенный ниже материал не претендует на полноту, поэтому за дополнительными сведениями обращайтесь к руководству по библиотеке Python.

`socket`

Поддерживает низкоуровневую сетевую связь (по протоколу TCP/IP, UDP и т.д.). Предоставляет интерфейсы для передачи и приема данных через сокеты типа BSDstyle. В частности, функция `socket.socket()` создает объект с методами обращения к сокетам (например, *`объект`*`.bind()`). Этот модуль используется в большинстве протокольных и серверных модулей.

`socketserver` (**`SocketServer`** в версии Python 2.X)

Предоставляет каркас для общих многопоточных и разветвляющих сетевых серверов.

xdrlib

Кодирует двоичные данные с учетом их переносимости (дополнительно см. упомянутый выше модуль `socket`).

select

Предоставляет интерфейс для сопряжения с функцией `select()` в Unix и Windows. Ожидает активных действий над одним из файлов или сокетов. Обычно используется для разделения многих потоков ввода-вывода или реализации простоев. В Windows подходит только для сокетов, но не для файлов.

cgi

Поддерживает серверные сценарии CGI. В частности, функция `cgi.FieldStorage()` выполняет синтаксический анализ потока ввода, а функция `cgi.escape()` (и `html.escape()` в последних выпусках версии 3.X) применяет правила экранирования HTML-разметки в потоках вывода. Для синтаксического анализа и доступа к данным формы после сценария CGI делается вызов *форма*`=cgi.FieldStorage()`, где *форма* — объект словарного типа с одной записью на каждое поле формы (например, *форма*`["`*имя*`"].`*значение* — это текстовое поле *имя* в форме).

urllib.request (**urllib**, **urllib2** в версии 2.X)

Извлекает веб-страницы и результаты выполнения серверных сценариев по адресам в Интернете (URL). В частности, функция `urllib.request.urlopen(`*url*`)` возвращает файлоподобный объект с методами типа `read`, а функция `urllib.request.urlretrieve(`*remote*`, `*local*`)` извлекает данные из удаленного источника *remote* и копирует их в локальный файл *local*. Поддерживает сетевые протоколы HTTP, HTTPS, FTP и URL локальных файлов.

urllib.parse (**urlparse** в версии 2.X)

Выполняет по частям синтаксический анализ символьной строки с URL. Предоставляет также средства для экранирования текста URL. В частности, функция `urllib.parse`.

`quote_plus(`*строка*`)` экранирует текст, вставляемый в HTML-разметку, направляемую в потоки вывода.

`ftplib`

Предоставляет интерфейсы для передачи файлов через Интернет по сетевому протоколу FTP в программах на Python. После присваивания *`ftp`*`=ftplib.FTP('`*`имя_сайта`*`')` переменная *`ftp`* содержит объект с методами для регистрации, смены каталогов, извлечения и сохранения файлов, составления списков и прочих операций. Поддерживает передачу как двоичных, так и текстовых данных и подходит для любых машин с Python и доступом к Интернету.

`poplib`, **`imaplib`**, **`smtplib`**

Служат для поддержки сетевых протоколов POP, IMAP (получения почты) и SMTP (отправки почты).

пакет **`email`**

Выполняет синтаксический анализ и составление сообщений электронной почты с заголовками и вложениями. Предоставляет также поддержку типов MIME как для содержимого, так и для заголовков сообщений.

`http.client` (**`httplib`** в версии Python 2.X), **`nntplib`**, **`telnetlib`**

Служат для поддержки сетевых протоколов HTTP (веб), NNTP (новости) и Telnet на стороне клиента.

`http.server` (**`CGIHTTPServer`**, **`SimpleHTTPServer`** в версии 2.X)

Реализует запросы сервера по сетевому протоколу HTTP.

пакет **`xml`**, пакет **`html`** (**`htmllib`** в версии Python 2.X)

Выполняют синтаксический анализ XML- и HTML-документов и содержимого веб-страниц. Пакет `xml` поддерживает модели синтаксического анализа DOM, SAX и ElementTree с библиотекой Expat.

пакет **`xmlrpc`** (**`xmlrpclib`** в версии Python 2.X)

Поддерживает протокол XML-RPC вызова удаленных методов.

uu, binhex, base64, binascii, quopri

Кодирует и декодирует двоичные (и другие) данные, передаваемые в текстовом виде.

Некоторые из упомянутых выше модулей перечислены в табл. 24 по типу сетевого протокола. Отличия в их именах для версии 2.X приведены выше в круглых скобках.

Таблица 24. Перечень модулей для доступа к Интернету по типу сетевого протокола в версии Python 3.X

Протокол	Общее назначение	Номер порта	Модуль Python
HTTP	Веб-страницы	80	`http.client`, `urllib.request`, `xmlrpc.*`
NNTP	Новости Usenet	119	`nntplib`
FTP (по умолчанию для передачи данных)	Передача данных	20	`ftplib`, `urllib.request`
FTP (для управления)	Передача данных	21	`ftplib`, `urllib.request`
SMTP	Отправка электронной почты	25	`smtplib`
POP3	Получение электронной почты	110	`poplib`
IMAP4	Получение электронной почты	143	`imaplib`
Telnet	Режим командной строки	23	`telnetlib`

Другие стандартные библиотечные модули

В этом разделе описывается ряд дополнительных стандартных библиотечных модулей, устанавливаемых вместе с самим языком Python. Если не указано иное, то рассматриваемые здесь средства применимы в обеих версиях Python, 2.X и 3.X. Подробнее обо всех стандартных библиотечных модулях см. в руководстве по библиотеке Python, а о сторонних модулях и средствах — на веб-сайте PyPI, упомянутом в последнем разделе "Разные рекомендации" данной книги.

Модуль math

Экспортирует стандартные средства из библиотеки математических функций на C для применения в Python. В табл. 25 перечислены математические функции, экспортируемые этим модулем в версии Python 3.3 наряду с семью дополнительными функциями, внедренными в версиях 3.2 и 3.3 и выделенными полужирным. А в версии 2.7 этот модуль отличается лишь отсутствием функций `log2()` и `isfinite()`. Содержимое табл. 25 может немного изменяться для других версий Python. Все математические функции, перечисленные в табл. 25, кроме `pi` и `e`, являются вызываемыми, хотя они и указаны без завершающих круглых скобок.

Подробнее о модуле `math` см. в руководстве по библиотеке Python. С другой стороны, можно импортировать модуль `math` и вызвать функцию `help(math.`*имя*`)`, чтобы получить краткие сведения об аргументах интересующей функции и примечания к ее применению. А для ознакомления с содержимым модуля `math` достаточно вызвать функцию `dir(math)`. Дополнительно о средствах для выполнения операций с комплексными числами см. описание модуля `cmath`, а о других средствах для выполнения числовых операций — доступное в Интернете описание сторонних систем вроде *NumPy* и пр.

Таблица 25. Математические функции, экспортируемые модулем math

acos	acosh	asin	asinh	atan
atan2	atanh	ceil	copysign	cos
cosh	degrees	e	**erf**	**erfc**
exp	**expm1**	fabs	factorial	floor
fmod	frexp	fsum	**gamma**	hypot
isfinite	isinf	isnan	ldexp	**lgamma**
log	log10	log1p	**log2**	modf
pi	pow	radians	sin	sinh
sqrt	tan	tanh	trunc	

Модуль time

Содержит средства, связанные с доступом, форматированием и приостановкой отсчета *даты и времени*. Ниже перечислены

лишь некоторые функции даты и времени, экспортируемые модулем `time`. Дополнительно о подобных функциях см. ниже, в разделах "Модуль `datetime`" и "Модуль `timeit`", а также в руководстве по библиотеке Python.

`time.clock()`

Возвращает время ЦП или реальное время с момента запуска процесса или первого вызова функции `clock()`. Точность и семантика представления времени зависят от конкретной платформы (см. руководство по Python). Возвращаемое время выражается в секундах числом с плавающей точкой. Эта функция удобна для эталонных испытаний и определения времени выполнения альтернативных частей кода.

`time.time()`

Возвращает число с плавающей точкой, представляющее универсальное синхронизированное время (UTC) в секундах с момента начала эпохи. В Unix эпоха начинается с 1970 года. На некоторых платформах возвращаемое число может точнее обозначать время, чем то, что возвращает функция `clock()` (см. руководство по Python).

`time.ctime(`*`секунды`*`)`

Преобразует время, выражаемое в секундах с момента начала эпохи, в символьную строку, представляющую локальное время (например, `ctime(time())`). Аргумент *`секунды`* является необязательным, и если он опускается, то по умолчанию выбирается текущее время.

`time.sleep(`*`секунды`*`)`

Приостанавливает выполнение процесса (вызывающего потока исполнения) на заданное время в секундах. Аргумент *`секунды`* может принимать числовое значение с плавающей точкой для обозначения долей секунды.

Две следующие функции стали доступны *только* в версии Python 3.3. Они предназначены для предоставления временны*х* характеристик с учетом переносимости, но их нельзя непосредственно сравнивать с функциями из предыдущих версий Python.

Точка отсчета в значении, возвращаемом обеими функциями, не определена, поэтому единственное отличие результатов последовательных вызовов этих функций состоит в том, что они достоверны.

`time.perf_counter()`

Возвращает значение счетчика производительности в долях секунды, определяемое с максимально возможным разрешением для измерения коротких промежутков времени. Оно включает в себя время, прошедшее в состоянии ожидания, и рассчитывается на уровне системы. Может рассматриваться как фактическое время и использоваться по умолчанию в модуле `timeit`, если таковой имеется.

`time.process_time()`

Возвращает значение, выражающее в долях секунды сумму системного и пользовательского времени ЦП в текущем процессе. Не включает в себя время, прошедшее в состоянии ожидания, и по определению рассчитывается на уровне процесса.

Модуль `timeit`

Предоставляет средства для *измерения времени выполнения* прикладного кода или вызываемой функции с учетом переносимости. Подробнее об этом модуле см. в руководстве по Python.

Интерфейс командной строки:

```
py[thon] -m timeit [-n количество] [-r повтор]
        [-s установка]* [-t] [-c] [-p] [-h] [оператор]*
```

где *`количество`* обозначает, сколько раз выполняются операторы (по умолчанию вычисляется в степени **`10`**); *`повтор`* — количество прогонов (по умолчанию **`3`**); *`установка`* — код, выполняемый перед рассчитанными по времени операторами (указывается от нуля и больше с суффиксом **`-s`**); *`оператор`* — рассчитываемый по времени код (от нуля и больше); `-h` — вывод справки; **`t`**, **`-c`**, и **`-p`** — используемые таймеры, например, `time.time()`,

time.clock() или time.process_time(), начиная с версии Python 3.3, а по умолчанию — time.perf_counter(), начиная с версии 3.3. Выводимые результаты предоставляют наилучшее время среди всех выполненных *повторов*, что помогает нейтрализовать кратковременные колебания системной нагрузки.

Прикладной программный интерфейс API библиотеки:

timeit.Timer(stmt='pass', setup='pass', timer=dflt)

Используется в приведенных ниже служебных функциях. Оба параметра, stmt и setup, обозначают символьную строку кода (для разделения нескольких операторов следует использовать знак ; или последовательность символов \n, а для отступов — пробелы или последовательность символов \t) или же вызываемую функцию без аргументов. Устанавливаемое по умолчанию значение *dflt* параметра timer определяет функцию таймера в зависимости от конкретной платформы и версии Python.

timeit.repeat(stmt='pass', setup='pass',
timer=dflt, repeat=3, number=1000000)

Создает экземпляр класса Timer с заданным кодом stmt и setup и функцией timer и выполняет его метод repeat() с подсчетом повторов repeat и величиной number, обозначающей, сколько раз должен выполняться код. Возвращает список результатов вычисления временны*х* характеристик. Для получения наилучшего количества повторов repeat следует воспользоваться функцией min() из данного модуля.

timeit.timeit(stmt='pass', setup='pass',
timer=dflt, number=1000000)

Создает экземпляр класса Timer с заданным кодом stmt и setup и функцией timer и выполняет его метод timeit() с параметром number, обозначающим, сколько раз должен выполняться код. Выполняет код setup один раз и возвращает время, затраченное на выполнение кода stmt, столько раз, сколько указано в параметре number.

Модуль datetime

Предоставляет средства для *обработки дат*, в том числе для вычитания и сложения дней с датами и т.д. Подробнее об этих и других средствах обработки дат см. выше, в разделе "Модуль time", а также в руководстве по библиотеке Python. Ниже приведены примеры применения средств обработки дат из модуля datetime.

```
>>> from datetime import date, timedelta
>>> date(2013, 11, 15) - date(2013, 10, 29) # Промежуточная дата
datetime.timedelta(17)

>>> date(2013, 11, 15) + timedelta(60)    # Будущая дата
datetime.date(2014, 1, 14)
>>> date(2013, 11, 15) - timedelta(410    # Прошлая дата
datetime.date(2012, 10, 1)
```

Модуль random

Содержит избранные функции *рандомизации*, включая вычисление, перетасовку и выборку случайных чисел. Подробнее об этом модуле см. в руководстве по Python. Ниже приведены примеры применения средств из модуля random.

```
>>> import random
>>> random.random()          # Произвольное число с плавающей
                             # точкой в пределах [0, 1]
0.7082048489415967
>>> random.randint(1, 10)    # Произвольное целое число
                             # в пределах [x, y]
8
>>> L = [1, 2, 3, 4]
>>> random.shuffle(L)        # Непосредственная перетасовка
списка L
>>> L
[2, 1, 4, 3]
>>> random.choice(L)         # Выбор случайного элемента из
                             # списка
4
```

Модуль json

Содержит средства для взаимного преобразования структур словарей и списков в Python и текста в *переносимом формате представления данных* JSON, применяемом в таких системах, как MongoDB (где он называется BSON) и SL4A в Android (где он называется JSON-RPC). Дополнительно см. описание платформенно-зависимой сериализации объектов выше, в разделе "Модуль pickle"; поддержки XML — в разделе "Модули и средства доступа к Интернету"; а также принципов организации баз данных — ниже, в разделе "Прикладной интерфейс API базы данных SQL в Python". Ниже приведены примеры применения средств из модуля json.

```
>>> R = {'job': ['dev', 1.5], 'emp': {'who': 'Bob'}}

>>> import json
>>> json.dump(R, open('savejson.txt', 'w'))
>>> open('savejson.txt').read()
'{"emp": {"who": "Bob"}, "job": ["dev", 1.5]}'

>>> json.load(open('savejson.txt'))
{'emp': {'who': 'Bob'}, 'job': ['dev', 1.5]}

>>> R = dict(title='PyRef5E', pub='orm', year=2014)
>>> J = json.dumps(R, indent=4)
>>> P = json.loads(J)
>>> P
{'year': 2014, 'title': 'PyRef5E', 'pub': 'orm'}
>>> print(J)
{
    "year": 2014,
    "title": "PyRef5E",
    "pub": "orm"
}
```

Модуль subprocess

Содержит средства для выполнения процессов из *командной строки*, получения доступа к любому из трех стандартных потоков ввода-вывода, извлечения кодов завершения и указания средств, исполняемых из командной оболочки в качестве

альтернативы некоторым средствам из модуля os, в том числе функций s.popen() и os.spawnv(). Подробнее об этом см. выше, в разделе "Системный модуль os", а также в руководстве по Python. *Совет:* этими средствами не следует пользоваться для вычисления символьных строк ненадежного кода, поскольку они могут содержать любые команды, разрешенные для выполнения в процессе Python. Ниже приведен пример сценария *m.py* для вывода списка sys.argv аргументов командной строки.

```
>>> from subprocess import call, Popen, PIPE
>>> call('python m.py -x', shell=True)
['m.py', '-x']
0
>>> pipe = Popen('python m.py -x', stdout=PIPE)
>>> pipe.communicate()
(b"['m.py', '-x']\r\n", None)
>>> pipe.returncode
0
>>> pipe = Popen('python m.py -x', stdout=PIPE)
>>> pipe.stdout.read()
b"['m.py', '-x']\r\n"
>>> pipe.wait()
0
```

Модуль enum

Стал доступен в версии Python 3.4. Он предоставляет стандартную поддержку *перечислений*, устанавливая символические имена (т.е. *члены*), связанные с однозначными значениями констант. Этот модуль не следует путать с вызовом функции enumerate(), используемой для последовательного перечисления результатов выполнения числовых итераторов (см. выше раздел "Встроенные функции"). Ниже приведены примеры применения средств из модуля enum.

```
>>> from enum import Enum
>>> class PyBooks(Enum):
    Learning5E = 2013
    Programming4E = 2011
    PocketRef5E = 2014
>>> print(PyBooks.PocketRef5E)
```

```
PyBooks.PocketRef5E
>>> PyBooks.PocketRef5E.name,
    PyBooks.PocketRef5E.value
('PocketRef5E', 2014)
>>> type(PyBooks.PocketRef5E)
<enum 'PyBooks'>
>>> isinstance(PyBooks.PocketRef5E, PyBooks)
True
>>> for book in PyBooks: print(book)
...
PyBooks.Learning5E
PyBooks.Programming4E
PyBooks.PocketRef5E

>>> bks = Enum('Books', 'LP5E PP4E PR5E')
>>> list(bks)
[<Books.LP5E: 1>, <Books.PP4E: 2>, <Books.PR5E: 3>]
```

Модуль `struct`

Предоставляет интерфейс для синтаксического анализа и формирования *упакованных двоичных данных* в форматах, предназначенных для отражения форматов типа `struct` из языка C. Он нередко используется в сочетании с режимами `'rb'` и `'wb'` доступа к двоичным файлам в функции `open()` или другим источником двоичных данных. См. также описание типа данных `format` и кодов, определяющих порядок следования байтов, в руководстве по библиотеке Python. Ниже перечислены некоторые средства из модуля `struct`.

строка **= struct.pack(*формат*, *v1*, *v2*, ...)**

Возвращает символьную *строку* (типа `bytes` в версии 3.X и типа `str` в версии 2.X), содержащую значения *v1*, *v2* и т.д., упакованные в соответствии с заданной символьной строкой *формат*. Все эти аргументы должны точно соответствовать значениям, определяемым кодами типов в символьной строке *формат*. В первом символе этой строки может быть указан формат, определяющий порядок следования байтов в получаемом результате, а в остальной ее части — подсчет повторов для кодов отдельных типов.

***кортеж* = struct.unpack(*формат*, *строка*)**

Распаковывает *строку* (типа bytes в версии 3.X и типа str в версии 2.X) в *кортеж* значений объектов Python в соответствии с заданной символьной строкой *формат*.

struct.calcsize(*формат*)

Возвращает размер элемента данных типа struct, а следовательно, и символьную строку, состоящую из байтов, в заданном *формате*.

В приведенном ниже примере демонстрируется порядок упаковки и распаковки данных с помощью модуля struct в версии Python 3.X. А в версии Python 2.X для этой цели используются обычные символьные строки типа str, а не bytes. В выпуске 3.2 версии Python 3.X требуются строковые значения типа bytes, а не str. И наконец, символьная строка '4si' в приведенном ниже примере означает то же самое, что и выражение char[4]+int в C.

```
>>> import struct
>>> data = struct.pack('4si', b'spam', 123)
>>> data
b'spam{\x00\x00\x00'
>>> x, y = struct.unpack('4si', data)
>>> x, y
(b'spam', 123)
>>> open('data', 'wb').write(
            struct.pack('>if', 1, 2.0))
8
>>> open('data', 'rb').read()
b'\x00\x00\x00\x01@\x00\x00\x00'
>>> struct.unpack('>if', open('data', 'rb').read())
(1, 2.0)
```

Модули многопоточной обработки

Потоки исполнения являются упрощенными процессами, совместно использующими оперативную память (т.е. лексическими областями действия и внутренними ресурсами интерпретатора) и выполняющими функции *параллельно* в одном и том же процессе. Модули многопоточной обработки в Python действуют

с учетом переносимости на разные платформы. Они пригодны для выполнения неблокирующих задач в контекстах, связанных с вводом-выводом и пользовательским интерфейсом. Дополнительно см. описание функций `setcheckinterval()` и `setswitchinterval()` выше, в разделе "Модуль `sys`", а также стандартного библиотечного модуля `multiprocessing`, в котором реализуется потокоподобный прикладной программный интерфейс API для процессов, порождаемых с учетом переносимости.

`_thread` (называется **`thread`** в версии Python 2.X)

Это основной, низкоуровневый модуль поточной обработки, предоставляющий средства для запуска, остановки и синхронизации параллельно выполняющихся функций. В частности, для порождения потока исполнения служит функция `_thread.start_new_thread(`*`функция`*`, `*`args`*` [, `*`kargs`*`])`, выполняющая заданную *функцию* в новом потоке исполнения и принимающая позиционные аргументы из кортежа *args*, а также именованные аргументы из словаря *kargs*. Эта функция является синонимом функции `start_new()`, которая считается устаревшей в версии 3.X. Для синхронизации потоков исполнения следует применять блокировки потоков следующим образом:

```
lock=thread.allocate_lock();
lock.acquire();
обновление объектов;
lock.release()
```

`threading`

Этот модуль основывается на модуле `thread`, чтобы предоставлять следующие специализированные средства, ориентированные на многопоточную обработку: классы `Thread`, `Condition`, `Semaphore`, `Lock`, `Timer`, потоки типа демонов, соединения потоков и прочие. Для перегрузки метода-действия `run()` следует выполнить подклассификацию класса `Thread`. Этот модуль функционально более богатый, чем модуль `_thread`, но в более простых случаях он требует больше кода.

queue (называется **Queue** в версии Python 2.X)

Этот модуль предоставляет очередь обратного магазинного типа, содержащую объекты Python и поддерживающую многих поставщиков и потребителей. Он особенно удобен для многопоточных приложений, поскольку автоматически блокирует свои операции `get()` и `put()`, чтобы синхронизировать доступ к данным в очереди. Подробнее об этом модуле см. в руководстве по библиотеке Python.

Прикладной интерфейс API базы данных SQL в Python

Прикладной программный интерфейс API *реляционной базы данных* SQL предоставляет в Python средства переносимости сценариев среди пакетов баз данных SQL разных поставщиков. Для каждого поставщика следует установить отдельный модуль расширения, но сценарии следует писать с учетом переносимости прикладного программного интерфейса API базы данных. Сценарии для стандартной базы данных SQL будут в основном действовать и далее без изменений после переноса среди базовых пакетов разных поставщиков.

Следует, однако, иметь в виду, что большинство модулей расширения базы данных не являются частью стандартной библиотеки Python. Они являются *сторонними* компонентами, которые должны извлекаться и устанавливаться по отдельности. Исключением из этого правила служит пакет SQLite реляционной базы данных, встраиваемый в процесс и включенный в состав Python в виде *стандартного библиотечного* модуля `sqlite3`, предназначенного для хранения данных программы и создания прототипов.

Более простые альтернативные варианты сохранения данных см. выше, в разделе "Модули сохраняемости объектов". У сторонних поставщиков имеются и другие распространенные инструментальные средства базы данных, включая СУБД *MongoDB* для хранения документов в формате JSON; объектно-ориентированные

базы данных вроде *ZODB*; объектно-реляционные преобразователи, в том числе *SQLAlchemy* и *SQLObject*; а также прикладные программные интерфейсы API, ориентированные на облачные вычисления, например информационное хранилище *App Engine*.

Пример применения прикладного интерфейса API базы данных SQL

В приведенном ниже примере используется стандартный библиотечный модуль SQLite и ради краткости опускаются некоторые возвращаемые значения. Для применения баз данных масштаба предприятия вроде MySQL, PostgreSQL и Oracle и им подобных потребуются другие параметры соединения и установка модулей расширения, хотя могут поддерживаться и характерные для отдельных пользователей (и поэтому непереносимые) расширения SQL.

```
>>> from sqlite3 import connect
>>> conn = connect(r'C:\code\temp.db')
>>> curs = conn.cursor()

>>> curs.execute('create table emp (who, job, pay)')
>>> prefix = 'insert into emp values '
>>> curs.execute(prefix + "('Bob', 'dev', 100)")
>>> curs.execute(prefix + "('Sue', 'dev', 120)")

>>> curs.execute("select * from emp where pay > 100")
>>> for (who, job, pay) in curs.fetchall():
...     print(who, job, pay)
...
Sue dev 120
>>> result = curs.execute("select who, pay from emp")
>>> result.fetchall()
[('Bob', 100), ('Sue', 120)]
>>> query = "select * from emp where job = ?"
>>> curs.execute(query, ('dev',)).fetchall()
[('Bob', 'dev', 100), ('Sue', 'dev', 120)]
```

Интерфейсный модуль

В этом и последующих разделах описываются *лишь* некоторые средства, экспортируемые из рассматриваемого здесь прикладного интерфейса API для обращения с базой данных. Полное их описание можно найти по адресу `http://www.python.org`. Ниже перечислены самые основные средства из интерфейсного модуля *dbmod*.

***dbmod*.connect(*параметры*...)**

Конструктор объекта (*conn*), представляющего соединение с базой данных. Конкретные его *параметры* зависят от поставщика.

***dbmod*.paramstyle**

Символьная строка, предоставляющая вид форматирования маркера параметра (например, стиль `qmark = ?`).

***dbmod*.Warning**

Исключение, генерируемое для важных предупреждений вроде урезания данных.

***dbmod*.Error**

Исключение, которое служит базовым классом для всех остальных исключений, связанных с ошибками.

Объекты подключения к базе данных

Объекты подключения к базе данных (*conn*) реагируют на следующие методы:

***conn*.close()**

Разрывает соединение с базой данных немедленно, а не при вызове метода `__del__`.

***conn*.commit()**

Фиксирует любые ожидающие завершения транзакции с базой данных.

***conn*.rollback()**

Выполняет откат базы данных в начало любой ожидающей завершения транзакции; разрыв подключения к базе данных без предварительной фиксации изменений может привести к неявном откату.

***conn*.cursor()**

Возвращает новый объект курсора (*curs*) для передачи символьных строк с командами SQL через соединение с базой данных.

Объекты курсоров

Объекты курсоров (*curs*) представляют курсоры базы данных, используемые для управления контекстом операции извлечения данных. Эти объекты предоставляют следующие средства.

***curs*.description**

Последовательность, состоящая из семиэлементных последовательностей, каждая из которых содержит сведения, описывающие один столбец результатов: (*имя*, *код_типа*, *отображаемый_размер*, *внутренний_размер*, *точность*, *масштаб*, *пусто_готово*).

***curs*.rowcount**

Обозначает количество строк таблицы, которое было получено в результате выполнения последнего варианта команды `execute*` (для операторов DQL, подобных `select`) или на которое она воздействовала (для операторов DML вроде `update` или `insert`).

curs.callproc(*имя_процедуры* [, *параметры*])

Вызывает хранимую в базе данных процедуру по указанному имени. Последовательность *параметров* должна содержать одну запись на каждый аргумент, ожидаемый хранимой процедурой. Возвращаемый результат представляет собой видоизмененную копию вводимых данных.

***curs*.close()**

Закрывает курсор немедленно, а не при вызове метода __del__.

***curs*.execute(*операция* [, *параметры*])**

Подготавливает и выполняет операцию с базой данных (запрос или команду). В качестве *параметров* может быть указан список кортежей для ввода во многие строки таблицы в течение одной операции, хотя для этой цели предпочтительнее функция executemany().

***curs*.executemany(*операция*, *последовательность_параметров*)**

Подготавливает операцию с базой данных (запрос или команду) и выполняет ее по отношению ко всем последовательностям параметров или их соответствиям в *последовательности_параметров*. Действует аналогично нескольким вызовам функции execute().

***curs*.fetchone()**

Извлекает следующую строку таблицы из результата запроса, возвращая единственную последовательность или значение None, если данных больше нет. Этой функцией удобно пользоваться при обработке крупных массивов данных или медленной доставке данных.

***curs*.fetchmany([size=*curs*.arraysize])**

Извлекает следующий ряд строк таблицы из результата запроса, возвращая последовательность, состоящую из нескольких последовательностей (например, список кортежей). Пустая последовательность возвращается в том случае, если строк больше нет.

***curs*.fetchall()**

Извлекает все (или только оставшиеся) строки таблицы из результата запроса, возвращая их в виде последовательностей, состоящей из нескольких последовательностей (например, списка кортежей).

Объекты типов и конструкторы

`Date(год, месяц, день)`

Конструирует объект, содержащий значение даты.

`Time(час, минута, секунда)`

Конструирует объект, содержащий значение времени.

`None`

Пустые значения NULL языка SQL, представляемые значениями `None` в Python при вводе и выводе.

Дополнительные рекомендации и идиомы

В этом разделе вкратце рассматриваются типичные образцы программирования на языке Python и рекомендации по его применению, помимо изложенных ранее в данной книге. Дополнительные сведения по обсуждаемым здесь вопросам можно получить в руководстве по библиотеке и языку Python (`http://www.python.org/doc/`) и другим ресурсам, доступным в Интернете.

Общие рекомендации по языку

- Форма *S*`[:]` означает создание неполной (верхнего уровня) копии любой последовательности; вызов `copy.deepcopy(`*X*`)` — полной копии; вызовы `list(`*L*`)` и *D*`.copy()` — копий списков и словарей; а вызов *L*`.copy()` — копий списков, начиная с версии 3.3.
- Форма *L*`[:0]=`*итерируемый_объект* означает ввод нескольких элементов из *итерируемого_объекта* непосредственно в начало списка *L*.
- Все три формы, *L*`[len(`*L*`):]=`*итерируемый_объект*, *L*`.extend(`*итерируемый_объект*`)` и `L+=`*итерируемый_объект*, означают ввод нескольких элементов из *итерируемого_объекта* непосредственно в конец списка *L*.

- Формы *L*.append(*X*) и *X*=*L*.pop() могут быть использованы для непосредственной реализации операций со стеком, где *X* — размещаемые в стеке элементы, а конец списка *L* находится на вершине стека.
- Форма for *ключ* in *D*.keys(), или просто for *ключ* in *D*, начиная с версии 2.2, служит для перебора словарей. А в версии Python 3.X обе эти формы равнозначны, поскольку функция keys() возвращает итерируемое представление.
- Форма for *ключ* in sorted(*D*) служит для перебора ключей словаря в отсортированном порядке, начиная с версии 2.4; а форма *K*=D.keys(); *K*.sort(); for *ключ* in *K*: подходит для версии Python 2.X, но не для Python 3.X, поскольку в результате вызова функции keys() получаются объекты представлений, а не списки.
- В логическом выражении *X*=*A* or *B* or None переменной *X* присваивается первый истинный объект *A* или *B*, а иначе значение None, если оба объекта ложны (т.е. равны нулю или пусты).
- В выражении *X*, *Y* = *Y*, *X* значения *X* и *Y* меняются местами, не прибегая к явному промежуточному присваиванию.
- В выражении red, green, blue = range(3) ряд целочисленных значений присваивается в виде простого перечисления имен. Атрибуты классов и словари могут также присваиваться в виде перечислений. Начиная с версии Python 3.4, в стандартном библиотечном модуле enum обеспечивается расширенная функциональная поддержка перечислений.
- Пользуйтесь операторами try/finally для выполнения произвольного кода завершения. Это особенно удобно для блокируемых операций (например, для получения блокировки перед выполнением блока оператора try и снятия блокировки в блоке оператора finally).
- Пользуйтесь операторами with/as для выполнения кода завершения, характерного для объектов, поддерживающих только протокол диспетчера контекста (например, для

автоматического закрытия файлов или автоматического снятия блокировки с потока исполнения).

- Заключайте итерируемые объекты в оболочку вызываемой функции `list()` для представления результатов всех этих объектов в диалоговом режиме, а также для обеспечения правильности их многократного обхода. Эта рекомендация распространяется также на функции `range()`, `map()`, `zip()`, `filter()`, `dict.keys()` и прочие в версии Python 3.X.

Рекомендации по среде исполнения

- Пользуйтесь формой `if __name__ == '__main__':` для самопроверки прикладного кода или вызова главной функции в самом конце файлов модулей. Эта форма дает истинное значение только в том случае, если файл выполняется, а не импортируется в виде библиотечного компонента.
- Пользуйтесь формой `data=open(filename).read()` для загрузки содержимого файла в одном выражении. А для немедленного освобождения системных ресурсов могут потребоваться явные вызовы средств закрытия (например, в циклах), за исключением реализации CPython.
- Начиная с версии 2.2, пользуйтесь формой `for` *`строка`* `in` *`файл`* для перебора текстовых файлов по строкам. А в прежних версиях для этой цели подходит форма `for` *`строка`* `in` *`файл`*`.xreadlines()`.
- Пользуйтесь атрибутом `sys.argv` для извлечения аргументов командной строки.
- Пользуйтесь атрибутом `os.environ` для доступа к параметрам настройки среды исполнения командной оболочки.
- Пользуйтесь в качестве стандартных следующими потоками ввода-вывода: `sys.stdin`, `sys.stdout` и `sys.stderror`.
- Вызывайте функцию `glob.glob(`*`шаблон`*`)` для возврата списка файлов, совпадающих с заданным шаблоном.

- Вызывайте функцию `os.listdir(`*`путь`*`)` для возврата списка файлов и подкаталогов по заданному пути (например, `"."`).
- Вызывайте функцию `os.walk()` в обеих версиях Python, 2.X и 3.X, для обхода всего дерева каталогов. А функция `os.path.walk()` вызывается для этой цели только в версии Python 2.X.
- Вызывайте функцию `os.system(`*`командандная_строка`*`)` или пользуйтесь формой *`output`*`=os.popen(`*`командандная_строка`*`, 'r').read()` для выполнения команд из командной строки в сценариях Python. Последняя форма предполагает чтение из стандартного потока вывода порожденной программы и может быть также использована для выполнения операций построчного или чересстрочного чтения.
- Другие потоки ввода-вывода порожденной команды доступны через модуль `subprocess` в обеих версиях Python, 2.X и 3.X, тогда как разновидности функции `os.popen2/3/4()` доступны для вызова только в версии Python 2.X. Вызовы функций `os.fork()/os.exec*()` приводят к аналогичному результату на Unix-подобных платформах.
- Введите строку кода вроде `#!/usr/bin/env python` или `#!/usr/local/bin/python` в начале исходного файла и установите права доступа к нему по команде `chmod`, чтобы превратить этот файл в исполняемый сценарий на Unix-подобных платформах.
- Благодаря зарегистрированным сопоставлениям файлов в Windows их можно выполнять непосредственно щелчком на их именах. В версии 3.3 программа запуска Windows распознает также строки `#!`, составленные в стиле Unix (см. выше раздел "Запуск программ на Python в Windows").
- В функциях `print()` и `input()`, называемых также `print` и `raw_input()` в версии Python 2.X, используются стандартные потоки ввода-вывода `sys.stdin` и `sys.stdout`. Для внутренней переадресации ввода-вывода присвойте эти потоки ввода-вывода файлоподобным объектам, а

иначе воспользуйтесь формой `print(..., file=F)` в версии Python 3.X или формой `print >> F, ...` в версии Python 2.X.

- Установите кодировку `utf8` (или другую кодировку) в переменной окружения `PYTHONIOENCODING`, если сценарии не в состоянии выводить текст в уникоде, а не в коде ASCII, например имена файлов и их содержимое.

Рекомендации по применению

- Пользуйтесь формой `from __future__ import` *`имя_языкового_средства`* для активизации изменений в языке, которые ожидают официального одобрения и могут нарушить работоспособность существующего кода, хотя и допускают совместимость версий.
- Интуиция относительно производительности программ на Python зачастую подводит, поэтому старайтесь всегда измерить производительность своих программ, прежде чем оптимизировать их или переносить на C. Пользуйтесь для этой цели модулями `profile`, `time`, `timeit`, а также `cProfile`.
- Для применения инструментальных средств автоматизированного тестирования, входящих в состав стандартной библиотеки Python, обращайтесь к модулям `unittest` (или *PyUnit*) и `doctest`. Модуль `unittest` представляет собой сложный каркас классов, а модуль `doctest` просматривает символьные строки документации на наличие тестов и выводит результаты для повторного выполнения сеансов работы в диалоговом режиме.
- Функцией `dir([`*`объект`*`])` удобно пользоваться для проверки пространств имен атрибутов. А функция `print(`*`объект`*`.__doc__)` предоставляет неформатированные строки документации на заданный объект.
- Функция `help([`*`объект`*`])` предоставляет справку по модулям, функциям, типам данных, методам типов и прочим языковым средствам Python в диалоговом режиме. Так, в результате вызова `help(str)` предоставляется справка по

типу `str`; в результате вызова `help('модуль')` — справка по указанному модулю, даже если он не импортирован; в результате вызова `help('раздел')` — справка по ключевым словам и прочим разделам справки (для получения перечня разделов справки в качестве аргумента данной функции следует указать строку `'topics'`).

- Для извлечения и отображения символьных строк документации на модули, функции, классы и методы пользуйтесь командой `pydoc` из библиотечного модуля *PyDoc* и соответствующим сценарием в составе Python. В версии 3.2 по команде **`python -m pydoc -b`** запускается основанный на браузере интерфейс модуля PyDoc. А для работы с ГПИ в клиентском режиме достаточно указать параметр **`-g`** вместо параметра **`-b`** в упомянутой выше команде.
- Подробнее о запрете предупреждений, выдаваемых интерпретатором Python по поводу языковых средств, не рекомендуемых к употреблению в будущем, см. выше раздел "Каркас предупреждений", а также описание параметра командной строки **`-W`** в разделе "Параметры командной строки в Python".
- Подробнее о вариантах распространения программ на Python см. описание соответствующих утилит под общим названием *Distutils*, пакетов с расширением **`.egg`** (так называемых *eggs*) в руководстве по Python, а также приведенные ниже рекомендации.
- Подробнее об упаковке программ на Python в виде автономных исполняемых файлов (например, файлов с расширением **`.exe`** в Windows) см. описание сервисных программ *PyInstaller*, *py2exe*, *cx_freeze*, *py2app*.
- Подробнее о расширениях, превращающих язык Python в инструментальное средство для автоматизации числовых и научных расчетов с использованием векторных объектов, библиотек математических функций и прочих средств см. описание пакетов *NumPy*, *SciPy*, *Sage* и других связанных с ними пакетов. Обращайтесь также к стандартному библиотечному модулю `statistics` в версии Python 3.4.

- Подробнее о полноценной поддержке объектно-ориентированных баз данных для хранения объектов Python по ключам на конкретных платформах см. описание *ZODB* и других аналогичных баз данных, а об объектно-реляционных преобразователях, позволяющих использовать классы в таблицах реляционных баз данных, — описание *SQLObject*, *SQLAlchemy* и прочих аналогичных инструментальных средств. О возможностях базы данных на основе формата JSON, а не SQL см. описание системы *MongoDB*.
- Подробнее о разработке веб-приложений на Python см. описание *Django*, *App Engine*, *Web2py*, *Zope*, *Pylons*, *TurboGears* и прочих аналогичных сред веб-разработки.
- Подробнее об автоматическом генерировании связующего кода для применения библиотек на C и C++ в сценариях, написанных на Python, см. описание *SWIG* и прочих аналогичных инструментальных средств.
- Подробнее об ИСР с ГПИ, редакторами текста, выделяющими синтаксис разным цветом, браузерами объектов, отладчиками и прочими средствами разработки программ на Python см. описание ИСР *IDLE*, а также *PythonWin*, *Komodo*, *Eclipse*, *NetBeans*.
- Рекомендации по редактированию и выполнению разрабатываемого кода в текстовом редакторе *Emacs* см. в описании этого редактора. Операции с исходным текстом программ на Python, в том числе автоматическая расстановка отступов и выделение синтаксиса разным цветом, поддерживаются и в большинстве других редакторов, включая *VIM* и *IDLE*. Найти подходящие редакторы для исходного текста программ на Python можно по адресу `http://www.python.org`.
- При переносе программ в версию Python 3.X из версии Python 2.X следует указать параметр командной строки **`-3`**, чтобы получить предупреждения о несовместимости. Сценарий *2to3* автоматически преобразует большую часть исходного кода из версии 2.X, чтобы он мог выполняться в версии 3.X. См. также описание системы *six*,

предоставляющей требуемый уровень совместимости кода из версий 2.X и 3.X. Сценарий *3to2* автоматически преобразует бо́льшую часть исходного кода из версии 3.X, чтобы он мог выполняться в версии 2.X. Система *pies* также содействует совместимости исходного кода обеих версий Python.

Разные рекомендации

- За дополнительными сведениями по Python обращайтесь на следующие веб-сайты:

  `http://www.python.org`

  Начальная страница веб-сайта, посвященного языку Python.

  `http://www.python.org/pypi`

  Дополнительные инструментальные средства сторонних производителей для разработки программ на Python.

  `http://www.rmi.net/~lutz`

  Веб-сайт автора книги.
- Основной принцип Python: импорт нужных средств.
- В примерах исходного кода Python следует употреблять слова `spam` и `eggs` вместо традиционных слов `foo` и `bar`.
- Всегда обращайте внимание на светлую сторону жизни.

Предметный указатель

К

М

Н

О

П

С

Т

У

Ф

Ш

Я

PYTHON
СОЗДАНИЕ ПРИЛОЖЕНИЙ
БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТРЕТЬЕ ИЗДАНИЕ

Уэсли Чан

Книга содержит всю необходимую информацию для создания реальных приложений на языке Python. В ней описаны профессиональные приемы программирования на языке Python, методы создания клиентов и серверов с помощью протоколов TCP, UDP и XML-RPC, работа с высокоуровневыми библиотеками SocketServer и Twisted, изложены основы разработки GUI-приложений с помощью библиотеки Tkinter, продемонстрированы расширения на языке C/C++ и использование многопоточности, описана работа с реляционными базами данных, ORM и MongoDB, изложены основы веб-программирования, регулярных выражений, программирования COM-клиентов веб-разработки с помощью каркаса Django и облачных вычислений на платформе Google App Engine, а также программирование на языке Java в среде Jython и способы установления соединений с социальными сетями Twitter и Google+. Книга представляет собой ценный источник знаний по языку Python и предназначена для программистов всех уровней.

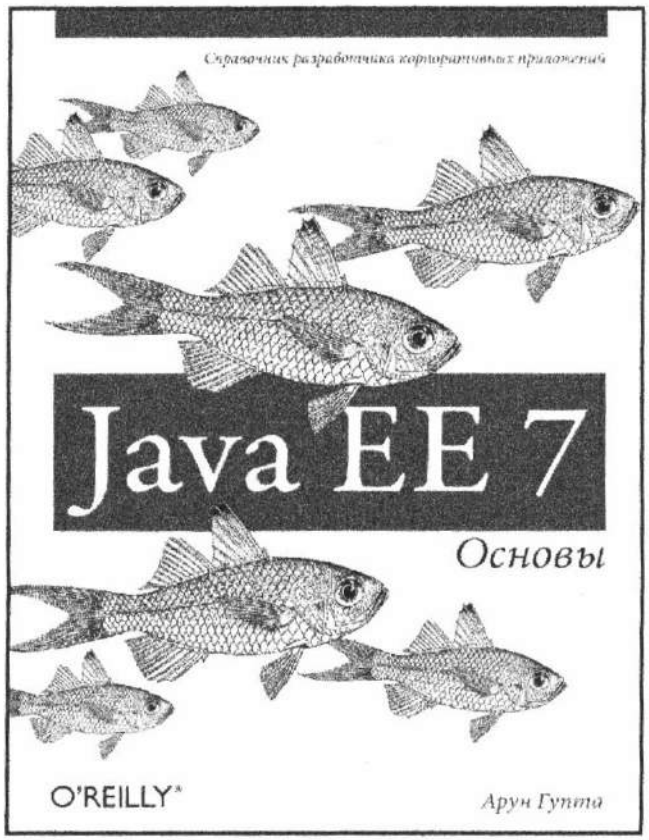
www.williamspublishing.com

ISBN 978-5-8459-1793-5

в продаже

Java EE 7
Основы

Арун Гупта



www.williamspublishing.com

Книга написана одним из ведущих разработчиков проекта Java EE, и каждая глава в ней посвящена рассмотрению одной из ключевых спецификаций платформы, включая WebSockets, Batch Processing, RESTful Web Services и Java Message Service.

- Ознакомьтесь с ключевыми компонентами платформы Java EE, руководствуясь многочисленными примерами в виде фрагментов кода, сопровождаемых подробными пояснениями автора.
- Изучите все новые технологии, которые были добавлены в версии Java EE 7, включая веб-сокеты, JSON, пакетную обработку и утилиты параллельного выполнения.
- Узнайте о применении веб-служб RESTful, служб на основе SOAP и службы сообщений Java (JMS).
- Изучите технологии Enterprise JavaBeans, CDI (Contexts and Dependency Injection) и Java Persistence.
- Узнайте о том, каким изменениям подверглись различные компоненты при переходе от Java EE 6 к Java EE 7.

ISBN 978-5-8459-1896-3 **в продаже**

O'REILLY®

Python Карманный справочник

Этот краткий справочник по Python карманного типа обновлен с учетом версий 3.4 и 2.7 и очень удобен для наведения быстрых справок в процессе разработки программ на Python. В лаконичной форме здесь представлены все необходимые сведения о типах данных и операторах Python, специальных методах, встроенных функциях и исключениях, наиболее употребительных стандартных библиотечных модулях и других примечательных языковых средствах Python.

Данное справочное пособие написано Марком Лутцом — известным и широко признанным во всем мире инструктором по Python. Оно послужит отличным дополнением к обширной литературе по Python, включая следующие книги самого автора: *Learning Python* (издательство O'Reilly), а также *Programming Python* (издательство O'Reilly).

В пятом издании этого справочника рассматриваются следующие вопросы

- Встроенные типы объектов, включая числа, списки, словари, множества и многое другое
- Операторы и синтаксис для создания и обработки объектов
- Функции и модули для структуризации и повторного использования кода
- Инструментальные средства объектно-ориентированного программирования на Python
- Встроенные функции, исключения и атрибуты
- Специальные методы перегрузки операторов
- Широко употребляемые стандартные библиотечные модули и расширения
- Параметры командной строки и инструментальные средства разработки
- Дополнительные рекомендации и идиомы
- Прикладной интерфейс API базы данных SQL в Python

ВИЛЬЯМС
www.williamspublishing.com

Категория: программирование
Предмет рассмотрения: язык Python
Уровень: промежуточный/опытный

ISBN 978-5-8459-1965-6
15044
9 785845 919656

www.oreilly.com