Г. Россум, Ф.Л.Дж. Дрейк, Д.С. Откидач

Язык программирования Python

Г. Россум, Ф.Л.Дж. Дрейк, Д.С. Откидач, М. Задка, М. Левис, С. Монтаро, Э.С. Реймонд, А.М. Кучлинг, М.-А. Лембург, К.-П. Йи, Д. Ксиллаг, Х.Г. Петрилли, Б.А. Варсав, Дж.К. Ахлстром, Дж. Роскинд, Н. Шеменор, С. Мулендер. **Язык программирования Python.** / 2001 — 454 с.

Python является простым и, в то же время, мощным интерпретируемым объектно-ориентированным языком программирования. Он предоставляет структуры данных высокого уровня, имеет изящный синтаксис и использует динамический контроль типов, что делает его идеальным языком для быстрого написания различных приложений, работающих на большинстве распространенных платформ. Книга содержит вводное руководство, которое может служить учебником для начинающих, и справочный материал с подробным описанием грамматики языка, встроенных возможностей и возможностей, предоставляемых модулями стандартной библиотеки. Описание охватывает наиболее распространенные версии Python: от 1.5.2 до 2.0.

© Stichting Mathematisch Centrum, 1990–1995
© Corporation for National Research Initiatives, 1995–2000
© BeOpen.com, 2000
© Д.С. Откидач, 2001

Дорогой читатель!

Вашему вниманию предлагается книга “Язык программирования Python”. Книга эта — не просто перевод английского учебника. Автор перевода проделал огромную работу по проверке примеров и упражнений, добавил в книгу немало других материалов.

Появление данной книги знаменует новый важный этап. Это признание того, что Python стал распространенным языком программирования, что его пользователи нуждаются в современной литературе на русском языке. В свою очередь появление книги будет способствовать еще большей популярности языка.

Python — это свободный интерпретируемый объектно-ориентированный расширяемый встраиваемый язык программирования очень высокого уровня.

- *свободный — все исходные тексты интерпретатора и библиотек доступны для любого, включая коммерческое, использования;*
- *интерпретируемый — потому что использует “позднее связывание”;*
- *объектно-ориентированный — классическая ОО модель, включая множественное наследование;*
- *расширяемый — имеет строго определенные API для создания модулей, типов и классов на C или C++;*
- *встраиваемый — имеет строго определенные API для встраивания интерпретатора в другие программы;*
- *очень высокого уровня — динамическая типизация, встроенные типы данных высокого уровня, классы, модули, механизм исключений.*

Python — язык универсальный, он широко используется во всем мире для самых разных целей — базы данных и обработка текстов, встраивание интерпретатора в игры, программирование GUI и быстрое создание прототипов (RAD). И, конечно же, Python используется для программирования Internet и Web приложений — серверных (CGI), клиентских (роботы), Web-серверов и серверов приложений. Python обладает богатой стандартной библиотекой, и еще более богатым набором модулей, написанных третьими лицами. Python и приложения, написанные на нем, используют самые известные и крупные фирмы — IBM, Yahoo!, Google.com, Hewlett Packard, Infoseek, NASA, Red Hat, CBS MarketWatch, Microsoft.

На этом языке написаны:

- *Mailman — менеджер списков рассылки (mailing list manager), ставший официальным менеджером списков рассылки проекта GNU;*
- *Medusa — архитектура для высокопроизводительных надежных TCP/IP серверов, таких как HTTP, FTP, NNTP, XML-RPC и SOAP;*
- *Zope — сервер Web-приложений (Web application server), приобретший широкую популярность.*

Python используется и в России. Многие компании используют его для внутренних нужд; на этом языке пишутся утилиты, фильтры, резидентные программы, GUI и Web-сайты. На некоторых сайтах все CGI-программы написаны на языке Python (сайт Фонда "Общественное мнение" www.fom.ru), другие используют системы публикации, написанные на языке Python (Русский Журнал, www.russ.ru). Находит использование и Zope. На нем сделаны сайты: Каталог Full.RU (www.full.ru), Банк МЕНАТЕП СПб (www.menatep.spb.ru), сайт красноярской компании Интербит (www.interbit.ru) и другие.

Существует русскоязычная группа пользователей Python и Zope, сайт которой (zope.net.ru) также построен на технологии Zope. Группа имеет список рассылки, в котором обсуждаются вопросы и решаются проблемы, связанные с использованием Python и Zope.

Олег Бройтман

Появление этой книги стало возможным благодаря всесторонней поддержке окружающих меня друзей. Особую признательность хотелось бы выразить моей жене Катерине — за редакторскую правку и проявленное терпение, Олегу Бройтману — за ценные замечания и прекрасное предисловие и всей русскоязычной группе пользователей Python и Zope — за полезные обсуждения приведенных здесь примеров. В списке рассылки группы Вы сможете высказать свои замечания и предложения, а также получить информацию, связанную с обновлением книги.

Денис Откидач

Оглавление

BEOPEN.COM TERMS AND CONDITIONS FOR PYTHON 2.0 13

I Вводное руководство 15

1 Разжигая Ваш аппетит 17

2 Использование интерпретатора 19
2.1 Вызов интерпретатора 19
2.1.1 Передача аргументов 20
2.1.2 Интерактивный режим 20
2.2 Интерпретатор и его среда 21
2.2.1 Обработка ошибок 21
2.2.2 Исполняемые файлы 21
2.2.3 Инициализация при запуске в интерактивном режиме 21

3 Неформальное введение в Python 23
3.1 Использование интерпретатора Python в качестве калькулятора 23
3.1.1 Числа 23
3.1.2 Строки 26
3.1.3 Строки Unicode 30
3.1.4 Списки 32
3.2 Первые шаги к программированию 34

4 Средства управления логикой 36
4.1 Инструкция if 36
4.2 Инструкция for 36
4.3 Функции range() и xrange() 37
4.4 Инструкции break и continue, ветвь else в циклах 39
4.5 Инструкция pass 39
4.6 Определение функций 40
4.7 Дополнительные возможности в определении функций 42
4.7.1 Значения аргументов по умолчанию 42
4.7.2 Произвольный набор аргументов 43
4.7.3 Именованные аргументы 43
4.7.4 Короткая форма 45
4.7.5 Строки документации 45
4.7.6 Вызов функций 46

5 Структуры данных 47
5.1 Подробнее о списках 47
5.1.1 Стеки 48
5.1.2 Очереди 48

5.2 Средства функционального программирования . . . 49
5.3 Дополнительные возможности при конструировании списков . . . 51
5.4 Инструкция `del` . . . 52
5.5 Кортежи . . . 53
5.6 Словари . . . 55
5.7 Подробнее об условиях . . . 56
5.8 Сравнение последовательностей . . . 57

6 Модули **58**
6.1 Создание и использование модулей . . . 58
6.2 Поиск модулей . . . 61
6.3 “Компилированные” файлы . . . 61
6.4 Стандартные модули . . . 62
6.5 Функция `dir()` . . . 63
6.6 Пакеты . . . 64
6.6.1 Импортирование всего содержимого пакета (модуля) . . . 66
6.6.2 Связи между модулями пакета . . . 67

7 Ввод/вывод **68**
7.1 Форматированный вывод . . . 68
7.2 Чтение и запись файлов . . . 72
7.2.1 Методы объектов-файлов . . . 73
7.2.2 Модуль `pickle` . . . 74

8 Ошибки и исключения **76**
8.1 Синтаксические ошибки . . . 76
8.2 Исключения . . . 76
8.3 Обработка исключений . . . 77
8.4 Генерация исключений . . . 80
8.5 Исключения, определяемые пользователем . . . 80
8.6 “Страхование” от ошибок . . . 80

9 Классы **82**
9.1 Несколько слов о терминологии . . . 82
9.2 Области видимости и пространства имен . . . 83
9.3 Первый взгляд на классы . . . 84
9.3.1 Синтаксис определения класса . . . 85
9.3.2 Объекты-классы . . . 85
9.3.3 Объекты-экземпляры . . . 86
9.3.4 Методы экземпляров классов . . . 87
9.4 Выборочные замечания . . . 88
9.5 Наследование . . . 90
9.6 Частные атрибуты . . . 91
9.7 Примеры использования классов . . . 92
9.7.1 Экземпляры классов в качестве исключений . . . 92
9.7.2 Классы-помощники . . . 94
9.7.3 Множества . . . 95
9.7.4 Контроль доступа к атрибутам . . . 98

II Встроенные возможности языка 99

10 Синтаксис и семантика 102
10.1 Структура строк программы . . . 102
10.1.1 Логические и физические строки . . . 102
10.1.2 Отступы . . . 103
10.2 Выражения . . . 104
10.2.1 Атомы . . . 105
10.2.2 Первичные выражения . . . 106
10.2.3 Арифметические и битовые операторы . . . 108
10.2.4 Условные операторы . . . 108
10.2.5 Истинность . . . 110
10.2.6 Логические операторы . . . 110
10.2.7 Оператор `lambda` . . . 111
10.2.8 Списки выражений . . . 111
10.2.9 Сводная таблица приоритетов . . . 112
10.3 Простые инструкции . . . 112
10.3.1 Инструкции-выражения . . . 113
10.3.2 Присваивание . . . 113
10.3.3 Инструкция `del` . . . 115
10.3.4 Пустая инструкция . . . 115
10.3.5 Инструкция `print` . . . 115
10.3.6 Инструкция `break` . . . 116
10.3.7 Инструкция `continue` . . . 116
10.3.8 Инструкция `return` . . . 117
10.3.9 Инструкция `global` . . . 117
10.3.10 Инструкция `import` . . . 117
10.3.11 Инструкция `exec` . . . 119
10.3.12 Отладочные утверждения . . . 120
10.3.13 Генерация исключений . . . 120
10.4 Составные инструкции . . . 121
10.4.1 Инструкция `if` . . . 122
10.4.2 Цикл `while` . . . 122
10.4.3 Цикл `for` . . . 123
10.4.4 Инструкция `try` . . . 124
10.4.5 Определение функций . . . 125
10.4.6 Определение класса . . . 128
10.5 Пространства имен . . . 128

11 Встроенные типы данных 131
11.1 Числовые типы . . . 131
11.1.1 Целые и длинные целые числа . . . 132
11.1.2 Вещественные числа . . . 132
11.1.3 Комплексные числа . . . 133
11.1.4 Арифметические операторы . . . 133
11.1.5 Битовые операции над целыми числами . . . 134
11.2 Последовательности . . . 135
11.2.1 Строки . . . 136

11.2.2 Строки Unicode . . . 143
11.2.3 Кортежи . . . 144
11.2.4 Объекты xrange . . . 144
11.2.5 Объекты buffer . . . 145
11.2.6 Изменяемые последовательности . . . 145
11.3 Отображения . . . 147
11.4 Объекты, поддерживающие вызов . . . 148
11.4.1 Функции, определенные пользователем . . . 149
11.4.2 Методы, определенные пользователем . . . 149
11.4.3 Встроенные функции и методы . . . 150
11.4.4 Классы . . . 151
11.4.5 Экземпляры классов . . . 151
11.5 Модули . . . 151
11.6 Классы и экземпляры классов . . . 152
11.6.1 Классы . . . 152
11.6.2 Экземпляры классов . . . 152
11.6.3 Специальные методы . . . 153
11.7 Файловые объекты . . . 163
11.8 Вспомогательные объекты . . . 165
11.8.1 Пустой объект . . . 165
11.8.2 Объекты типа . . . 165
11.8.3 Представление расширенной записи среза . . . 166
11.9 Детали реализации . . . 166
11.9.1 Объекты кода . . . 166
11.9.2 Кадр стека . . . 168
11.9.3 Объекты traceback . . . 169

12 Встроенные функции 170

13 Встроенные классы исключений 182

III Библиотека стандартных модулей 187

14 Конфигурационные модули 190
14.1 site — общая конфигурация . . . 190
14.2 user — конфигурация пользователя . . . 191

15 Служебные модули 192
15.1 sys — характерные для системы параметры и функции . . . 192
15.2 gc — управление “сборщиком мусора” . . . 198
15.3 atexit — выполнение действий при окончании работы программы . . . 200
15.4 types — имена для всех встроенных типов . . . 201
15.5 operator — операторы в виде функций . . . 204
15.6 traceback — модуль для работы с объектами traceback . . . 208
15.7 imp — доступ к операциям, производимым инструкцией import . . . 211
15.8 pprint — представление и вывод данных в более привлекательном виде 212
15.9 repr — альтернативная реализация функции repr() . . . 215

16 Работа со строками 217
16.1 string — наиболее распространенные операции над строками 217
16.2 re — операции с регулярными выражениями 220
16.2.1 Синтаксис регулярных выражений 220
16.2.2 Сопоставление в сравнении с поиском 225
16.2.3 Функции и константы, определенные в модуле 225
16.2.4 Объекты, представляющие регулярные выражения 228
16.2.5 Объекты, представляющие результат сопоставления 229
16.3 StringIO и cStringIO — работа со строками как с файловыми объектами 231
16.4 codecs — регистрация кодеров и работа с ними 232

17 Средства интернационализации 236
17.1 locale — использование национальных особенностей 236
17.2 gettext — выдача сообщений на родном языке 240
17.2.1 Интерфейс GNU gettext 240
17.2.2 Интерфейс, основанный на классах 241
17.2.3 Изготовление каталога переведенных сообщений 244

18 Математический аппарат 245
18.1 math — математические функции для работы с вещественными числами . 245
18.2 cmath — математические функции для работы с комплексными числами . 247
18.3 random — псевдослучайные числа с различными распределениями 249
18.4 whrandom — генератор псевдослучайных чисел 250
18.5 bisect — поддержание последовательностей в сортированном состоянии 251
18.6 array — эффективные массивы чисел 252

19 Интерфейсные классы к встроенным типам 255
19.1 UserString — интерфейсный класс для создания строковых объектов . 255
19.2 UserList — интерфейсный класс для создания последовательностей . . 256
19.3 UserDict — интерфейсный класс для создания отображений 257

20 Сохранение и копирование объектов 258
20.1 pickle и cPickle — представление объектов в виде последовательности байтов 258
20.2 shelve — сохранение объектов в базе данных в стиле DBM 262
20.3 marshal — байт-компилированное представление объектов 263
20.4 struct — преобразование объектов в структуры языка C 264

21 Доступ к средствам, предоставляемым операционной системой 266
21.1 os — основные службы операционной системы 267
21.1.1 Параметры процесса 267
21.1.2 Создание файловых объектов 269
21.1.3 Операции с файловыми дескрипторами 270
21.1.4 Файлы и каталоги 273
21.1.5 Управление процессами 276
21.1.6 Различная системная информация 279
21.2 os.path — работа с именами путей 280
21.3 stat — интерпретация os.stat() 283

21.4 statvfs — интерпретация os.statvfs() . . . 285
21.5 filecmp — сравнение файлов и каталогов . . . 286
21.6 popen2 — доступ к потокам ввода/вывода дочерних процессов . . . 288
21.7 time — определение и обработка времени . . . 289
21.8 sched — планирование задач . . . 294
21.9 getpass — запрос пароля и определение имени пользователя . . . 295
21.10 getopt — обработка опций в командной строке . . . 296
21.11 tempfile — создание временных файлов . . . 298
21.12 errno — символические имена стандартных системных ошибок . . . 299
21.13 glob — раскрытие шаблона имен путей . . . 302
21.14 fnmatch — сопоставление имен файлов с шаблоном . . . 303
21.15 shutil — операции над файлами высокого уровня . . . 304
21.16 signal — обработка асинхронных событий . . . 305
21.17 socket — сетевой интерфейс низкого уровня . . . 307
21.18 select — ожидание завершения ввода/вывода . . . 313
21.19 mmap — отображение файлов в память . . . 315

22 Средства организации многопоточных программ **318**

22.1 thread — создание нескольких потоков и управление ими . . . 318
22.2 threading — средства высокого уровня организации потоков . . . 320
22.2.1 Объекты, реализующие блокировку . . . 321
22.2.2 Условия . . . 321
22.2.3 Семафоры . . . 323
22.2.4 События . . . 323
22.2.5 Объекты, представляющие потоки . . . 324
22.3 Queue — синхронизированные очереди . . . 325

23 Работа с базами данных **327**

23.1 Интерфейс к базам данных в стиле DBM . . . 327
23.1.1 Общая для всех модулей часть интерфейса . . . 328
23.1.2 Дополнительные методы объектов, возвращаемых функцией dbhash.open() . . . 329
23.1.3 Дополнительные методы объектов, возвращаемых функцией gdbm.open() . . . 329
23.2 whichdb — определение формата файла базы данных . . . 330
23.3 bsddb — интерфейс к библиотеке баз данных BSD . . . 330

24 Сжатие данных **333**

24.1 zlib — алгоритм сжатия, совместимый с **gzip** . . . 333
24.2 gzip — работа с файлами, сжатыми программой **gzip** . . . 336
24.3 zipfile — работа с **zip**-архивами . . . 336

25 Отладка и оптимизация кода на языке Python **340**

25.1 Отладчик кода на языке Python . . . 340
25.1.1 Функции запуска отладчика . . . 341
25.1.2 Команды отладчика . . . 342
25.2 Замер производительности . . . 345
25.2.1 Введение . . . 345

25.2.2 profile — замер производительности . . . 347
25.2.3 pstats — обработка статистических данных и вывод отчетов . . . 348

26 Выполнение в защищенном режиме 350
26.1 rexec — основные средства настройки защищенного режима . . . 351
26.2 Bastion — ограничение доступа к экземплярам классов . . . 354

27 Поддержка протоколов Internet 355
27.1 cgi — протокол CGI . . . 355
27.1.1 Введение . . . 355
27.1.2 Использование модуля cgi . . . 356
27.1.3 Дополнительные возможности модуля . . . 359
27.1.4 Вопросы безопасности . . . 361
27.1.5 Установка CGI-программы . . . 361
27.1.6 Отладка . . . 362
27.2 urllib — чтение произвольных ресурсов по URL . . . 363
27.3 urlparse — операции над URL . . . 367

28 Поддержка форматов, используемых в Internet 369
28.1 rfc822 — обработка заголовков электронных писем . . . 369
28.2 mimetools — обработка сообщений в формате MIME . . . 373
28.3 MimeWriter — средства для записи в формате MIME . . . 374
28.4 multifile — чтение сообщений, состоящих из нескольких частей . . . 376
28.5 xdrlib — представление данных в формате XDR . . . 378

29 Средства работы с языками структурной разметки 383
29.1 sgmllib — обработка SGML-документов . . . 383
29.2 htmllib — обработка HTML-документов . . . 386
29.3 htmlentitydefs — определения сущностей HTML . . . 388
29.4 xml.parsers.expat — быстрая обработка XML-документов с помощью библиотеки Expat . . . 388
29.5 xml.sax — SAX2 интерфейс к синтаксическим анализаторам XML-документов . . . 392
29.6 xml.sax.handler — базовые классы для обработчиков SAX-событий . . . 394
29.6.1 Интерфейс класса ContentHandler . . . 395
29.6.2 Интерфейс класса DTDHandler . . . 397
29.6.3 Интерфейс класса ErrorHandler . . . 397
29.6.4 Интерфейс класса EntityResolver . . . 398
29.7 xml.sax.saxutils — вспомогательные средства для приложений, использующих SAX . . . 398
29.8 xml.sax.xmlreader — интерфейс объектов, реализующих чтение и синтаксический анализ XML-документов . . . 399
29.8.1 Интерфейс класса XMLReader . . . 400
29.8.2 Интерфейс класса IncrementalParser . . . 401
29.8.3 Интерфейс класса Locator . . . 402
29.8.4 Экземпляры класса InputSource . . . 402
29.8.5 Экземпляры классов AttributesImpl и AttributesNSImpl . . . 403

29.9 xmllib — обработка XML-документов 404

30 Разное 409
30.1 fileinput — перебор строк из нескольких входных потоков 409
30.2 ConfigParser — чтение конфигурационных файлов 412
30.3 shlex — простой синтаксический анализатор 415
30.4 cmd — создание командных интерпретаторов 417
30.5 calendar — функции для работы с календарем 419

Приложения 423

A Параметры командной строки интерпретатора и переменные окружения 425

B Грамматика языка 428

Указатель модулей 433

Предметный указатель 435

BEOPEN.COM TERMS AND CONDITIONS FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI OPEN SOURCE LICENSE AGREEMENT

Python 1.6 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1012. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1012.

CWI PERMISSIONS STATEMENT AND DISCLAIMER

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Часть I

Вводное руководство

Глава 1

Разжигая Ваш аппетит

Если Вы когда-либо писали большие shell-сценарии, Вам, возможно, знакомо чувство: хочется добавить еще какую-то возможность, но и без нее программа уже такая медленная, такая большая, или же Вам требуются системные вызовы или другие функции, доступные только в C... Обычно задача не настолько серьезная, чтобы переписывать все на C. Возможно, задача требует строк переменной длины или других типов данных (например, сортированные списки имен файлов), которые присутствуют в shell, но требуют значительных усилий для реализации на C, или же Вы не настолько знакомы с C.

Другая ситуация: Вам необходимо работать с несколькими библиотеками C, а на обычный цикл написание — компиляция — тестирование уходит слишком много времени. Вам нужно разрабатывать быстрее. Или же Вы уже написали программу, которая может использовать язык расширения, а у Вас нет желания его разрабатывать, писать и отлаживать для него интерпретатор, затем привязывать его к приложению.

В таком случае, Python — это то, что Вам нужно. Python прост в использовании, но это настоящий язык программирования, предоставляющий гораздо больше средств для структурирования и поддержки больших программ, чем shell. С другой стороны, он лучше обрабатывает ошибки, чем C и, будучи *языком очень высокого уровня*, имеет встроенные типы данных высокого уровня, такие как гибкие массивы и словари, эффективная реализация которых на C стоила бы Вам значительных затрат времени. Благодаря более общим типам данных, Python применим к более широкому кругу задач, чем Awk и даже Perl, в то время как многие вещи в языке Python делаются настолько же просто.

Python позволяет разбивать программы на модули, которые затем могут быть использованы в других программах. Python поставляется с большой библиотекой стандартных модулей, которые Вы можете использовать как основу для Ваших программ или в качестве примеров при изучении языка. Стандартные модули предоставляют средства для работы с файлами, системных вызовов, сетевые соединения и даже интерфейсы к различным графическим библиотекам.

Python — интерпретируемый язык, что позволит Вам сэкономить значительное количество времени, обычно расходуемого на компиляцию. Интерпретатор можно использовать интерактивно, что позволяет экспериментировать с возможностями языка, писать наброски программ или тестировать функции при разработке "снизу вверх". Он также удобен в качестве настольного калькулятора.

Python позволяет писать очень компактные и удобочитаемые программы. Программы, написанные на языке Python, обычно значительно короче эквивалента на C или

C++ по нескольким причинам:

- типы данных высокого уровня позволят Вам выразить сложные операции одной инструкцией;
- группирование инструкций выполняется с помощью отступов вместо фигурных скобок;
- нет необходимости в объявлении переменных.

Python расширяемый: знание C позволит Вам добавлять новые встраиваемые функции или модули для выполнения критичных операций с максимальной скоростью или написания интерфейса к коммерческим библиотекам, доступным только в двоичном виде. Вы можете вставить интерпретатор языка Python в приложение, написанное на C, и использовать его в качестве расширения или командного языка для этого приложения.

Кстати, язык назван в честь шоу BBC “Monty Python’s Flying Circus” и не имеет ничего общего с мерзкими рептилиями.

Теперь, когда у Вас появился интерес к языку Python, Вы захотите узнать о нем более подробно. Так как лучший способ изучить язык — использовать его, приглашаем Вас так и сделать. В следующей главе мы объясним Вам, как пользоваться интерпретатором. Это довольно скучная информация, но она необходима для того, чтобы Вы могли попробовать приведенные в книге примеры. Последующие главы ознакомят Вас с различными особенностями языка на примерах, начиная с простых выражений, инструкций и типов данных, через функции и модули, заканчивая такими концепциями, как исключения и классы.

Глава 2

Использование интерпретатора

2.1 Вызов интерпретатора

Если расположение исполняемого файла Python включено в пути поиска, то для его запуска достаточно набрать команду

```
python
```

Для выхода из интерпретатора необходимо набрать символ конца файла EOF (`Ctrl-D` в UNIX, `Ctrl-Z` в DOS и Windows) когда отображается первичное приглашение. Если это не работает, Вы можете набрать команду '`import sys; sys.exit()`'.

Интерпретатор ведет себя подобно UNIX shell: если его стандартный ввод соединен с терминалом — читает и исполняет команды интерактивно; если он вызывается с именем файла в качестве аргумента или стандартный ввод интерпретатора ассоциирован с файлом, он считывает и исполняет команды из этого файла.

Еще одним способом использования интерпретатора является вызов '`python -c` *command* [`arg ...`]'. В этом случае исполняются одна или несколько инструкций в команде *command*, аналогично использованию опции `-c` в UNIX shell. Так как инструкции в языке Python часто содержат пробелы, воспринимаемые как разделитель аргументов, а также другие специальные символы, лучше всего заключать *command* полностью в двойные кавычки.

Следует заметить, что есть разница между '`python` *file*' и '`python <` *file*'. В последнем случае, запросы на ввод, такие как `input()` и `raw_input()` будут удовлетворяться из файла. Так как файл уже будет прочитан, прежде чем программа начнет исполняться, программа немедленно получит EOF. В первом же случае (который Вам обычно и будет нужен), ввод осуществляется из устройства, с которым соединен стандартный ввод интерпретатора Python.

Иногда бывает полезно после исполнения инструкций из файла перейти в интерактивный режим. Это можно сделать, передав параметр `-i` перед именем файла. (Такой способ не сработает, если чтение производится из стандартного ввода, по той же причине, которая описана в предыдущем абзаце.)

Описание всех возможных параметров командной строки интерпретатора приведено в приложении A.

2.1.1 Передача аргументов

Имя исполняемого файла (программы) и дополнительные аргументы передаются программе в переменной `sys.argv`, которая является списком строк. Его длина (количество элементов в списке) всегда больше или равна единице. Имя программы хранится в `sys.argv[0]`. В интерактивном режиме `sys.argv[0]` содержит пустую строку. Если же имя программы передано как '-' (имея в виду стандартный ввод) или интерпретатор запущен с опцией `-c`, то значение `sys.argv[0]` устанавливается в `'-'` и `'-c'` соответственно. Все, что указывается после `-c` *command* не воспринимается как опции интерпретатором Python, а передается в `sys.argv` для обработки инструкциями в *command*.

2.1.2 Интерактивный режим

Когда команды считываются с терминала, говорят, что интерпретатор находится в *интерактивном режиме*. В этом режиме для ввода последующих команд выводится *первичное приглашение*, обычно три знака больше ('`>>> `'); для продолжения ввода незаконченных инструкций выводится *вторичное приглашение*, по умолчанию — три точки ('`... `'). При запуске в интерактивном режиме интерпретатор выводит приветственное сообщение — номер версии и замечания об авторском праве — перед выводом первичного приглашения, например:

```
$ python
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license" for more
information.
>>>
```

Продолжение ввода незаконченных инструкций необходимо при вводе многострочных конструкций. В качестве примера, взгляните на инструкцию `if`:

```
>>> the_world_is_flat = 1
>>> # Если земля плоская, вывести предупреждение
... if the_world_is_flat:
...     print "Осторожно, не свалитесь!"
...
Осторожно, не свалитесь!
```

2.2 Интерпретатор и его среда

2.2.1 Обработка ошибок

При возникновении ошибки интерпретатор печатает сообщение и остаток содержимого стека. В интерактивном режиме, после этого снова выдается первичное приглашение. Если программа читается из файла, интерпретатор печатает сообщение об ошибке, остаток содержимого стека и выходит с ненулевым кодом завершения. (Исключения, перехваченные ветвью `except` в инструкции `try`, не являются ошибками в данном контексте.) Некоторые ошибки — внутренние противоречия и некоторые случаи нехватки памяти — являются безусловно фатальными и приводят к выходу с ненулевым значением. Все сообщения об ошибках выводятся в стандартный поток ошибок; нормальные сообщения, возникающие в процессе выполнения команд, направляются в стандартный поток вывода.

Нажатие прерывающей комбинации клавиш (обычно Ctrl-C) во время выполнения генерирует исключение `KeyboardInterrupt`, которое может быть обработано с помощью инструкции `try`.

2.2.2 Исполняемые файлы

В операционных системах UNIX программу на языке Python можно сделать исполняемой непосредственно, поместив, например, строку

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

(подразумевая, что путь к интерпретатору включен в переменную окружения PATH пользователя) и установив разрешение на исполнение. Символы '`#!`' должны быть первыми двумя символами файла. Заметьте, что символ '#' в языке Python используется для обозначения комментария.

2.2.3 Инициализация при запуске в интерактивном режиме

Если Вы используете Python интерактивно, часто удобно иметь стандартный набор команд, исполняемых при каждом запуске интерпретатора. Для этого нужно присвоить переменной окружения PYTHONSTARTUP имя файла, содержащего команды инициализации (аналогично '.profile' для UNIX shell).

Указанный файл читается только в интерактивном режиме и не используется, если команды читаются из файла, через конвейер или если терминал явно указан в качестве источника. Инициализационный файл исполняется в том же пространстве имен, что и интерактивные команды, то есть определенные в нем объекты и импортированные модули могут быть использованы далее без каких-либо ограничений. В этом файле Вы также можете переопределить первичное (`sys.ps1`) и вторичное (`sys.ps2`) приглашения.

Если Вы хотите считывать дополнительно инициализационный файл из текущего каталога, следует включить соответствующие инструкции в глобальный инициализационный файл, например:

```
if os.path.isfile('.pythonrc.py'):
    execfile('.pythonrc.py')
```

Если необходимо использовать инициализационный файл в программе, Вы должны указать это явно:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

Глава 3

Неформальное введение в Python

В следующих примерах ввод и вывод различаются наличием или отсутствием приглашения (‘>>> ’ или ‘... ’): для воспроизведения примера Вам следует после появления приглашения набрать весь следующий за приглашением текст. Строки в примере, не начинающиеся с приглашения, выдаются самим интерпретатором. Обратите внимание, что наличие в строке только вторичного приглашения в примерах означает, что Вы должны ввести пустую строку — таким образом в интерактивном режиме обозначается конец многострочных команд.

Многие примеры в книге, даже если они вводятся в интерактивном режиме, снабжены комментариями. Комментарии в языке Python начинаются с символа ‘#’ и продолжаются до конца строки. Комментарий может начинаться в начале строки или после кода, но не внутри строковых выражений. Символ ‘#’ в строковом выражении является всего лишь символом ‘#’.

Несколько примеров:

```
# это первый комментарий
SPAM = 1                        # а это второй
                                # ... и, наконец, третий!
STRING = "# Это не комментарий."
```

3.1 Использование интерпретатора Python в качестве калькулятора

Давайте испробуем несколько простых команд Python. Запустите интерпретатор и дождитесь появления первичного приглашения ‘>>> ’ (это не должно занять много времени.)

3.1.1 Числа

Интерпретатор работает как простой калькулятор: Вы можете набрать выражение, и он выведет результат. Синтаксис выражений прост: операторы +, -, * и / работают, как и в большинстве других языков (например, в Pascal и C). Для группирования можно использовать скобки. Например:

```
>>> 2+2
4
>>> # Это комментарий
... 2+2
4
>>> 2+2  # Комментарий в той же строке, что и код
4
>>> (50-5*6)/4
5
>>> # При целочисленном делении результат округляется в
... # меньшую сторону:
... 7/3
2
>>> 7/-3
-3
```

Подобно C, знак равенства (‘=’) используется для присваивания значения переменной. Присвоенное значение при этом не выводится:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Значение можно присвоить одновременно нескольким переменным:

```
>>> x = y = z = 0 # Переменным x, y и z присваивается 0
>>> x
0
>>> y
0
>>> z
0
```

Имеется полная поддержка чисел с плавающей точкой. Операторы со смешанными типами операндов преобразуют целый операнд в число с плавающей точкой:

```
>>> 4 * 2.5 / 3.3
3.0303030303030303
>>> 7.0 / 2
3.5
```

Также поддерживаются и комплексные числа. Мнимая часть записывается с суффиксом ‘`j`’ или ‘`J`’. Комплексные числа записываются как ‘`(`*`real`*`+`*`imag`*`j)`’ или могут быть созданы функцией ‘`complex(`*`real`*`, `*`imag`*`)`’.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Комплексные числа представляются двумя числами с плавающей точкой — действительной и мнимой частью. Чтобы извлечь эти части из комплексного числа z, используйте `z.real` and `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Функции преобразования к целому числу и числу с плавающей точкой (`int()`, `long()` и `float()`) не работают для комплексных чисел — такое преобразование неоднозначно. Используйте `abs(z)` для получения абсолютного значения и `z.real` для получения вещественной части.

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g.
abs(z)
>>> a.real
1.5
>>> abs(a)
1.5811388300841898
```

В интерактивном режиме последнее выведенное значение сохраняется в переменной _. Это позволяет использовать Python в качестве настольного калькулятора, например:

```
>>> tax = 17.5 / 100
>>> price = 3.50
>>> price * tax
0.61249999999999993
>>> price + _
```

```
4.1124999999999998
>>> print round(_, 2)
4.11
```

Пользователь должен обращаться с ней как с переменной, доступной только для чтения. Не присваивайте ей значение явно — Вы создадите независимую локальную переменную с таким же именем, сделав встроенную переменную недоступной.

3.1.2 Строки

Кроме чисел, Python также может работать со строками (string), которые могут быть записаны различными путями[1]. Они могут быть заключены в одинарные или двойные кавычки:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Длинные строковые выражения могут быть разбиты различными способами на несколько строк. Символ новой строки может быть "спрятан" с помощью обратной косой черты ('\'), например:

```
hello = "Это длинное строковое выражение, содержащее\n\
несколько строк текста, как Вы бы это сделали в C.\n\
    Обратите внимание, что пробелы в\
 начале строки\nимеют значение.\n"
print hello
```

Результат будет следующим:

```
Это длинное строковое выражение, содержащее
несколько строк текста, как Вы бы это сделали в C.
```

[1]В настоящее время Python считает печатными 7-битное подмножество символов. Представление же остальных символов выводится с использованием управляющих последовательностей, что делает работу в интерактивном режиме, например, с русским языком неудобной. По этой причине текст в некоторых примерах данной книги оставлен без перевода.

```
     Обратите внимание, что пробелы в начале строки
имеют значение.
```

По-другому, текст может быть заключен в утроенные кавычки: `"""` или `'''`. Концы строк не нужно "прятать" при использовании утроенных кавычек, но они будут включены в текст.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

выведет следующее:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Интерпретатор выводит результат строковых операций в том же виде, в котором они могли бы быть набраны: в кавычках, обходя кавычки и другие специальные символы с помощью управляющих последовательностей, начинающихся с обратной косой черты ('\'). Текст заключается в двойные кавычки, если он не содержит двойные кавычки, во всех остальных случаях, он выводится в одинарных кавычках. Инструкция `print`, описанная ниже, может быть использована для вывода текста без кавычек и специальных последовательностей.

Существует также "необрабатываемый" режим ввода строк, задаваемый с помощью буквы '`r`' или '`R`' перед кавычками: в этом случае символ обратной косой черты также может быть использован для маскировки символов одинарной и двойной кавычек, если им предшествует нечетное число символов обратной косой черты, однако сам символ обратной косой черты остается частью строки. Даже в таком режиме строка не может заканчиваться нечетным количеством символов обратной косой черты. Необрабатываемый режим наиболее полезен в тех случаях, когда необходимо вводить значительное количество символов обратной косой черты, например, в регулярных выражениях.

Строки можно объединить (склеить) с помощью оператора + и размножить оператором *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Две строки, записанные друг за другом, автоматически объединяются. Первая строка в приведенном примере может быть также записана как '`word = 'Help' 'A'`'.

Такой метод работает только для строк записанных непосредственно, но не для произвольных строковых выражений.

```
>>> 'str' 'ing'              # Правильно
'string'
>>> 'str'.strip() + 'ing'   # Правильно
'string'
>>> 'str'.strip() 'ing'     # Ошибка
  File "<stdin>", line 1
    'str'.strip() 'ing'
                      ^
SyntaxError: invalid syntax
```

Строка — последовательность символов с произвольным доступом, Вы можете получить любой символ строки по его *индексу*. Подобно C, первый символ имеет индекс 0. Нет отдельного типа для символа, символ — это просто строка единичной длины. Подобно Icon, подстрока может быть определена с помощью *среза* — двух индексов, разделенных двоеточием.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Строки в языке Python невозможно изменить. Попытка изменить символ в определенной позиции или подстроку вызовет ошибку:

```
>>> word[0] = 'x'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:-1] = 'Splat'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Индексы среза имеют полезные значения по умолчанию: опущенный первый индекс считается равным 0, опущенный второй индекс дает такой же результат, как если бы он был равен длине строки.

```
>>> word[:2]  # Первые два символа
'He'
>>> word[2:]  # Вся строка, кроме первых двух символов
'lpA'
```

Полезный инвариант операции среза: `s[:i] + s[i:]` равно `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Срезы с вырожденными индексами обрабатываются изящно: слишком большой индекс обрабатывается, как если бы он был равен длине строки; если верхняя граница меньше нижней, то возвращается пустая строка.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Индексы могут иметь отрицательные значения, для отсчета с конца:

```
>>> word[-1]    # Последний символ
'A'
>>> word[-2]    # Предпоследний символ
'p'
>>> word[-2:]   # Последние два символа
'pA'
>>> word[:-2]   # Кроме последних двух символов
'Hel'
```

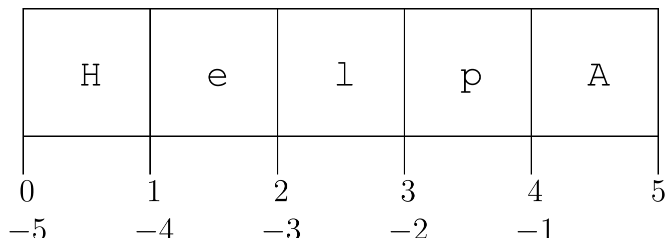
Однако `-0` обозначает то же самое, что и `0`, то есть не будет отсчитываться с конца.

```
>>> word[-0]      # (так как -0 равен 0)
'H'
```

Отрицательные индексы в срезах выходящие за пределы обрабатываются, как если бы они были равны нулю, но не пытайтесь использовать это для простых индексов (с одним элементом):

```
>>> word[-100:]
'HelpA'
>>> word[-10]     # Ошибка
Traceback (innermost last):
  File "<stdin>", line 1
IndexError: string index out of range
```

Лучший способ запомнить, как определяются индексы в срезе — считать их указывающими *между* символами, с номером 0 на левой границе первого символа. А правая граница последнего символа имеет индекс равный длине строки, например:



Первая строка чисел показывает позиции в строке, на которые указывают индексы от 0 до 5, вторая — соответствующие отрицательные индексы. Срез от *i* до *j* включает в себя все символы между краями, помеченными *i* и *j*, соответственно.

Для неотрицательных индексов длина подстроки равняется разности индексов, если они оба не выходят за пределы диапазона, например, длина подстроки *word*[1:3] равна 2.

Встроенная функция `len()` возвращает длину строки:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

3.1.3 Строки Unicode

Начиная с версии 1.6, в языке Python доступен новый тип данных для хранения текста — строка Unicode. Его можно использовать для работы с текстом, содержащим одновременно буквы и символы нескольких языков, доступных в Unicode (см. http://www.unicode.org). Строки Unicode полностью интегрируется с обычными строками, автоматически производя, где это необходимо, преобразование.

Unicode имеет значительное преимущество — предоставляет возможность использовать все символы, используемые в современных и древних текстах. Ранее мы могли использовать только 256 символов из определенной кодовой страницы, что приводило к большим затруднениям, особенно при интернационализации (internationalization, обычно записывается как i18n — i + 18 символов + n) программного обеспечения. Unicode решает эту проблему, определяя одну кодовую страницу для всех символов.

Создаются строки Unicode настолько же просто, как и обычные строки:

```
>>> u'Hello World !'
u'Hello World !'
```

Маленькая буква 'u' перед кавычками указывает, что предполагается создание строки Unicode. Если Вы хотите включить в строку специальные символы, используйте управляющие последовательности:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

Управляющая последовательность `\u0020` указывает, что необходимо вставить Unicode символ с порядковым номером в шестнадцатеричной системе исчисления 0x0020 (пробел).

Благодаря тому, что первые 256 символов Unicode те же, что и в стандартной кодировке Latin-1, ввод текста на большинстве языков, используемых в западных странах, сильно упрощается.

Как и для обычных строк, для строк Unicode существует "необрабатываемый" режим, задаваемый с помощью буквы 'r' или 'R' перед кавычками. Управляющими считаются только последовательности, которые применяются для обозначения символов Unicode, и только если используется нечетное количество символов обратной косой черты перед буквой 'u':

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

Кроме описанного выше метода, Python предоставляет возможность создать строку Unicode на основе строки в известной кодировке. Встроенная функция `unicode()` может работать с Latin-1, ASCII, UTF-8, UTF-16, с русскими кодировками ISO-8859-5, KOI8-R, CP1251, CP866 и Mac-cyrillic, и многими другими. Python по умолчанию использует кодировку ASCII[2], например, при выводе на экран инструкцией `print` и записи в файл. Если у Вас есть данные в определенной кодировке, для получения строки Unicode используйте встроенную функцию `unicode()`, указав кодировку в качестве второго аргумента:

```
>>> s = unicode("Привет", "KOI8-R")
>>> s
u'\u041F\u0440\u0438\u0432\u0435\u0442'
```

Если строка Unicode содержит символы с кодом больше 127, преобразование в ASCII не возможно:

```
>>> str(s)
Traceback (most recent call last):
```

[2]ASCII является общей частью для подавляющего большинства кодировок. Вы можете изменить кодировку по умолчанию с помощью функции `sys.set_string_encoding()`. Однако лучше все же указывать ее явно.

```
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in
range(128)
```

Метод encode() позволяет преобразовывать строки Unicode в обычные строки, содержащие текст в указанной кодировке:

```
>>> s.encode("KOI8-R")
'\360\322\311\327\305\324'
>>> s.encode("UTF-8")
'\320\237\321\200\320\270\320\262\320\265\321\202'
```

3.1.4 Списки

В Python имеется несколько типов данных, используемых для группирования вместе нескольких значений. Самым гибким является *список* (list), который может быть записан в виде списка значений (элементов), разделенных запятыми, заключенного в квадратные скобки. Совсем не обязательно, чтобы элементы списка были одного типа.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Как и для строк, для списков нумерация индексов начинается с нуля. Для списка можно получить срез, объединить несколько списков и так далее:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam',
'eggs', 100, 'Boe!']
```

В отличие от строк, которые *неизменяемы* (immutable), существует возможность изменения отдельных элементов списка:

```
>>> a
['spam', 'eggs', 100, 1234]
```

```
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Присваивание срезу также возможно, и это может привести к изменению размера списка:

```
>>> # Заменить несколько элементов:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # удалить:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Вставить:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Вставить копию самого себя в начало:
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy',
1234]
```

Встроенная функция `len()` также применима и к спискам:

```
>>> len(a)
8
```

Списки могут быть вложенными (списки, содержащие другие списки), например:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Как Вы уже, наверное, догадались, метод `append()` добавляет элемент в конец списка. Заметьте, что *p*`[1]` и *q* на самом деле ссылаются на один и тот же объект!

3.2 Первые шаги к программированию

Конечно, мы можем использовать Python для более сложных задач, чем “два плюс два”. Например, можно вывести начало ряда Фибоначчи:

```
>>> # Ряд Фибоначчи:
... # сумма двух предыдущих элементов определяет
... # следующий
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Этот пример знакомит с несколькими новыми особенностями.

- Первая строка содержит *множественное присваивание* (multiple assignment): переменным `a` и `b` одновременно присваиваются новые значения 0 и 1. В последней строке оно используется снова, демонстрируя, что выражения в правой части вычисляются до того как будет осуществлено присваивание. Выражения в правой части вычисляются слева направо.

- Цикл `while` выполняется пока условие (здесь: `b < 10`) является истинным. В Python, как и в C, любое ненулевое значение является истиной, ноль — ложь. В качестве условия может служить также строка, список — на самом деле любая последовательность. Последовательность с ненулевой длиной является истиной, пустая — ложью. Проверка, использованная в примере, является простым сравнением. Стандартные операторы сравнения записываются так же, как в C: <, >, ==, <= (меньше или равно), >= (больше или равно) и != (не равно).

- *Тело* цикла записано с *отступом*: отступы используются в языке Python для записи группирующих инструкций. Интерпретатор (пока) не предоставляет разумных средств редактирования вводимых строк, поэтому Вам необходимо нажимать клавишу табуляции или пробела для каждой строки с отступом. На практике Вы будете готовить более сложный ввод для Python с помощью текстового редактора, большинство из которых позволяют делать сдвиг автоматически. Когда составная инструкция вводится интерактивно, за ним должна следовать пуста строка, как признак завершения (поскольку синтаксический анализатор не может угадать, когда Вы ввели последнюю строку). Заметим, что сдвиг строк внутри блока должен быть одинаковым.

- Инструкция `print` выводит переданные ей значения. В отличие от простого вывода значений интерпретатором в интерактивном режиме (которым мы пользовались ранее в примерах использования интерпретатора в качестве калькулятора), инструкция `print` выводит строки без кавычек и между элементами вставляет пробел, так что Вы можете удобно форматировать вывод:

  ```
  >>> i = 256*256
  >>> print 'Значение переменной i равно', i
  Значение переменной i равно 65536
  ```

  Завершающая запятая позволяет после вывода значения вставлять пробел вместо перехода на новую строку:

  ```
  >>> a, b = 0, 1
  >>> while b < 1000:
  ...     print b,
  ...     a, b = b, a+b
  ...
  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
  ```

  Обратите внимание, что интерпретатор переходит на новую строку перед выводом следующего приглашения, даже если последняя выведенная строка не завершается переходом на новую строку.

Глава 4

Средства управления логикой

Кроме только что представленной инструкции `while`, в языке Python присутствуют другие привычные средства управления логикой программы с небольшими изменениями.

4.1 Инструкция `if`

Пожалуй, наиболее известной является инструкция `if`:

```
>>> x = int(raw_input("Введите, пожалуйста, число: "))
>>> if x < 0:
...      x = 0
...      print 'Отрицательное, меняем на ноль'
... elif x == 0:
...      print 'Ноль'
... elif x == 1:
...      print 'Один'
... else:
...      print 'Больше'
...
```

Ветвь `elif` может, как совсем отсутствовать, так и присутствовать несколько раз; наличие ветви `else` необязательно. Ключевое слово `elif` является короткой формой для `else if` и позволяет избежать чрезмерных отступов. Последовательность `if ... elif ... elif ...` эквивалентна инструкциям `switch` и `case` в других языках.

4.2 Инструкция `for`

Инструкция `for` в языке Python немного отличается от того, что используется в таких языках как C или Pascal. Вместо того, чтобы всегда перебирать числа арифметической прогрессии (как в Pascal), или предоставлять пользователю полную свободу выбора итератора и условия выхода из цикла (как в C), перебирает элементы произвольной[1]

[1] С формальной точки зрения это не совсем так: в языке Python под последовательностью всегда подразумевается последовательность с произвольным доступом; средства для работы с последовательностями

последовательности (например, списка или строки) в порядке их следования:

```
>>> # Измерение нескольких строк:
... a = ['кот', 'окно', 'выбросить']
>>> for x in a:
...     print x, len(x)
...
кот 3
окно 4
выбросить 9
```

Небезопасно изменять в цикле итерируемую последовательность (такое возможно только для последовательностей, допускающих изменение, например, списков). Если Вы собираетесь вносить изменения в список, элементы которого перебираете, например, продублировать избранные элементы, следует перебирать элементы копии исходного списка. Запись в виде среза делает это особенно удобным:

```
>>> for x in a[:]: # сделать копию (среза) всего списка
...     if len(x) > 4: a.insert(0, x)
...
>>> for x in a:
...     print x,
...
выбросить кот окно выбросить
```

Используя средства функционального программирования (см. раздел 5.2), можно одновременно перебирать элементы нескольких последовательностей.

4.3 Функции range() и xrange()

Если Вам необходимо перебирать последовательность чисел, то пригодится встроенная функция `range()`. Она создает список, содержащий арифметическую прогрессию:

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

Указанная верхняя граница никогда не входит в созданную последовательность. `range(10)` создает список из 10 значений, точно соответствующих допустимым индексам для элементов последовательности, имеющей длину 10. Можно указать другую нижнюю границу или другое приращение (шаг), в том числе и отрицательное:

(в том числе и инструкция `for`) требует возможности получить произвольный элемент по индексу.

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Для того, чтобы перебрать индексы последовательности, используйте совместно `range()` и `len()`:

```
>>> a = ['У', 'Марии', 'есть', 'маленькая', 'овечка']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 У
1 Марии
2 есть
3 маленькая
4 овечка
```

Дотошный читатель может заметить, что если нужно перебирать числа большого диапазона, создание списка будет неоправданно, а в некоторых случаях просто не хватит памяти:

```
>>> l=range(10000000)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
MemoryError
```

Действительно, если мы не собираемся изменять список, достаточно создать псевдосписок — объект, для которого мы можем получить значения "элементов", но не можем изменить их или порядок их следования. Для этих целей в языке Python предусмотрена функция `xrange()`:

```
>>> xrange(5, 10)
(5, 6, 7, 8, 9)
>>> xrange(0, 10, 3)
(0, 3, 6, 9)
>>> xrange(-10, -100, -30)
(-10, -40, -70)
>>> a = ['У', 'Марии', 'есть', 'маленькая', 'овечка']
>>> for i in xrange(len(a)):
...     print i, a[i]
...
0 У
1 Марии
```

```
2 есть
3 маленькая
4 овечка
```

4.4 Инструкции break и continue, ветвь else в циклах

Инструкция break, как и в C, выходит из самого внутреннего вложенного цикла for или while. Инструкция continue, также позаимствованная из C, продолжает выполнение цикла со следующей итерации.

Циклы могут иметь ветвь else, которая выполняется при “нормальном” выходе (исчерпание последовательности в цикле for, неудовлетворение условия в цикле while), без прерывания инструкцией break. Продемонстрируем ее использование на примере поиска простых чисел:

```
>>> for n in xrange(2, 10):
...     for x in xrange(2, n):
...         if n % x == 0:
...             print n, '=', x, '*', n/x
...             break
...     else:
...         print n, '- простое число'
...
2 - простое число
3 - простое число
4 = 2 * 2
5 - простое число
6 = 2 * 3
7 - простое число
8 = 2 * 4
9 = 3 * 3
```

4.5 Инструкция pass

Инструкция pass ничего не делает и может быть использована там, где инструкция требуется синтаксисом языка, однако действий никаких выполнять не требуется:

```
>>> while 1:
...     pass # Ожидание прерывания от клавиатуры
...
```

4.6 Определение функций

Мы можем создать функцию, которая будет выводить последовательность чисел Фибоначчи с произвольной верхней границей:

```
>>> def fib(n):
...     '''Выводит последовательность чисел Фибоначчи,
...     не превышающих n'''
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Теперь вызовем только что определенную функцию
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Ключевое слово def представляет *определение* функции. После него должно следовать имя функции и, в скобках, список формальных параметров. Инструкции, образующие тело функции, записываются с отступом, начиная со следующей строки. Первой инструкцией тела функции может быть *строка документации* (см. раздел 4.7.5). Существуют средства для автоматического создания документации, а также средства, позволяющие пользователю просматривать строки документации в интерактивном режиме. Включение строк документации в код является хорошей традицией.

Выполнение функции вводит новую таблицу имен, используемую для локальных переменных. Точнее, все присваивания переменным в теле функции сохраняются в локальной таблице имен. При ссылке на переменную, ее поиск производится сначала в локальной таблице имен, затем в глобальной и, в последнюю очередь, в таблице встроенных имен. Так, глобальным переменным нельзя прямо присвоить значение в теле функции (не упомянув их перед этим в инструкции global), хотя на них можно ссылаться.

Аргументы функции в момент вызова помещаются в локальную таблицу имен вызываемой функции. Таким образом, аргументы передаются *по значению* (где *значение* всегда является *ссылкой* на объект, а не его значением)[2]. В случае вызова функции другой функцией, также создается новая локальная таблица имен для этого вызова.

Определение функции вводит имя этой функции в текущую таблицу имен. Значение имени функции имеет тип, распознаваемый интерпретатором как определенная пользователем функция. Это значение может быть присвоено другому имени, которое затем также можно использовать в качестве функции (механизм переименования):

```
>>> fib
<function object at 10042ed0>
```

[2]На самом деле, правильнее было бы сказать, что аргументы передаются *по ссылке на объект*: если передается объект, допускающий изменения, то изменения (например, добавление элементов в список) будут видны в том месте, откуда функция была вызвана.

```
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Вы можете возразить, что *fib* является процедурой, а не функцией. В языке Python, как и в C, процедура — всего лишь функция, которая не возвращает никакого значения. Строго говоря, процедуры все же возвращают значение, которое, скорее, не представляет большого интереса. Это значение — `None` (встроенное имя). Значение `None` обычно не выводится интерпретатором, но Вы можете его увидеть, если действительно хотите этого:

```
>>> print fib(0)
None
```

Довольно просто написать функцию, которая возвращает список чисел ряда Фибоначчи, вместо того, чтобы выводить их:

```
>>> def fib2(n):
...     '''Возвращает список, содержащий числа ряда
...     Фибоначчи, не превышающие n'''
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)
...         a, b = b, a+b
...     return result
...
>>> # Вызываем только что определенную функцию
... f100 = fib2(100)
>>> # Выводим результат
... f100
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Этот пример, как обычно, демонстрирует несколько новых особенностей языка Python:

- Инструкция `return` выходит из функции и возвращает значение. Без аргументов `return` используется для выхода из середины процедуры (иначе выход происходит при достижении конца процедуры), в этом случае возвращается значение `None`.

- Инструкция *result*.append(*b*) вызывает *метод* объекта-списка *result*. Метод — это функция, "принадлежащая" объекту, вызывается как *obj.methodname*, где *obj* — объект (или выражение его определяющее) и *methodname* — имя метода, определенного для данного типа объектов. Различные типы имеют различные наборы методов. Методы различных типов могут иметь одинаковые имена, не приводя к неопределенности. (Вы можете определить собственные типы объектов —

классы — и методы работы с ними.) Метод `append()`, используемый в примере, определен для объектов-списков. Он добавляет элемент в конец списка. В приведенном примере его использование эквивалентно записи '`result = result + [b]`', но более эффективно.

4.7 Дополнительные возможности в определении функций

В языке Python можно определить функцию с переменным числом аргументов. Для этого существуют три способа, которые можно комбинировать.

4.7.1 Значения аргументов по умолчанию

Наиболее полезный способ — установить значения по умолчанию для одного или нескольких аргументов. Таким образом, получается функция, которую можно вызывать с меньшим количеством аргументов, чем в определении, например:

```
def ask_ok(prompt, retries=4,
           complaint='Да или нет, пожалуйста!'):
    while 1:
        ok = raw_input(prompt)
        if ok in ('д', 'да'): return 1
        if ok in ('н', 'не', 'нет'): return 0
        retries = retries - 1
        if retries < 0:
            raise IOError(
                  'Пользователь отказывается отвечать')
        print complaint
```

Эта функция может быть вызвана так: '`ask_ok('Вы действительно хотите выйти?')`', или так: '`ask_ok('Перезаписать файл?', 2)`'.

Значения по умолчанию вычисляются в месте определения функции в области видимости определения, так что

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

выведет `5`.

Важное замечание: значение по умолчанию вычисляется только один раз. Это отражается в том случае, когда аргумент со значением по умолчанию является объектом,

допускающим изменения, таким как список или словарь. Например, следующая функция накапливает передаваемые аргументы (то есть переменная `l` является статической)[3]:

```
def f(a, l = []):
    l.append(a)
    return l
print f(1)
print f(2)
print f(3)
```

Результат выполнения будет следующий:

```
[1]
[1, 2]
[1, 2, 3]
```

Если Вы не хотите, чтобы аргумент по умолчанию совместно использовался при последующих вызовах, можете немного изменить функцию:

```
def f(a, l = None):
    if l is None:
        l = []
    l.append(a)
    return l
```

4.7.2 Произвольный набор аргументов

Наконец, не так часто используемый вариант — определить функцию таким образом, чтобы ее можно было вызвать с произвольным числом аргументов. В этом случае аргументы будут переданы в виде кортежа[4]. Перед переменным числом аргументов может присутствовать произвольное число обычных аргументов:

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.3 Именованные аргументы

Функция может быть вызвана с использованием *именованных аргументов* (keyword arguments) в виде '*`keyword = value`*'. Например, функция `ask_ok()` (раздел 4.7.1) может быть вызвана следующими способами:

[3]Такое поведение является, скорее, нежелательной особенностью языка, использование которой может привести к трудно отлавливаемым ошибкам. В качестве статических переменных рекомендуется использовать глобальные переменные модуля.

[4]Кортеж — своего рода последовательность объектов, в которую Вы не можете вносить изменения.

```
ask_ok('Вы действительно хотите выйти?')
ask_ok(retries = 2, prompt = 'Перезаписать файл?')
ask_ok('Продолжим?', complaint =
       'А теперь то же самое, только по-русски!')
ask_ok('Удалить все файлы из корневого раздела?', 100,
       'А может все-таки удалить?')
```

Однако следующие вызовы ошибочны:

```
# отсутствует обязательный аргумент
ask_ok()

# неименованный аргумент следует за именованным
ask_ok(prompt = 'Перезаписать файл?', 2)

# два значения для одного аргумента
ask_ok('Продолжим?', prompt = 'Да/нет')

# неизвестное имя аргумента
ask_ok(actor = 'Никулин')
```

В общем, в списке аргументов именованные аргументы должны следовать после позиционных. Не имеет значение, имеет неименованный аргумент значение по умолчанию или нет. Никакой аргумент не может быть передан более одного раза:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Если в определении функции присутствует формальный параметр в виде `**name`, то его значением становится словарь, содержащий все именованные аргументы, чьи имена не соответствуют формальным параметрам. Такой способ можно комбинировать с формой `*name`, позволяющей получить кортеж (tuple), содержащий позиционные аргументы, не входящие в список формальных параметров. (Запись `**name` должна следовать после записи `*name`.) Например, определим функцию:

```
def example(formal, *arguments, **keywords):
    print "Количество пользователей:", formal
    print '-'*40
    for arg in arguments: print arg
    print '-'*40
    for kw in keywords.keys():
        print kw, ':', keywords[kw]
```

Ее можно вызвать следующим образом:

```
example(1000000,
    'Это очень много',
    'Это действительно очень много',
    language = 'Python',
    author = 'Guido van Rossum')
```

Вывод будет таким:

```
Количество пользователей: 1000000
----------------------------------------
Это очень много
Это действительно очень много
----------------------------------------
language: Python
author: Guido van Rossum
```

4.7.4 Короткая форма

Python содержит несколько популярных особенностей, характерных для языков функционального программирования. С помощью ключевого слова `lambda` Вы можете создать простую функцию без имени. Например, функция, возвращающая сумму двух своих аргументов, может быть записана как '`lambda` *a*, *b*: *a*+*b*'. Короткая форма может быть использована везде, где требуется объект-функция. Ее синтаксис ограничен одним выражением. Как и в обычных определениях функций, при записи в короткой форме Вы не можете ссылаться на переменные, находящиеся в области видимости, которая содержит определение этой функции. Однако это ограничение можно обойти, используя значения аргументов по умолчанию:

```
>>> def make_incrementor(n):
...     return lambda x, incr=n: x+incr
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

4.7.5 Строки документации

Строки документации обычно состоят из одной строки с кратким изложением назначения (или использования) объекта и более подробного описания, разделенных пустой

строкой. Первая строка часто используется интегрированными средами разработки (например, IDLE) в качестве подсказки при использовании объекта. Большинство встроенных объектов и объектов, определенных в стандартной библиотеке, снабжены строками документации — используйте их в качестве примеров оформления. Например:

```
>>> print map.__doc__
map(function, sequence[, sequence, ...]) -> list

Return a list of the results of applying the function
to the items of the argument sequence(s).  If more than
one sequence is given, the function is called with an
argument list consisting of the corresponding item of
each sequence, substituting None for missing values
when not all sequences have the same length.  If the
function is None, return a list of the items of the
sequence (or a list of tuples if more than one
sequence).
```

Интерпретатор языка Python не убирает отступы в многострочных строковых литеральных выражениях. Поэтому средства, обрабатывающие строки документации, должны это делать сами. Для определения уровня отступа всей строки документации используется первая непустая строка после первой строки (первая строка обычно записывается без отступа). Именно на столько следует уменьшить уровень отступа всей документации при выводе.

4.7.6 Вызов функций

Помимо описанного (`func(arg ...)`), язык Python предоставляет еще несколько способов вызова функций. Начиная с версии 1.6, Вы можете указать кортеж позиционных и словарь именованных аргументов, например:

```
args = ('Это очень много',
        'Это действительно очень много')
kwds = {'language': 'Python',
        'author': 'Guido van Rossum')
example(1000000, *args, **kwds)
```

Такой же эффект можно получить используя встроенную функцию `apply()`:

```
apply(example, (1000000,)+args, kwds)
```

С помощью средств функционального программирования Вы можете применить функцию к элементам последовательности (см. раздел 5.2).

Глава 5

Структуры данных

В этой главе описываются более детально ранее изученные возможности, а также некоторые новые особенности.

5.1 Подробнее о списках

Ранее мы уже говорили, что метод `append()` позволяет добавить элемент в конец списка:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> a.append(333)
>>> a
[66.6, 333, 333, 1, 1234.5, 333]
```

Однако иногда необходимо вставить элемент в начало или другую позицию списка. Это позволяет сделать метод `insert` — Вы указываете индекс элемента, перед которым новый элемент будет добавлен:

```
>>> a.insert(2, -1)
[66.6, 333, -1, 333, 1, 1234.5, 333]
```

Кроме того, для списков определены методы, позволяющие анализировать его содержимое: найти, в каком положении находится (первый) элемент с определенным значением (метод `index`), или подсчитать количество таких элементов (метод `count`):

```
>>> a.index(333)
1
>>> print a.count(333), a.count(66.6), a.count('x')
3 1 0
```

Метод `remove()` позволяет удалить из списка (первый) элемент, имеющий заданное значение:

```
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
```

Элементы списка можно отсортировать (метод `sort()`) и изменить порядок следования элементов на противоположный (метод `reverse()`):

```
>>> a.sort()  # сортируем по возрастанию
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> a.reverse()
>>> a
[1234.5, 333, 333, 66.6, 1, -1]
```

Более подробно эти и другие операции над списками описаны в разделе 11.2.6. Приведем лишь несколько примеров, показывающих насколько широка область их применения.

5.1.1 Стеки

Методы, определенные для списков, позволяют использовать список в качестве стека, где последний добавленный элемент извлекается первым (LIFO, “last-in, first-out”). Для добавления элемента в стек, используйте метод `append()`, для извлечения элемента с вершины стека — метод `pop()` без явного указания индекса:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Очереди

Еще одно применение списков — очереди, где элементы извлекаются в том же порядке, в котором они добавлялись (FIFO, “first-in, first-out”). Для добавления элемента в стек, используйте метод `append()`, для извлечения “очередного” элемента — метод `pop()` с индексом `0`:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")   # Terry добавлен в очередь
>>> queue.append("Graham")  # Graham добавлен в очередь
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

5.2 Средства функционального программирования

Существуют три встроенные функции, которые могут быть полезны при работе с последовательностями: `filter()`, `map()`, `zip()` и `reduce()`.

`filter(`*`function`*`, `*`sequence`*`)` возвращает последовательность (по возможности того же типа, что и *sequence*), состоящую из тех элементов последовательности *sequence*, для которых *function*`(`*`item`*`)` является истиной. Например, выделим простые числа из списка:

```
>>> def f(x):
...     for y in xrange(2, x):
...         if x%y==0: return 0
...     return 1
...
>>> filter(f, xrange(2, 40))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

`map(`*`function`*`, `*`sequence`*` [...])` возвращает список значений, полученных применением функции *function* к элементам одной или нескольких последовательностей. Например, создадим список кубов натуральных чисел:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, xrange(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Функция *function* должна воспринимать столько аргументов, сколько последовательностей передается. Она вызывается с соответствующими элементами из каждой последовательности или `None`, если какая-нибудь последовательность оказалась короче других. Если *function* равно `None`, то используется функция, возвращающая свои аргументы.

Учитывая эти две особенности, мы видим, что '`map(None, `*`list1`*`, `*`list2`*`)`' является удобным способом превращения пары списков в список пар:

```
>>> seq = xrange(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25),
(6, 36), (7, 49)]
```

Таким образом Вы можете перебирать элементы одновременно несколько последовательностей одинаковой длины:

```
>>> seq1 = ['cat', 'mouse', 'bird']
>>> seq2 = ['кот', 'мышь', 'птица']
>>> for x, y in map(None, seq1, seq2):
...     print x, y
...
cat кот
mouse мышь
bird птица
```

Как мы уже говорили, если какая-либо последовательность оказывается короче других, функция `map()` дополняет ее элементами, равными `None`. Такое поведение не всегда желательно. Поэтому в версии 2.0 была добавлена функция `zip()`. `zip(`*sequence* `[...]`) возвращает список кортежей, каждый из которых состоит из соответствующих элементов аргументов-последовательностей. При этом длина полученной последовательности будет равна длине самой короткой последовательности среди аргументов. Например:

```
>>> a = (1, 2, 3, 4)
>>> b = (5, 6, 7, 8)
>>> c = (9, 10, 11)
>>> d = (12, 13)
>>> zip(a, b)
[(1, 5), (2, 6), (3, 7), (4, 8)]
>>> zip(a, d)
[(1, 12), (2, 13)]
>>> zip(a, b, c, d)
[(1, 5, 9, 12), (2, 6, 10, 13)]
```

Если последовательности имеют одинаковую длину, поведение функции `zip()` полностью аналогично поведению `map()` с `None` в качестве первого аргумента. Кроме того, в этом случае действие функций `zip()` и `map()` обратимо:

```
>>> a = (1, 2, 3)
>>> b = (4, 5, 6)
>>> x = zip(a, b)
>>> y = zip(*x) # или apply(zip, x)
```

```
>>> z = zip(*y) # или apply(zip, y)
>>> x
[(1, 4), (2, 5), (3, 6)]
>>> y
[(1, 2, 3), (4, 5, 6)]
>>> z
[(1, 4), (2, 5), (3, 6)]
>>> x == z
1
```

`reduce(`*`function`*`, `*`sequence`* [`, `*`initial`*]) возвращает значение, полученное путем последовательного применения бинарной функции *`function`* сначала к первым двум элементам последовательности *`sequence`*, затем к результату и следующему элементу и т. д. Например, вычислим сумму арифметической последовательности:

```
>>> def add(x, y): return x+y
...
>>> reduce(add, xrange(1, 11))
55
```

Если последовательность содержит единственный элемент, возвращается его значение, если же последовательность пуста, то генерируется исключение `TypeError`.

В качестве третьего аргумента можно указать начальное значение. В этом случае оно возвращается для пустой последовательности, и функция сначала применяется к начальному значению и первому элементу последовательности, затем к результату и следующему элементу и т. д.:

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(xrange(1, 11))
55
>>> sum([])
0
```

5.3 Дополнительные возможности при конструировании списков

Начиная с версии 2.0, языка Python, существуют дополнительные возможности конструирования списков, не прибегая к средствам функционального программирования. Такие определения списков записываются с использованием в квадратных скобках выражения и следующих за ним блоков `for`:

```
>>> freshfruit = ['  banana',
...               '  loganberry ',
...               'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Элементы можно фильтровать, указав условие, записываемое с помощью ключевого слова `if`:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

Выражение, дающее кортеж, обязательно должно быть записано в скобках:

```
>>> [x, x**2 for x in vec]
  File "<stdin>", line 1
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
```

Если в конструкторе указано несколько блоков `for`, элементы второй последовательности перебираются для каждого элемента первой и т. д., то есть перебираются все комбинации:

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

5.4 Инструкция `del`

Существует способ удалить не только элемент по его значению, но и элемент с определенным индексом, элементы с индексами в определенном диапазоне (ранее мы произво-

дили эту операцию путем присваивания пустого списка срезу): инструкция `del`:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

Инструкция `del` может быть также использована для удаления переменных:

```
>>> del a
```

После этого ссылка на переменную *a* будет генерировать исключение `NameError`, пока ей не будет присвоено другое значение. Позже мы расскажем о других применениях инструкции `del`.

5.5 Кортежи

Списки и строки имеют много общего, например, для них можно получить элемент по индексу или применить операцию среза. Это два примера последовательностей. Еще одним встроенным типом последовательностей является *кортеж* (tuple).

Кортеж состоит из набора значений, разделенных запятой, например:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Кортежи могут быть вложенными:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Как Вы заметили, при выводе кортежи всегда заключаются в круглые скобки, для того, чтобы вложенные кортежи воспринимались корректно. Вводить их можно как в скобках, так и без них, хотя в некоторых случаях скобки необходимы (если кортеж является частью более сложного выражения).

Кортежи имеют множество применений: пара координат (x, y), запись в базе данных и т. д. Кортежи, как и строки, не допускают изменений: нельзя присвоить значение

элементу кортежа (тем не менее, Вы можете имитировать такую операцию с помощью срезов и последующего объединения).

Для того, чтобы сконструировать пустые или содержащие один элемент кортежи, приходится идти на синтаксические ухищрения. Пустой кортеж создается с помощью пустой пары скобок, кортеж с одним элементом — с помощью значения и следующей за ним запятой (не достаточно заключить значение в скобки). Несколько неприятно, но эффективно. Например:

```
>>> empty = ()
>>> # обратите внимание на завершающую запятую
... singleton = 'hello',
>>> len(empty)
0
>>> empty
()
>>> len(singleton)
1
>>> singleton
('hello',)
```

Инструкция `t = 12345, 54321, 'hello!'` — пример упаковки в кортеж: значения `12345`, `54321` и `'hello!'` упаковываются вместе в один кортеж. Также возможна обратная операция — распаковки кортежа:

```
>>> x, y, z = t
```

Распаковка кортежа требует, чтобы слева стояло столько переменных, сколько элементов в кортеже. Следует заметить, что множественное присваивание является всего лишь комбинацией упаковки и распаковки кортежа. Иногда может оказаться полезной распаковка списка:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a1, a2, a3, a4 = a
```

Как мы уже говорили, кортежи, как и строки, не допускают изменений. Однако, в отличие от строк, кортежи могут содержать объекты, которые можно изменить с помощью методов:

```
>>> t = 1, ['foo', 'bar']
>>> t
(1, ['foo', 'bar'])
>>> t[1] = []            # Ошибка
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

```
>>> t[1].append('baz')   # Правильно
>>> t
(1, ['foo', 'bar', 'baz'])
```

5.6 Словари

Еще один встроенный в Python тип данных — *словарь* (dictionary) — часто называют ассоциативным массивом. В отличие от последовательностей, индексируемых диапазоном чисел, доступ к элементам словаря производится по *ключу*, который может быть любого типа, не допускающего изменений[1] — строки и числа всегда могут быть ключами. Кортежи могут использоваться в качестве ключа, если они содержат строки, числа и кортежи, удовлетворяющие этому правилу. Нельзя использовать списки, так как их можно изменить (не создавая нового объекта-списка) с помощью, например, метода `append()`.

Лучше всего представлять словарь как неупорядоченное множество пар *ключ: значение*, с требованием уникальности ключей в пределах одного словаря. Пара фигурных скобок `{}` создает пустой словарь. Помещая список пар *key*: *value*, разделенных запятыми, в фигурные скобки, Вы задаете начальное содержимое словаря. В таком же виде записывается словарь при выводе.

Основные операции над словарем — сохранение с заданным ключом и извлечение по нему значения. Также можно удалить пару *key*: *value* с помощью инструкции `del`. Если Вы сохраняете значение с ключом, который уже используется, то старое значение забывается. При попытке извлечь значение с несуществующим ключом, генерируется исключение `KeyError`.

Метод `keys()` для словаря возвращает список всех используемых ключей в произвольном порядке (если Вы хотите, чтобы список был отсортирован, просто примените к нему метод `sort()`). Чтобы определить, используется ли определенный ключ — используйте метод `has_key()`.

Приведем простой пример использования словаря в качестве телефонного справочника:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
```

[1] На самом деле, в качестве ключа может служить любой объект (в том числе допускающий изменения), для которого определена функция `hash()`. Правило относится, скорее, к *стандартным* типам данных языка.

```
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

5.7 Подробнее об условиях

Помимо описанных ранее операторов сравнения, существует еще несколько условных операторов.

Операторы `in` и `not in` проверяют, есть указанное значение в последовательности. Операторы `is` и `is not` определяют, ссылаются ли две переменные на один и тот же объект. Все приведенные здесь операторы имеют одинаковый приоритет, который ниже, чем у арифметических операторов.

Логические выражения могут быть сцеплены: например, '`a < b == c`' проверяет, меньше ли *a* чем *b* и равны ли *b* и *c*.

Логические выражения можно группировать с помощью логических операторов `and` и `or`, а также результат можно инвертировать оператором `not`. Все логические операторы имеют меньший приоритет, чем операторы сравнения. Среди логических операторов, `not` имеет наибольший приоритет и `or` — наименьший. Таким образом, '`A or not B and C`' эквивалентно '`A or ((not B) or C)`'. Безусловно, можно использовать скобки для определения порядка вычисления.

Аргументы логических операторов `and` и `or` вычисляются справа налево до тех пор, пока результат не будет определен. Например, если выражения *A* и *C* истинны, но *B* ложно, в '`A and B and C`' выражение *C* вычисляться не будет. Вообще говоря, возвращаемое значение операторов `and` и `or` является не логическим, а равно значению последнего вычисленного аргумента.

Можно присвоить результат сравнения или логического выражения переменной:

```
>>> string1, string2, string3 = \
...     '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Обратите внимание, что, в отличие от C, присваивание не может находиться внутри выражения. Такое ограничение позволяет избежать ошибок, типичных для программ на C: использование = вместо ==.

5.8 Сравнение последовательностей

Объекты-последовательности можно сравнивать с другими объектами того же типа. Сравнение производится *лексикографически*: сначала сравниваются первые элементы последовательностей, и, если они отличаются, то результат их сравнения определяет результат; если они равны, сравниваются следующие элементы и т. д., пока не будет определен результат или одна из последовательностей не будет исчерпана. Если два элемента сами являются последовательностями одного типа, то лексикографическое сравнение выполняется рекурсивно. Если все элементы двух последовательностей в результате сравнения оказываются равными, то последовательности считаются равными. Если одна из последовательностей является началом другой, то меньшей считается последовательность с меньшим количеством элементов. Лексикографический порядок строк определяется порядком следования ASCII символов. Приведем несколько примеров сравнения последовательностей одинакового типа:

```
(1, 2, 3)              <  (1, 2, 4)
[1, 2, 3]              <  [1, 2, 4]
'ABC' < 'C' < 'Pascal' <  'Python'
(1, 2, 3, 4)           <  (1, 2, 4)
(1, 2)                 <  (1, 2, -1)
(1, 2, 3)              == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   <  (1, 2, ('abc', 'a'), 4)
```

Заметим, что сравнение объектов различного типа допустимо. Результат будет вполне определенным, однако не следует на него полагаться — правила сравнения объектов различного типа могут измениться в следующих версиях языка. Числа сравниваются в соответствии с их значениями, так `0` равен `0.0`, и т. д.

Глава 6

Модули

Когда Вы выходите из интерпретатора и заходите снова, все сделанные Вами определения (функции, переменные) безвозвратно теряются. Поэтому, если Вы хотите набрать сколько-нибудь длинную программу, для приготовления ввода лучше воспользоваться текстовым редактором. По мере возрастания программы, Вы, возможно, захотите разбить ее на несколько файлов, чтобы их было легче поддерживать. Вы можете захотеть использовать написанные Вами полезные функции в нескольких программах, не копируя их определения в каждую из программ.

Python позволяет поместить определения в файл и использовать их в программах и интерактивном режиме. Такой файл называется модулем. Определения из модуля могут быть *импортированы* в другие модули и в *главный* модуль (коллекция переменных, доступная в программе и в интерактивном режиме).

6.1 Создание и использование модулей

Модуль — файл, содержащий определения и другие инструкции языка Python. Имя файла образуется путем добавления к имени модуля суффикса (расширения) '.py'. В пределах модуля, его имя доступно в глобальной переменной `__name__`. Например, используя Ваш любимый текстовый редактор, создайте в текущем каталоге файл с именем 'fibo.py' следующего содержания:

```
'''\
Генерация и вывод чисел Фибоначчи
'''

def fib(n):
    '''Выводит последовательность чисел Фибоначчи,
    не превышающих n'''
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):
    '''Возвращает список, содержащий числа ряда
    Фибоначчи, не превышающие n'''
```

```
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Теперь запустите интерпретатор и импортируйте только что созданный модуль:

```
>>> import fibo
```

Эта инструкция не вводит имена функций, определенных в `fibo` прямо в текущее пространство имен, она только вводит имя модуля `fibo`. Используя имя модуля, Вы можете получить доступ к функциям:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Если Вы собираетесь использовать функцию часто, можете присвоить ее локальной переменной:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Модуль может содержать любые инструкции, предназначенные для его инициализации, а не только определения функций. Они выполняются, только когда модуль импортируется первый раз.

Каждый модуль имеет собственное пространство имен, являющееся глобальной областью видимости для всех определенных в нем функций. Таким образом, автор модуля может использовать глобальные переменные, не беспокоясь о возможных конфликтах с глобальными переменными пользователя. С другой стороны, если Вы знаете, что делаете, Вы можете получить доступ к глобальным переменным модуля точно так же, как и к его функциям (на самом деле, функции тоже являются переменными), *modname.itemname*.

Модули могут импортировать другие модули. Обычно инструкцию `import` располагают в начале модуля или программы. Имена импортируемых модулей помещаются в текущее пространство имен импортирующего модуля или программы.

Другой вариант инструкции `import` импортирует имена из модуля непосредственно в текущее пространство имен:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

В этом случае имя модуля не будет представлено в текущей области видимости (в приведенном примере, имя `fibo` не определено).

Еще один вариант инструкции `import` позволяет импортировать все имена, определенные в модуле, кроме имен, начинающихся с символа подчеркивания (‘_’):

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Часто возникает необходимость импортировать модуль или объект модуля, используя для него локальное имя, отличное от исходного. Например, следующий код позволяет заменить имя `string` на `_string` (которое не будет импортироваться инструкцией ‘`from` *`my_module`* `import *`’) при написании модуля:

```
import string
_string = string
del string
```

Еще один пример показывает, как можно избежать конфликта имени, определенного в модуле, со встроенным именем:

```
import anydbm
dbopen = anydbm.open
```

Начиная с версии 2.0, подобные операции можно произвести гораздо проще (и безопаснее) благодаря расширению синтаксиса инструкции `import`:

```
import string as _string

from anydbm import open as dbopen
```

Следует заметить, что `as` не является зарезервированным словом, и Вы можете попрежнему определять переменные с таким именем.

Если имя модуля, который необходимо импортировать, становится известным только во время выполнения программы, Вы можете воспользоваться инструкцией `exec` (‘`exec 'import ' +` *`module_name`*’) или встроенной функцией `__import__()` (см. раздел 12).

6.2 Поиск модулей

Когда импортируется модуль, например spam, интерпретатор ищет файл с именем 'spam.py' в текущем каталоге, затем в каталогах, указанных в переменной окружения PYTHONPATH, затем в зависящих от платформы путях по умолчанию.

Каталоги, в которых осуществляется поиск, хранятся в переменной sys.path. Таким образом, программы на языке Python могут изменять пути поиска модулей во время их выполнения.

6.3 "Компилированные" файлы

Для ускорения запуска программ, использующих большое количество модулей, если уже существует файл с именем 'spam.pyc' в том же каталоге, где найден 'spam.py', считается, что он содержит "байт-компилированный" модуль spam. Если такого файла нет, то он создается, и время последнего изменения 'spam.py' записывается в созданном 'spam.pyc' (при последующем использовании, '.pyc'-файл игнорируется, если исходный '.py'-файл был изменен).

Обычно Вам не надо ничего делать для создания 'spam.pyc'. Как только 'spam.py' успешно откомпилирован, интерпретатор пытается записать компилированную версию в 'spam.pyc'. Если интерпретатору по каким-либо причинам это не удается (например, недостаточно пользовательских полномочий), ошибки не возникает. Если же файл записан не полностью, далее он распознается как неработоспособный и игнорируется. Содержимое байт-компилированных файлов является платформно-независимым (но может быть разным для разных версий интерпретатора), так что каталог с модулями может совместно использоваться машинами с разными архитектурами.

Несколько тонкостей для опытных пользователей:

- Если интерпретатор вызывается с опцией -O или переменная окружения PYTHONOPTIMIZE имеет непустое значение, интерпретатор генерирует оптимизированный байт-код и сохраняет его в '.pyo'-файлах. В настоящий момент оптимизация дает не слишком много: при этом удаляются инструкции assert, игнорируются инструкции 'if __debug__: ...', не сохраняется информация о нумерации строк в исходных '.py'-файлах. В этом случае оптимизируются *все* используемые модули, '.pyc'-файлы игнорируются.
- Опция -OO приводит к тому, что интерпретатор выполняет оптимизацию которая, в некоторых (редких) случаях, может привести к сбоям в работе программ. В настоящий момент, помимо действий, выполняемых с опцией -O, удаляются строки документации, давая более компактные '.pyo'-файлы. Так как некоторые программы могут рассчитывать на наличие строк документации, используйте эту опцию с осторожностью.
- Для программы, запускаемой из командной строки, байт-код никогда не записывается в '.pyc'- или '.pyo'-файл. Поэтому, если Вы хотите уменьшить время, требую-

щееся для загрузки, поместите большую часть кода в модуль, оставив в программе лишь загрузочную часть, которая импортирует этот модуль.

- Возможно использование модуля (или запуск программы) при наличии '.pyc'-файла (или '.pyo'-файла, если используется одна из опций -O или -OO), даже если отсутствует '.py'-файл. Таким образом, Вы можете распространять библиотеки и программы в виде, из которого относительно сложно извлечь информацию об используемых алгоритмах.

- Модуль `compileall` позволяет создать '.pyc'- (или '.pyo'-) файлы для всех модулей в каталоге. Это может быть особенно полезно, если вы хотите ограничить доступ к каталогу, в котором находится библиотека. Заметим, что интерпретатор не будет использовать '.pyc'-файлы, если он запущен с включенной оптимизацией, и '.pyo'-файлы, если оптимизация выключена (если же отсутствует '.py'-файл, модуль окажется недоступным).

6.4 Стандартные модули

Python распространяется с библиотекой стандартных модулей, которой посвящена третья часть книги. Часть модулей встроена в интерпретатор, обеспечивая доступ к операциям, которые не входят в ядро языка, но, тем не менее, встроены либо из соображений эффективности, либо для обеспечения доступа к примитивам операционной системы. Набор таких модулей зависит от конфигурации, так, например, модуль `amoeba` присутствует только в системах, которые поддерживают примитивы Amoeba. Один модуль заслуживает особого внимания: `sys`, который присутствует всегда. Переменные `sys.ps1` и `sys.ps2` при работе в интерактивном режиме определяют строки, используемые для первичного и вторичного приглашения:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Привет!'
Привет!
C>
```

Переменная `sys.path` содержит список строк с именами каталогов, в которых происходит поиск модулей. Она инициализируется из значения переменной окружения PYTHONPATH и встроенного значения по умолчанию. Вы можете изменить ее значение, используя стандартные операции со списками:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.5 Функция dir()

Для выяснения имен, определенных в модуле, можно использовать встроенную функцию `dir()`. Она возвращает отсортированный список строк:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names',
'copyright', 'exit', 'maxint', 'modules', 'path',
'ps1', 'ps2', 'setprofile', 'settrace', 'stderr',
'stdin', 'stdout', 'version']
```

Без аргументов, `dir()` возвращает список имен, определенных в текущей области видимости:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Обратите внимание, что перечисляются имена объектов *всех* типов: переменные, модули, функции и т. д.

Список, возвращаемый функцией `dir()` не содержит имена встроенных функций и переменных — они определены в стандартном модуле `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError',
'EOFError', 'IOError', 'ImportError', 'IndexError',
'KeyError', 'KeyboardInterrupt', 'MemoryError',
'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit',
'TypeError', 'ValueError', 'ZeroDivisionError',
'__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile',
'filter', 'float', 'getattr', 'hasattr', 'hash', 'hex',
'id', 'input', 'int', 'len', 'long', 'map', 'max',
'min', 'oct', 'open', 'ord', 'pow', 'range',
'raw_input', 'reduce', 'reload', 'repr', 'round',
'setattr', 'str', 'type', 'xrange']
```

6.6 Пакеты

Пакеты — способ структурирования пространств имен модулей, используя “точечную запись”. Например, имя модуля `A.B` обозначает подмодуль с именем `A` в пакете `B`. Так же, как использование модулей делает безопасным использование глобального пространства имен авторами различных модулей, использование точечной записи делает безопасным использование имен модулей авторами многомодульных пакетов.

Предположим, Вы хотите спроектировать совокупность модулей (“пакет”) для единообразной обработки звуковых файлов и данных. Существует множество форматов звуковых файлов (обычно распознаваемых по их расширению, например ‘.wav’, ‘.aiff’, ‘.au’), так что Вам необходимо создавать и поддерживать все возрастающий набор модулей для преобразования между различными форматами файлов. Вы можете захотеть выполнять множество различных операций над звуковыми данными (микширование, добавление эха, частотная обработка, создание искусственного стереоэффекта), то есть, вдобавок, Вы будете писать нескончаемый поток модулей для выполнения этих операций. Вот примерная структура Вашего пакета (выраженная в терминах иерархической файловой системы):

```
Sound/                          Верхний уровень пакета
       __init__.py              Инициализация пакета
       Formats/                 Работа с файлами
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
       Effects/                 Звуковые эффекты
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
       Filters/                 Фильтры
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Файл ‘__init__.py’ необходим для того, чтобы Python распознавал каталог, как содержащий пакет — таким образом предотвращается маскировка полноценных модулей, расположенных далее в путях поиска, каталогами с распространенными именами (такими как ‘`string`’). В простейшем случае, ‘__init__.py’ — пустой файл, но может содер-

жать код инициализации пакета и/или устанавливать переменную `__all__`, описанную ниже.

Пользователи пакета могут импортировать индивидуальные модули пакета, например:

```
import Sound.Effects.echo
```

В этом случае загружается модуль `Sound.Effects.echo`. На него нужно ссылаться по полному имени:

```
Sound.Effects.echo.echofilter(input, output,
                              delay=0.7, atten=4)
```

Вы можете использовать альтернативный способ импортирования подмодуля:

```
from Sound.Effects import echo
```

В этом случае также загружается модуль `Sound.Effects.echo`, и делает его доступным для использования без префикса:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Еще один вариант — импортировать желаемую функцию или переменную непосредственно:

```
from Sound.Effects.echo import echofilter
```

И в этом случае загружается модуль `Sound.Effects.echo`, но на этот раз, функция `echofilter()` становится доступной для использования непосредственно:

```
echofilter(input, output, delay=0.7, atten=4)
```

Заметим, что при использовании '`from` *`package`* `import` *`item`*', *`item`* может быть модулем, подпакетом или другим именем, определенном в пакете *`package`*, таким как функция, класс или переменная. Инструкция `import` сначала проверяет, определено ли имя *`item`* в пакете, если нет, считает его модулем и пытается загрузить. Если при загрузке возникает ошибка, генерируется исключение `ImportError`.

И наоборот, при использовании инструкции '`import` *`item.subitem.subsubitem`*', каждая единица, кроме последней, должна быть пакетом. Последняя единица может быть модулем или пакетом, но не может быть классом, функцией или переменной, определенной в предыдущей единице.

6.6.1 Импортирование всего содержимого пакета (модуля)

Что же происходит, когда пользователь использует 'from Sound.Effects import *'? В идеале, интерпретатор должен каким-либо образом обойти файлы, находящиеся в каталоге пакета, и импортировать их все. К сожалению, такой подход не будет работать достаточно хорошо на таких платформах, как Macintosh и Windows, где файловая система не всегда имеет точную информацию о регистре букв в именах файлов. В этом случае нет надежного пути узнать, с каким именем должен быть импортирован файл с именем 'ECHO.PY': echo, Echo или ECHO.

Единственный выход для автора — снабдить пакет явным указателем его содержимого. Инструкция import использует следующее соглашение: если в инициализационном файле '__init__.py' определен список с именем __all__, он используется в качестве списка имен модулей, которые должны импортироваться при использовании 'from *package* import *'. Поддержка этого списка в соответствии с текущим составом пакета возлагается на автора. Можно также не определять список __all__, если авторы не считают уместным импортирование *. Например, файл 'Sounds/Effects/__init__.py' может содержать следующий код:

```
__all__ = ["echo", "surround", "reverse"]
```

Это означает, что from Sound.Effects import * импортирует три указанных модуля из пакета Sound.

Если список __all__ не определен, 'from Sound.Effects import *' *не* будет импортировать все модули пакета Sound.Effects, а только имена, явным образом определенные в инициализационном файле '__init__.py' (включая явно импортированные в нем модули). Кроме того, в текущую область видимости попадут модули пакета, явно загруженные предыдущими инструкциями import, например:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

В приведенном примере, модули echo и surround импортируются в текущее пространство имен, потому что они уже определены в пакете Sound.Effects на момент выполнения инструкции from ... import.

Заметим, что импортирование всех имен, определенных в модуле или пакете, обычно приводит к засорению пространства имен и, как результат, к возможным конфликтам. (Некоторые модули специально спроектированы таким образом, чтобы экспортировать только имена, следующие определенному шаблону.) Кроме того, при этом глобальные переменные модуля становятся доступными только для чтения — при попытке присвоить такой переменной новое значение Вы создадите новую (локальную) переменную с таким же именем. А все изменения, внесенные в глобальные переменные модуля после его инициализации, не будут видны за его пределами.

Мы рекомендуем стараться использовать запись 'from *package* import *specific_module*', за исключением случаев, когда необходимо использовать модули с одинаковыми именами из разных пакетов.

6.6.2 Связи между модулями пакета

Часто возникает необходимость в связях между модулями одного пакета. Например, модуль surround может использовать модуль echo. В самом деле, подобные связи распространены настолько, что инструкция import сначала просматривает содержимое пакета, в который входит содержащий эту инструкцию модуль, и только потом в путях поиска модулей. Таким образом, модуль surround может просто использовать 'import echo' или 'from echo import echofilter'.

Когда пакеты разделены на подпакеты (пакет Sound в примере), нет краткой записи для ссылок между ответвлениями пакета — нужно использовать полное имя. Например, если модуль Sound.Filters.vocoder должен использовать модуль echo пакета Sound.Effects, нужно использовать 'from Sound.Effects import echo'.

Глава 7

Ввод/вывод

Существует несколько способов представления программного вывода: данные могут быть напечатаны в виде, удобном для восприятия, или записаны в файл для дальнейшего использования. В этой главе обсуждаются некоторые возможности представления выводимых данных.

7.1 Форматированный вывод

До сих пор мы использовали два способа вывода: вывод значений выражений в интерактивном режиме и с помощью инструкции `print` (третий способ — метод объектов-файлов `write()`).

Часто возникает желание иметь больший контроль над форматированием вывода, чем просто выводить значения, разделенные пробелом. Конечно, Вы можете сами обрабатывать строки: с помощью операций среза и объединения можно создать любое расположение, какое только Вы сможете представить. Строки имеют методы, позволяющие дополнять их пробелами до необходимой ширины колонки[1]. Другой путь — использовать оператор `%` со строкой в качестве левого аргумента. Оператор `%` интерпретирует строку справа как строку формата в стиле функции `sprintf()` в C, которую нужно применить к правому аргументу, и возвращает строку с результатом форматирования.

Безусловно, остается еще один вопрос: как получить строковое представление для значений различного типа? К счастью, Python предоставляет возможность преобразовывать значение любого типа в строку: с помощью функции `str()`. Фактически язык предоставляет две функции для получения строкового представления — `repr()` (тот же эффект можно получить просто заключив выражение в обратные кавычки: `` `expr` ``) и `str()`. Первый способ, например, используется интерпретатором для вывода значений выражений в интерактивном режиме, второй — для вывода аргументов инструкцией `print`. Функция `str()` по возможности возвращает представление, наиболее пригодное для вывода, а функция `repr()` — для ввода выражения в интерактивном режиме. Приведем несколько примеров:

```
>>> x = 10 * 3.14
```

[1] Эти методы строк появились в версии 1.6 языка Python. В предыдущих версия они доступны в виде функций, определенных в модуле `string`.

Число 31.4 не может быть точно представлено в двоичном виде, поэтому:

```
>>> x
31.399999999999999
```

Однако функция `str()` выведет число с разумной точностью:

```
>>> y = 200*200
>>> s = 'Значение x равно ' + str(x) + \
...     ', значение y равно ' + str(y) + '...'
>>> print s
Значение x равно 31.4, значение y равно 40000...
```

Длинные целые числа записываются в языке Python с суффиксом ‘`L`’. Начиная с версии 1.6, функция `str()` его не выводит:

```
>>> repr(1000L)
'1000L'
>>> str(1000L)
'1000'
```

Строковое представление можно получить и для других типов:

```
>>> p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.399999999999999, 40000]'
>>> `x, y, ('spam', 'eggs')`
"(31.399999999999999, 40000, ('spam', 'eggs'))"
```

Функция `repr()` (или ` `` `) добавляет кавычки и записывает спецсимволы с помощью управляющих последовательностей:

```
>>> hello = 'hello, world\n'
>>> print hello
hello, world

>>> hellos = `hello`
>>> print hellos
'hello, world\012'
```

Выведем таблицу квадратов и кубов двумя описанными способами:

```
>>> for x in range(1, 11):
...     print str(x).rjust(2), str(x*x).rjust(3),
...     # Обратите внимание на завершающую запятую
...     print str(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

```
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Обратите внимание на то, что один пробел между колонками был добавлен инструкцией `print`.)

Этот пример демонстрирует использование метода строк `rjust()`, который выравнивает строку вправо в поле заданной ширины, дополняя ее слева пробелами. Аналогично действуют методы `ljust()` и `center()`. Они не выводят ничего — просто возвращают новую строку. Если исходная строка слишком длинная, она не обрезается, а возвращается в неизменном виде: обычно лучше внести беспорядок в расположение колонок, чем вывести неверное значение. (Если Вы действительно хотите ее обрезать, воспользуйтесь операцией среза: '`s.ljust(n)[0:n]`'.)

Также может быть полезна функция `zfill()`, определенная в модуле `string`, которая дополняет слева нулями строку с числом, корректно обрабатывая знаки плюс и минус:

```
>>> import string
>>> string.zfill('12', 5)
'00012'
```

```
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

Использование оператора `%` выглядит примерно так:

```
>>> import math
>>> print 'Значение PI приближенно равно %5.3f.' % \
...        math.pi
Значение PI приближенно равно 3.142.
```

Если в строке нужно вывести несколько значений, в качестве правого операнда используется кортеж:

```
>>> table = {'Sjoerd': 4127,
...          'Jack'  : 4098,
...          'Dcab'  : 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Sjoerd     ==>       4127
Dcab       ==>       7678
Jack       ==>       4098
```

Большинство форматов работают точно так же, как и в C. Однако, если типы аргументов не соответствуют формату, интерпретатор приводит их к необходимому типу (например, выражение любого типа может быть преобразовано в строку с помощью встроенной функции `str()`) или, если это невозможно, генерирует исключение[2]. Вы можете использовать `*` для того, чтобы передать отдельным параметром ширину поля или точность.

Заметим, что если правый аргумент кортеж, он *всегда* считается списком аргументов:

```
>>> def f(x):
...     print 'Функции передано значение "%s"' % x
...
>>> f(1)
Функции передано значение "1"
>>> f([1, 2])
Функции передано значение "[1, 2]"
>>> # интерпретируется не так, как Вы ожидали
... f((1,))
Функции передано значение "1"
```

[2]Из этого правила есть исключение: интерпретатор не будет преобразовывать строку к числовому типу.

```
>>> # ошибка
... f((1, 2))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in f
TypeError: not all arguments converted
```

В данном случае надежнее будет использовать в качестве правого операнда кортеж, состоящий из одного элемента:

```
>>> def f(x):
...     print 'Функции передано значение "%s"' % (x,)
...
>>> # теперь все правильно
... f((1,))
Функции передано значение "(1,)"
>>> f((1, 2))
Функции передано значение "(1, 2)"
```

В случае длинных строк формата, Вы можете захотеть ссылаться на переменные по имени вместо их положения. Это можно сделать, используя расширенную запись в виде `%(`*`name`*`)`*`format`*, например:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098,
...          'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; \
... Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Такая запись особенно полезна в комбинации со встроенной функцией `vars()`, которая возвращает словарь с переменными в текущей области видимости.

Более подробно операции над строками описаны в разделе 11.2.1.

7.2 Чтение и запись файлов

Встроенная функция `open()` возвращает объект-файл (file) и обычно используется с двумя аргументами: '`open(`*`filename`*`, `*`mode`*`)`'.

```
>>> f=open('/tmp/workfile', 'wb')
>>> print f
<open file '/tmp/workfile', mode 'wb' at 80a0960>
```

Первый аргумент — строка, содержащая имя файла, второй аргумент — строка, содержащая несколько символов, описывающих режим использования файла. Режим может быть `'r'`, если файл открывается только для чтения, `'w'` — только для записи (существующий файл будет перезаписан), и `'a'` — для дописывания в конец файла. В режиме `'r+'` файл открывается сразу для чтения и записи. Аргумент *mode* не является обязательным: если он опущен, подразумевается `'r'`.

В Windows (а в некоторых случаях и в Macintosh) файлы по умолчанию открываются в текстовом режиме — для того, чтобы открыть файл в двоичном режиме, необходимо к строке режима добавить `'b'`. Следует помнить, что двоичные данные, такие как картинки в формате JPEG и даже текст в UNICODE, при чтении из файла или записи в файл, открытый в текстовом режиме, будут испорчены! Лучший способ оградить себя от неприятностей — всегда открывать файлы в двоичном режиме, даже на тех платформах, где двоичный режим используется по умолчанию (возможно у Вас когда-нибудь возникнет желание запустить программу на другой платформе).

7.2.1 Методы объектов-файлов

Примеры в этом разделе рассчитаны на то, что объект-файл *f* уже создан.

f`.read(`*size*`)` считывает и возвращает некоторое количество данных из файла. Аргумент *size* не является обязательным. Если он опущен или отрицательный, будет считано все содержимое файла, в противном случае, будет считано не более *size* байт данных. По достижении конца файла, возвращается пустая строка ().

```
>>> f.read() # Считываем все содержимое файла
'This is the entire file.\012'
>>> f.read()
''
```

f`.readline()` считывает из файла одну строку. Возвращаемая строка всегда заканчивается символом новой строки (\n), за исключением последней строки файла, если файл не заканчивается символом новой строки. Это делает возвращаемое значение недвусмысленным: если `readline()` возвращает пустую строку — значит, достигнут конец файла, в то время как пустая строка будет представлена как `'\n'`.

```
>>> f.readline() # Считываем первую строку
'This is the first line of the file.\012'
>>> f.readline() # Считываем вторую строку
'Second line of the file\012'
>>> f.readline() # Достигли конца файла
''
```

f`.readlines()` считывает все содержимое файла, и возвращает список строк.

```
>>> f.readlines()
['This is the first line of the file.\012',
'Second line of the file\012']
```

f.write(*s*) записывает содержимое строки *s* в файл.

```
>>> f.write('This is a test\n')
```

f.tell() возвращает текущее положение в файле в байтах, отсчитываемое от начала файла. Чтобы изменить текущее положение, используйте '*f*.seek(*offset*, *from_what*)'. Новое положение вычисляется путем добавления *offset* к точке отсчета. Точка отсчета выбирается в зависимости от значения аргумента *from_what*: 0 соответствует началу файла (используется по умолчанию, если метод вызывается с одним аргументом), 1 — текущему положению и 2 — концу файла.

```
>>> f=open('/tmp/workfile', 'rb+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 5th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

После того, как Вы закончили все операции с файлом, закройте файл с помощью *f*.close(). При попытке использовать закрытый файл для операций чтения/записи генерируется исключение ValueError:

```
>>> f.close()
>>> f.read()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Объекты-файлы имеют еще несколько методов, используемых не так часто (isatty(), truncate()). Для получения о них полной информации смотрите раздел 11.7.

7.2.2 Модуль pickle

Строки легко могут быть записаны в файл и считаны из него. Числа требуют приложения небольших усилий, так как метод read() всегда возвращает строки, которые нужно обработать, например, с помощью функции int(). Однако задача сильно усложняется,

если Вы хотите сохранять более сложные типы данных, такие как списки, словари или экземпляры классов.

Чтобы пользователю не приходилось постоянно писать и отлаживать код для сохранения сложных типов данных, Python предоставляет для этих целей стандартный модуль `pickle`. Этот изумительный модуль позволяет получить представление почти любого объекта (даже некоторые формы кода) в виде набора байтов (строки), одинакового для всех платформ. Такой процесс называют “консервированием” (pickling). Такое представление (законсервированный объект) можно сохранить в файле или передать через сетевое соединение на другую машину. К считанному из файла или принятому на другой машине законсервированному объекту может быть применена операция восстановления (unpickling).

Если у Вас есть объект *x* и файловый объект *f*, открытый для записи, простейший способ сохранить объект потребует всего одну строку кода:

```
pickle.dump(x, f)
```

Так же просто можно восстановить объект (*f* — файловый объект, открытый для чтения):

```
x = pickle.load(f)
```

Для получения полного представления о возможностях модуля `pickle`, смотрите его описание в третьей части книги.

Использование модуля `pickle` является стандартным в языке Python путем сохранения так называемых *долгоживущих* (persistent) объектов для дальнейшего использования. Модуль настолько часто используется, что многие авторы дополнительных модулей заботятся о том, чтобы новые типы данных (например, матрицы) могли быть правильно законсервированы.

Глава 8

Ошибки и исключения

До сих пор мы лишь упоминали об ошибках, но если Вы пробовали приведенные примеры, то могли увидеть некоторые из них. Можно выделить (как минимум) два различимых типа ошибок: *синтаксические ошибки* и *исключения*[1].

8.1 Синтаксические ошибки

Синтаксические ошибки, пожалуй, чаще всего встречаются во время изучения языка:

```
>>> while 1 print 'Hello world'
  File "<stdin>", line 1
    while 1 print 'Hello world'
                ^
SyntaxError: invalid syntax
```

Синтаксический анализатор выводит строку, содержащую ошибку, и указывает место, где ошибка была обнаружена. Ошибка обычно вызвана лексемой, предшествующей стрелке: в приведенном примере, ошибка обнаружена на месте ключевого слова `print`, так как перед ним отсутствует двоеточие (‘`:`’). Имя файла и номер строки выводится для того, чтобы Вы знали, где искать ошибку в случае, если инструкции считываются из файла.

8.2 Исключения

Даже если инструкция или выражение синтаксически верно, ошибка может произойти при попытке выполнения. Ошибки, обнаруженные во время выполнения, не являются безусловно фатальными, и скоро Вы узнаете, как их можно обрабатывать в программах на языке Python. Большинство исключений, однако, не обрабатываются программами и приводят к сообщениям об ошибке:

```
>>> 10 * (1/0)
Traceback (innermost last):
```

[1]Синтаксические ошибки также являются исключениями, которые, однако, не могут быть перехвачены на том же уровне.

```
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (innermost last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (innermost last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

Последняя строка сообщения показывает, что произошло. Исключения бывают разного типа — он выводится в сообщении. Типы исключений в приведенном примере: `ZeroDivisionError`, `NameError` и `TypeError`. Имена стандартных исключений являются встроенными идентификаторами, но не являются зарезервированными ключевыми словами.

Сразу после типа выводятся подробности возникновения исключения. Предшествующая часть сообщения об ошибке показывает контекст возникновения исключительной ситуации в форме содержимого стека. При этом выводятся строки исходного текста, за исключением строк, читаемых со стандартного ввода.

В главе 13 перечислены все встроенные исключения и их назначение.

8.3 Обработка исключений

Вы можете писать программы, которые будут обрабатывать определенные исключения. Посмотрите на следующий пример, в котором пользователю будет выдаваться приглашение до тех пор, пока не будет введено целое число или выполнение не будет прервано (обычно `Ctrl-C`). Во втором случае генерируется исключение `KeyboardInterrupt`.

```
>>> while 1:
...     try:
...         x = int(raw_input(
...             "Введите, пожалуйста, целое число: "))
...         break
...     except ValueError:
...         print "Вы ошиблись. Попробуйте еще раз..."
...
```

Инструкция `try` работает следующим образом.

- Сначала выполняется *ветвь* `try` (инструкции, находящиеся между ключевыми словами `try` и `except`).

- Если не возникает исключительной ситуации, *ветвь except* пропускается и выполнение инструкции try завершается.
- Если во время выполнения ветви try генерируется исключение, оставшаяся часть ветви пропускается. Далее, если тип (класс) исключения соответствует указанному после ключевого слова except, выполняется ветвь except и выполнение инструкции try завершается.
- Если исключение не соответствует указанному после ключевого слова except, то оно передается внешнему блоку try или, если обработчик не найден, исключение считается *не перехваченным*, выполнение прерывается и выводится сообщение об ошибке.

Инструкция try может иметь более одной ветви except, определяя обработчики для разных исключений. Выполняться будет (как максимум) только один из них. Обрабатываются только исключения, сгенерированные в соответствующей ветви try, но не в других обработчиках инструкции try. После ключевого слова except может быть указано несколько типов исключений в виде кортежа:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

В последней ветви except тип исключения может быть опущен — в этом случае будут обрабатываться все исключения. Используйте такую запись с особой осторожностью — так Вы легко можете замаскировать реальные ошибки! Перехват всех исключений можно использовать для вывода сообщения об ошибке, а затем повторно сгенерировать его (позволяя обрабатывать исключение в другом месте):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, exc:
    print "Ошибка ввода/вывода", exc
except ValueError:
    print "Не могу преобразовать данные к целому типу."
except:
    print "Неожиданная ошибка:", sys.exc_info()[0]
    raise  # повторно генерирует последнее
           # перехваченное исключение
```

После всех ветвей except, инструкция try может содержать ветвь else, которая будет выполняться в случае, если во время выполнения ветви try исключения не генерируются. Например:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'Не могу открыть', arg
    else:
        print arg, 'содержит', len(f.readlines()), \
              'строк'
        f.close()
```

Обычно лучше использовать ветвь `else`, чем добавлять код в основную ветвь инструкции `try`, так как это позволяет избежать обработки исключений, сгенерированных кодом, который Вы вовсе не собирались защищать.

Исключение может иметь ассоциированное с ним значение — *аргумент*, переданный классу исключения при инициализации. Наличие и тип аргумента зависит от типа исключения. Ассоциированное значение используется при получении для исключения строкового значения. Чтобы получить значение исключения, в ветви `except` после класса исключения (или кортежа классов) укажите переменную:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'Имя', x, 'не определено'
...
Имя spam не определено
```

Если исключение не обрабатывается, значение исключения выводится в сообщении об ошибке после имени класса исключения.

Обработчик перехватывает не только исключения, сгенерированные непосредственно в блоке `try`, но и в вызываемых из него функциях. Например:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, exc:
...     print 'Ошибка времени выполнения:', exc
...
Ошибка времени выполнения: integer division or modulo
```

8.4 Генерация исключений

Инструкция `raise` позволяет программисту принудительно сгенерировать исключение. Например:

```
>>> raise NameError('HiThere')
Traceback (innermost last):
  File "<stdin>", line 1
NameError: HiThere
```

В качестве аргумента `raise` используется экземпляр класса. Класс указывает на тип исключения; аргумент, передаваемый конструктору, обычно описывает "подробности" возникновения исключительной ситуации.

8.5 Исключения, определяемые пользователем

Вы можете использовать свои собственные исключения, используя строковые выражения для обозначения его имени (устаревший способ) или создавая новые классы исключения. Например:

```
>>> class MyError(Exception): pass
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'Исключение MyError, value равно', e
...
Исключение MyError, value равно 4
>>> raise MyError(1)
Traceback (innermost last):
  File "<stdin>", line 1
__main__.MyError: 1
```

Подробную информацию о классах Вы можете получить в главе 9.

8.6 "Страхование" от ошибок

Еще один вариант записи инструкции `try` — с определением "страховочной" ветви `finally`, которая будет выполняться при любых обстоятельствах. Например:

```
>>> try:
...     raise KeyboardInterrupt()
```

```
... finally:
...     print 'До свидания!'
...
До свидания!
Traceback (innermost last):
  File "<stdin>", line 2
KeyboardInterrupt
```

Ветвь `finally` выполняется независимо от того, возникла ли исключительная ситуация во время выполнения блока `try` или нет, в том числе и если выход происходит по инструкции `break` или `return`.

Инструкция `try` может иметь либо одну или несколько ветвей `except`, либо одну ветвь `finally`, но не оба варианта сразу.

Глава 9

Классы

Механизм классов языка Python использует минимум нового синтаксиса и семантики. Он представляет собой смесь механизмов классов C++ и Modula-3. Так же, как и модули, классы не возводят абсолютного барьера между определением и пользователем, а скорее полагаются на хороший стиль пользователя “не вламываться” в определение. Однако наиболее важные особенности классов полностью сохранены: механизм наследования допускает несколько базовых классов, производный класс может переопределить любые методы базовых классов, из метода можно вызывать метод с тем же именем базового класса. Объекты могут содержать произвольное количество собственных данных.

Говоря в терминологии C++, все атрибуты класса (включая поля данных) являются *открытыми* (public), а все методы — *виртуальными* (virtual). Как и в Modula-3, нет сокращения, позволяющего ссылаться на атрибуты объекта из его метода: функция-метод определяется с явным первым аргументом, представляющим сам объект, и подставляемым автоматически при вызове. Подобно Smalltalk, классы сами являются объектами, хотя и в более широком смысле этого слова: в языке Python все типы данных являются объектами. Это позволяет использовать общую семантику для импортирования и переименования, но встроенные типы (как и в C++ и Modula-3) нельзя использовать в качестве базовых классов. Кроме того, так же как в C++, но в отличие от Modula-3, можно переопределить большинство встроенных операций над экземплярами классов.

9.1 Несколько слов о терминологии

По причине отсутствия универсальной терминологии, время от времени мы будем пользоваться терминами Smalltalk и C++. Также следует предупредить, что существует терминологическая ловушка для читателей, знакомых с объектно-ориентированным программированием: слово “объект” в языке Python совсем не обязательно означает экземпляр класса. Так, объекты встроенных типов, такие как целые числа, списки и даже некоторые экзотические, вроде файлов, тоже не являются экземплярами классов. Однако *все* объекты языка Python имеют общую часть семантики, лучше всего описываемую словом “объект”.

Один и тот же объект может быть привязан к нескольким именам, в том числе находящимся в различных пространствах имен (использование псевдонимов, aliasing). На это свойство обычно не обращают внимания при первом знакомстве, и его можно благополучно игнорировать, пока Вы работаете с неизменяемыми объектами (числа, строки, кортежи). Однако использование псевдонимов (преднамеренно!) отражается на семантике кода, работающего с изменяемыми объектами (списки, словари, файлы и т.

п.). Обычно из этого извлекают пользу, так как псевдонимы во многих отношениях работают аналогично указателям. Например, передача объекта в качестве аргумента достаточно эффективна, потому что реализована как передача указателя. Если функция изменяет объект, переданный в качестве аргумента, все изменения будут видны после возврата из функции — таким образом, отпадает необходимость в использовании двух различных механизмов передачи аргументов, как это сделано в языке Pascal.

9.2 Области видимости и пространства имен

Перед тем, как знакомиться с классами, следует рассказать о правилах языка, касающихся областей видимости. Определения классов выполняют несколько изящных приемов с пространствами имен, и Вам следует знать о работе областей видимости и пространств имен, для полного понимания происходящего.

Начнем с нескольких определений. *Пространство имен* определяет отображение имен в объекты. Большинство пространств имен в языке Python реализованы как словари, что, однако, никак себя не проявляет (кроме производительности) и может быть изменено в будущем. Вот несколько примеров пространств имен: множество встроенных имен (функции, исключения), глобальные имена в модуле, локальные имена при вызове функций. В этом смысле множество атрибутов объекта также образует пространство имен. Важно понимать, что между именами в разных пространствах имен нет связи. Например, два модуля могут определить функции с именем “maximize” не создавая при этом путаницы — пользователь должен ссылаться на них с использованием имени модуля в качестве приставки.

Под словом *атрибут* мы подразумеваем любое имя, следующее после точки: например, в выражении `z.real`, `real` является атрибутом объекта *z*. Строго говоря, имена в модулях являются атрибутами модуля: в выражении *modname.funcname*, *modname* является объектом-модулем и *funcname* является его атрибутом. В этом случае имеет место прямое соответствие между атрибутами модуля и глобальными именами, определенными в модуле: они совместно используют одно пространство имен[1]!

Атрибуты могут быть доступны только для чтения, а могут и допускать присваивание. Во втором случае Вы можете записать ‘*modname.attribute* `= 42`’ или даже удалить его, используя инструкцию `del`: ‘`del` *modname.attribute*’.

Пространства имен создаются в различные моменты времени и имеют разную продолжительность жизни. Пространство имен, содержащее встроенные имена, создается при запуске интерпретатора и существует все время его работы. Глобальное пространство имен модуля создается, когда он считывается, и, обычно, также существует до завершения работы интерпретатора. Инструкции, выполняемые на верхнем уровне, т. е. читаемые из файла-сценария или интерактивно, рассматриваются как часть модуля `__main__`, имеющего собственное глобальное пространство имен. (В действительности,

[1]Есть одно исключение. Объект-модуль имеет секретный атрибут `__dict__`, содержащий словарь, используемый для реализации пространства имен модуля. Имя `__dict__` является атрибутом, однако не является глобальным именем. Не следует использовать атрибут `__dict__` где-либо кроме отладчиков, так как это нарушает абстракцию реализации пространства имен.

встроенные имена также находятся в модуле — `__builtin__`.)

Локальное пространство имен функции создается при вызове функции и удаляется при выходе из нее (возвращается значение или генерируется исключение, которое не обрабатывается внутри функции). Безусловно, при рекурсивном вызове создается собственное локальное пространство имен для каждого вызова.

Область видимости — фрагмент программы, в котором пространство имен непосредственно доступно, то есть нет необходимости в использовании записи через точку для того, чтобы поиск имени производился в данном пространстве имен.

Несмотря на статическое определение, области видимости используются динамически. В любой момент времени выполнения программы используется ровно три вложенных области видимости (три непосредственно доступных пространства имен). Сначала поиск имени производится во внутренней области видимости, содержащей локальные имена. Далее — в средней, содержащей глобальные имена модуля. И, наконец, во внешней, содержащей встроенные имена.

Обычно локальная область видимости соответствует локальному пространству имен текущей функции (класса, метода). За пределами функции (класса, метода), локальная область видимости соответствует тому же пространству имен, что и глобальная: пространству имен текущего модуля.

Важно понимать, что область видимости определяется по тексту: глобальная область видимости функции, определенной в модуле — пространство имен этого модуля, независимо от того, откуда или под каким псевдонимом функция была вызвана. С другой стороны, реально поиск имен происходит динамически, во время выполнения. Однако язык развивается в сторону статического разрешения имен, определяемого во время "компиляции", поэтому не стоит полагаться на динамическое разрешение имен! (Фактически, локальные переменные уже определяются статически.)

В языке Python есть особенность: присваивание всегда производится имени в локальной области видимости, если перед этим не было указано явно (инструкция `global`), что переменная находится в глобальной области видимости. Присваивание не копирует данные — оно только привязывает имя к объекту. То же самое верно и для удаления: инструкция '`del` *x*' удаляет имя *x* из пространства имен, соответствующего локальной области видимости. В сущности, все операции, которые вводят новые имена, используют локальную область. Так, импортирование модуля и определение функции привязывают модуль или функцию к локальной области видимости.

9.3 Первый взгляд на классы

Классы требуют немного нового синтаксиса, добавляют три новых типа объектов и некоторое которое количество новой семантики.

9.3.1 Синтаксис определения класса

Простейшая модель определения класса выглядит следующим образом:

```
class имя_класса:
    инструкция1
    .
    .
    .
    инструкцияN
```

Определение класса, подобно определению функции (инструкция `class`, как и `def`), должно быть выполнено перед тем, как класс можно будет использовать. (Предполагается, что Вы можете поместить определение класса в одну из ветвей инструкции `if` или в тело функции.)

На практике инструкции внутри определения класса обычно являются определениями функций, но разрешены, и иногда полезны, другие инструкции. Определение функции внутри класса имеет специфическую форму записи списка аргументов, продиктованную соглашениями по вызову методов. Мы вернемся к этим особенностям позже.

Когда выполняется определение класса, создается новое пространство имен и используется в качестве локальной области видимости при выполнении инструкций в теле определения. Таким образом, все присваивания производятся именам из нового пространства имен. В частности, определение функции создает соответствующее имя в пространстве имен класса.

По окончании выполнения определения функции, создается *объект-класс*. По существу он является "оболочкой" пространства имен, созданного определением класса. В следующем разделе мы расскажем об объектах-классах более подробно. Исходная область видимости (которая была перед выполнением определения класса) восстанавливается, и объект-класс привязывается к имени класса (в приведенном примере — *имя_класса*) в пространстве имен, соответствующему исходной области видимости.

9.3.2 Объекты-классы

Объекты-классы поддерживают два вида операций: доступ к атрибутам и создание экземпляра класса.

Доступ к атрибутам объекта-класса осуществляется так же, как и для других объектов языка Python: *obj.attrname*. Действительными именами атрибутов являются все имена, помещенные в пространство имен класса при создании объекта-класса. Пусть определение класса выглядит следующим образом:

```
class MyClass:
    'Простой пример класса'
```

```
    i = 12345
    def f(x):
        return 'Привет всему миру'
```

Тогда i и f являются действительными атрибутами, ссылающимися на целое число и объект-метод соответственно. Атрибутам можно присваивать новые значения, например, Вы можете изменить значение MyClass.i. __doc__ также является действительным атрибутом, ссылающимся на строку документации класса: 'Простой пример класса'.

Создание экземпляра класса использует запись вызова функций. Просто считайте объект-класс функцией без параметров, возвращающей созданный экземпляр класса. Например,

```
x = MyClass()
```

создает новый *экземпляр* класса и присваивает его локальной переменной x.

В приведенном примере создается "пустой" объект. Во многих случаях необходимо создавать объект с определенным начальным состоянием — для этого класс должен содержать специальный метод __init__(), например:

```
class MyClass:
    def __init__(self):
        self.data = []
```

Если для класса определен метод __init__(), то он автоматически вызывается при создании каждого экземпляра этого класса.

Для большей гибкости метод __init__(), конечно, может иметь аргументы. В этом случае, аргументы, используемые при создании экземпляра класса, передаются методу __init__(). Например:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Объекты-экземпляры

Что мы теперь можем делать с объектами-экземплярами? Основная операция, воспринимаемая объектом-экземпляром — доступ к его атрибутам. Атрибуты могут быть двух

видов.

Первый — *атрибуты данных*. Они соответствуют “переменным экземпляра” в Smalltalk и “членам данных” в C++. Атрибуты данных не нужно декларировать: они возникают, когда им первый раз присваивают значение. Например, если `x` является экземпляром класса `MyClass`, определенного выше, следующий код выведет значение `16`:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Второй тип атрибутов — методы. Метод — это функция, “принадлежащая” объекту. В языке Python термин “метод” применим не только к экземплярам классов — другие объекты тоже могут иметь методы. Например, у объектов-списков есть методы `append()`, `insert()`, `remove()`, `sort()` и т. д. Однако ниже, если явно не указано другого, мы будем использовать термин “метод” для методов экземпляров (`instance method`).

Действительные имена методов объекта-экземпляра зависят от класса: все атрибуты класса, являющиеся объектами-функциями автоматически становятся методами при обращении к соответствующим атрибутам экземпляра[2]. Так, в нашем примере `x.f` является методом, привязанным к объекту `x`. Но `x.f` — это не то же самое, что и `MyClass.f`. В первом случае метод “знает” объект, к которому он применяется, во втором — он не привязан к какому-либо объекту и ведет себя аналогично функции.

9.3.4 Методы экземпляров классов

Обычно метод вызывают непосредственно:

```
x.f()
```

В нашем примере он вернет строку `'Привет всему миру'`. Однако совсем не обязательно вызывать метод прямо. `x.f` является объектом-методом, и его можно сохранить для дальнейшего использования:

```
xf = x.f
while 1:
    print xf()
```

[2]Функция может быть записана в lambda-форме, однако другие объекты, поддерживающие вызов (класс или экземпляр класса, для которого определен метод `__call__`) не подходят.

будет выводить `'Привет всему миру'` до тех пор, пока выполнение не будет прервано.

Что же происходит при вызове метода? Вы могли заметить, что `x.f()` вызывается без аргумента, несмотря на то, что в определении аргумент указан. Что же случилось с аргументом? Конечно, Python генерирует исключение, если функцию, требующую аргумент, вызвать без него — даже если аргумент на самом деле не используется.

Вы могли уже догадаться: особенность методов состоит в том, что объект, для которого он применяется, передается в качестве первого аргумента. В нашем примере вызов `x.f()` полностью эквивалентен `MyClass.f(x)`. В общем, вызов метода, привязанного к объекту, со списком из n аргументов полностью эквивалентен вызову соответствующего не привязанного метода (или функции) со списком аргументов, полученным добавлением объекта перед первым аргументом.

Если Вы все еще не поняли, как работают методы, возможно, взгляд на реализацию внесет ясность. Если Вы ссылаетесь на атрибут объекта, не являющийся атрибутом данных, то поиск атрибута производится в классе, экземпляром которого является объект. Если имя соответствует действительному атрибуту класса, который определен как функция, то атрибут считается методом. Метод является структурой, содержащей информацию о классе, в котором он определен, и функции, его представляющей, но "не знает", к какому именно объекту будет применяться (не привязанный к объекту метод). Соответствующий атрибут экземпляра класса, в свою очередь, содержит, помимо указанной информации, еще и ссылку на объект-экземпляр (привязанный к объекту метод) упакованные вместе в объекте-методе. При вызове метода со списком аргументов, он распаковывается, создается новый список аргументов из объекта-экземпляра и исходного списка аргументов, и с этим списком вызывается объект-функция.

9.4 Выборочные замечания

Атрибуты экземпляров (обычно атрибуты данных) записываются поверх атрибутов классов (обычно методов) с тем же именем. Чтобы избежать конфликтов имен, которые могут привести к тяжело обнаружимым ошибкам, полезно использовать своего рода соглашение, позволяющее минимизировать вероятность конфликтов. Например: называть методы именами, начинающимися с заглавной буквы, добавлять небольшую приставку в начало имен атрибутов данных (возможно просто символ подчеркивания) или использовать глаголы для методов и существительные для атрибутов данных.

В некоторых случаях изменение атрибутов данных напрямую, а не с помощью специально предназначенных для этого методов, может привести к порче инварианта объекта и непригодности его для дальнейшего использования. Вы можете "спрятать" данные и, тем самым, защитить их от случайного изменения. Для этого в языке существует соглашение: все атрибуты, имена которых содержат не менее двух символов подчеркивания в начале и не более одного символа подчеркивания в конце, считаются частными и доступны только из методов объекта. (На самом деле, Вы можете получить доступ к частным атрибутам извне, используя специальное имя, однако такой доступ никак не назовешь случайным. Более подробно работа с частными атрибутами описана в разделе 9.6.) С другой стороны, модули расширения, написанные на C, могут полностью

спрятать детали реализации и при необходимости полностью контролировать доступ.

Обычно первый аргумент в определении метода называют `self`. Это не более чем соглашение: имя `self` не имеет абсолютно никакого специального значения. Однако, следуя этому соглашению, Вы делаете код более понятным для других программистов. (Некоторые программы, например программы просмотра классов, рассчитаны на то, что пользователь всегда следует этому соглашению.)

Любой объект-функция, являющийся атрибутом класса, определяет метод экземпляров этого класса. Совсем не обязательно, чтобы определение функции находилось в определении класса: присваивание объекта-функции локальной переменной также будет работать. Например:

```
# Определение функции за пределами определения класса
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'Привет всему миру'
    h = g
```

Теперь `f`, `g` и `h` являются атрибутами класса `C` и ссылаются на объекты функции и, следовательно, все они будут методами экземпляров класса `C`. Вы можете также использовать функцию, определенную с помощью оператора `lambda`. Заметим, что такая практика обычно только сбивает с толку.

Методы могут вызывать другие методы, как атрибуты аргумента `self`, например:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Методы могут ссылаться на глобальные имена точно таким же способом, как и обычные функции. Глобальная область видимости, ассоциированная с методом — это глобальное пространство имен модуля, содержащего определение функции (так как определение функции обычно находится в определении класса, то — в глобальном пространстве имен модуля, содержащего определение класса). В то время как редко существует веская причина использовать глобальные данные в методе, глобальная область видимости все же находит множество разумных применений: использование импортированных в глобальную область видимости модулей, а также определенных в ней функций и классов.

Атрибуты классов ведут себя как статические атрибуты их экземпляров (то есть общие для всех экземпляров данного класса). Однако присвоить такому атрибуту новое значение Вы можете, только обратившись к нему как атрибуту того класса, в котором он определен (в противном случае Вы лишь создадите новый атрибут экземпляра с таким же именем).

9.5 Наследование

Конечно, особенность языка не достойна называться "классом" без поддержки наследования. Определение производного класса с именем *производный_класс* выглядит следующим образом:

```
class производный_класс(базовый_класс):
    инструкция1
    .
    .
    .
    инструкцияN
```

Базовый класс (*базовый_класс*) должен быть определен в области видимости, в которой находится определение производного класса (*производный_класс*). Вместо имени базового класса можно использовать выражение, например, если базовый класс определен в другом модуле:

```
class производный_класс(модуль.базовый_класс):
```

Определение производного класса выполняется так же, как и определение базового класса. Сконструированный объект-класс помнит базовый — он используется для разрешения имен атрибутов: если запрошенный атрибут не найден в классе, поиск продолжается в базовом классе. Это правило применяется рекурсивно, если базовый класс, в свою очередь, является производным от другого класса. В создании экземпляра производного класса нет ничего особенного: `производный_класс()` порождает новый экземпляр класса.

Производные классы могут переопределить методы базовых классов. Метод базового класса, вызывающего другой определенный для него метод, может, на самом деле, вызывать метод производного класса, переопределившего этот метод (говоря в терминах C++, *все* методы в языке Python являются *виртуальными*).

Переопределяя метод в производном классе, Вы можете также захотеть вызвать метод базового класса с тем же именем. Это можно сделать напрямую: просто вызовите '`базовый_класс.метод(self, список_аргументов)`', где *базовый_класс* — ссылка на базовый класс в том виде, в котором он доступен в текущей области видимости.

В языке Python есть ограниченная поддержка множественного наследования. Определения класса с несколькими базовыми классами выглядит следующим образом:

```
class производный_класс(базовый_класс1,
                        базовый_класс2,
                        базовый_класс3):
    инструкция1
    .
    .
    .
    инструкцияN
```

Единственное правило, необходимое для объяснения семантики, — правило разрешения имен атрибутов. Поиск производится сначала в глубину, затем слева направо. Таким образом, если атрибут не найден в *производный_класс*, то он ищется сначала в *базовый_класс1*, затем (рекурсивно) в базовых классах класса *базовый_класс1* и только потом в *базовый_класс2* и т. д. (Для некоторых людей кажется более естественным другой порядок разрешения имен атрибутов — сначала в классах *базовый_класс1*, *базовый_класс2*, *базовый_класс3* и только потом в базовых классах класса *базовый_класс1*. Однако в этом случае возникает зависимость результата от реализации каждого из базовых классов. С принятым же правилом, нет разницы между прямыми и унаследованными атрибутами базовых классов.)

9.6 Частные атрибуты

Python предоставляет ограниченную поддержку частных атрибутов классов. Любой атрибут вида __*атрибут* (имя которого содержит не менее двух символов подчеркивания в начале и не более одного в конце) заменяется на _*класс*__*атрибут*, где *класс* — имя текущего класса с "обрезанными" символами подчеркивания в начале. Такая обработка производится независимо от синтаксического расположения идентификатора, то есть может использоваться для определения частных атрибутов, доступ к которым будет возможен только из методов этого класса и методов его экземпляров. (Имя может быть обрезано, если его длина превысит 255 символов.) Если Вы ссылаетесь на имя, находясь за пределами класса, или если имя класса состоит только из символов подчеркивания, то оно преобразованию не подлежит.

Преобразование имен обеспечивает классам возможность определить "частные" атрибуты, не беспокоясь о том, что производные классы могут переопределить их, а также не дают к ним доступа коду, находящемуся за пределами класса. Заметьте, что правила преобразования разработаны в основном для того, чтобы избежать случайного вмешательства. Иногда доступ к частным атрибутам необходим, например, для отладчиков — это одна из причин, почему оставлена лазейка.

Если Вы из класса вызываете код с помощью `exec`, `execfile`, `eval()` или `evalfile()`, то внутри этого кода класс не будет считаться текущим: ситуация аналогична использованию инструкции `global` — действие ограничивается единовременно байт-компилированным кодом. Это ограничение распространяется и на `getattr()`, `setattr()` и `delattr()`, а также на прямое использование `__dict__`.

9.7 Примеры использования классов

Иногда полезно иметь тип данных (`record` в Pascal или `struct` в C), объединяющий несколько именованных единиц данных. С этой задачей прекрасно справится пустой класс:

```
class Employee:
    pass

# Создаем пустую карточку на служащего
john = Employee()

# Заполняем поля карточки:
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Использование экземпляров классов в качестве исключений позволяет расположить их в виде "дерева" и обрабатывать ошибки находящиеся на определенной ветви.

Часто вместо ожидаемого типа данных в функции (методе) можно использовать экземпляр класса, эмулирующего методы этого типа. Например, если есть функция, считывающая данные из файла, Вы можете определить класс с методами `read()` и `readline()`, которые будут брать данные из буфера вместо файла, и передать его экземпляр функции в качестве аргумента. Используя же специальные методы (см. раздел 11.6.3), можно эмулировать поведение чисел, списков, словарей и даже полностью контролировать доступ к атрибутам.

В библиотеке стандартных модулей Вы найдете множество примеров классов, эмулирующих поведение строк, списков, словарей, файлов. Рекомендуем посмотреть на реализацию таких модулей, как `UserString`, `UserList` и `UserDict`, `StringIO`. Кроме того, в дистрибутив обычно входит несколько демонстрационных модулей, среди которых Вы найдете много интересных примеров, показывающих, как, например, можно реализовать рациональные числа.

9.7.1 Экземпляры классов в качестве исключений

Исключения в языке Python могут быть строками (для совместимости со старыми версиями; поддержка строк в качестве исключений может быть удалена в будущих версиях) или экземплярами классов. Использование механизма наследования позволяет создавать легко расширяемые иерархии исключений.

Чаще всего инструкция `raise` употребляется в следующем виде:

```
raise экземпляр_класса
```

После ключевого слова `except` могут быть перечислены как строковые объекты, так и классы. Указанный класс считается "совместимым" с исключением, если исключение является экземпляром этого класса или класса, производного от него (но не наоборот — если в ветви `except` указан производный класс от того, экземпляром которого является исключение, то исключение не обрабатывается). Следующий пример выведет (именно в этом порядке) 'B', 'C', 'D':

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Обратите внимание, что если расположить ветви `except` в обратном порядке, то Вы получите 'B', 'B', 'B' — срабатывает первая ветвь `except`, совместимая с исключением.

Если исключение-экземпляр не обрабатывается, выводится сообщение об ошибке: имя класса, двоеточие, пробел и строка, полученная применением встроенной функции `str()` к исключению-экземпляру.

```
>>> class MyError:
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return `self.value`
...
>>> raise MyError(1)
Traceback (innermost last):
  File "<stdin>", line 1
__main__.MyError: 1
```

Язык имеет встроенный набор исключений, которые описаны в разделе 13. В качестве базового для всех исключений рекомендуется использовать класс `Exception` — это позволит полностью или частично избежать определения методов, необходимых для инициализации (метод `__init__()`) и вывода сообщения об ошибке (метод `__str__()`. В большинстве случаев определение нового исключения будет выглядеть совсем просто:

```
>>> class MyError(Exception): pass
...
>>> raise MyError('Oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: Oops!
```

9.7.2 Классы-помощники

Часто бывает полезно определить класс, помогающий выполнять какие-либо рутинные операции. Например, Вам часто необходимо перебирать параллельно элементы нескольких последовательностей, а поведение функции `map()` (см. раздел 5.2) Вас не устраивает или Вы работаете с длинными последовательностями и не хотите реально создавать последовательность пар (троек, и т. д.). Тогда Вы можете определить простой класс, создающий псевдопоследовательность:

```
class parallel:

    def __init__(self, *args):
        self.__args = args

    def __getitem__(self, item):
        return map(lambda s, i=item: s[i], self.__args)
```

С таким классом-помощником задача параллельного перебора элементов сильно упрощается:

```
>>> seq1 = xrange(10)
>>> seq2 = [1, 2, 3, 5, 7]
>>> for x, y in parallel(seq1, seq2):
...     print x, y
...
0 1
1 2
2 3
3 5
4 7
```

Как же наша псевдопоследовательность “узнает” о том, что элементы в одной из последовательностей закончились? Дело в том, что индикатором конца последовательности при использовании цикла `for` или средств функционального программирования в языке Python служит исключение `IndexError`. Исключение генерируется при первом использовании индекса, выходящего за пределы любой из последовательностей (в нашем примере `seq1` и `seq2`). Так что нам достаточно не обрабатывать его и инструкция `for` будет считать, что закончилась наша псевдопоследовательность.

В версии 2.0 языка появилась новая встроенная функции `zip()`, предназначенная специально для параллельного перебора нескольких последовательностей. Ее поведение аналогично приведенному здесь классу `parallel`, с одним лишь отличием — функция `zip()` создает полноценный список, в который можно вносить изменения.

9.7.3 Множества

По своей природе, множества находятся скорее ближе к словарям, чем к спискам. Так же, как и словари обеспечивают уникальность ключей, множества гарантируют уникальность входящих в него элементов. С другой стороны, списки обеспечивают порядок следования элементов, что для множеств совсем необязательно. Таким образом, встроенный тип `dictionary` может служить хорошей базой для реализации множеств. Ниже приведено примерное определение класса, реализующего множество.

```
class set:

    def __init__(self, seq = ()):
        # Атрибут '_data' содержит словарь, ключами
        # которого являются элементы множества. Делать
        # атрибут частным ('__data') нежелательно, так
        # как в этом случае будет сложно работать с
        # производными от set классами.
        if isinstance(seq, set):
            # Если seq является экземпляром set или
            # производного от него класса, можно
            # воспользоваться "секретами" реализации
            self._data = seq._data.copy()
        else:
            # Иначе мы считаем seq произвольной
            # последовательностью
            self._data = {}
            for item in seq:
                self._data[item] = None

    def items(self):
        # Этот метод позволит перебирать элементы
        # множества:
        # for item in myset.items():
        #     ...
        return self._data.keys()

    def tuple_key(self):
        # Сами множества изменяемые и не могут
        # использоваться в качестве ключа в словаре или
        # элемента в другом множестве. Для этого их
        # нужно преобразовать в кортеж.
```

```
        items = self._data.keys()
        # Сортируя элементы, мы можем гарантировать,
        # что порядок следования элементов в двух
        # множествах всегда будет одинаковым.
        items.sort()
        return tuple(items)

    def add(self, item):
        # Добавление элемента реализуется добавлением в
        # словарь пустой записи с соответствующим
        # ключем.
        self._data[item] = None

    def remove(self, item):
        if self._data.has_key(item):
            del self._data[item]
        else:
            # Множество не содержит такого элемента
            raise ValueError("item doesn't exist")

    def copy(self):
        return set(self)

    def __repr__(self):
        return 'set('+`self.items()`+')'

    def __len__(self):
        # Количество элементов в множестве, вызывается
        # функцией len(). Также определяет истинность
        # множества.
        return len(self._data)

    def __contains__(self, item):
        # Операторы 'in' и 'not in'.
        return self._data.has_key(item)

    def __cmp__(self, other):
        # Все операции сравнения.
        return cmp(self._data, other._data)

    def __or__(self, other):
        # Оператор '|' (объединение).
        res = set(self)
        res._data.update(other._data)
        return res

    def __ior__(self, other):
        # Оператор '|='.
```

```
        self._data.update(other._data)
        return self

    def __and__(self, other):
        # Оператор '&' (пересечение).
        # Будем перебирать элементы того множества,
        # которое содержит меньше элементов.
        if len(other._data) < len(self._data):
            self, other = other, self
        res = set()
        for item in self._data.keys():
            if other._data.has_key(item):
                res._data[item] = None
        return res

    def __iand__(self, other):
        # Оператор '&='.
        for item in self._data.keys():
            if not other._data.has_key(item):
                del self._data[item]
        return self

    def __sub__(self, other):
        # Оператор '-' (элементы, которые содержатся в
        # первом множестве и не содержатся во втором).
        res = set()
        for item in self._data.keys():
            if not other._data.has_key(item):
                res._data[item] = None
        return res

    def __isub__(self, other):
        # Оператор '-='.
        for item in other._data.keys():
            if self._data.has_key(item):
                del self._data[item]
        return self

    # Если мы реализуем вычитание, то для симметрии
    # следует также реализовать и сложение
    # (объединение).
    __add__ = __or__
    __iadd__ = __ior__

    def __xor__(self, other):
        # Оператор '^' (элементы, содержащиеся только в
        # одном из множеств).
```

```
        if len(other._data) < len(self._data):
            self, other = other, self
        res = set(other)
        for item in self._data.keys():
            if res._data.has_key(item):
                del res._data[item]
            else:
                res._data[item] = None
        return res

    def __ixor__(self, other):
        # Оператор '^='
        for item in other._data.keys():
            if self._data.has_key(item):
                del self._data[item]
            else:
                self._data[item] = None
        return self
```

9.7.4 Контроль доступа к атрибутам

С помощью специальных методов `__getattr__()`, `__setattr__()` и `__delattr__()` Вы можете контролировать все обращения к атрибутам экземпляра. Приведем пример класса, реализующего собственные методы `__getattr__()` и `__setattr__()` и сохраняющего все атрибуты в частной переменной:

```
class VirtualAttributes:
    __vdict = None
    # Таким образом мы можем получить преобразованное
    # имя атрибута __vdict:
    __vdict_name = locals().keys()[0]


    def __init__(self):
        # мы не можем записать 'self.__vdict = {}',
        # т.к. при этом произойдет обращение к методу
        # __setattr__
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

Часть II

Встроенные возможности языка

Эта часть книги является справочным руководством, описывающим встроенные возможности языка Python. Здесь описаны синтаксис и семантика различных конструкций языка (выражений, инструкций), встроенные типы данных — в общем то, что обычно рассматривается как “ядро” языка.

Здесь также описаны встроенные функции и исключения — объекты, использование которых не требует предварительного импортирования. Их имена содержатся в отдельном пространстве имен, поиск в котором происходит в последнюю очередь. Таким образом, пользователь может переопределять их в локальном или глобальном пространстве имен.

Глава 10

Синтаксис и семантика

В настоящей главе описаны синтаксис и семантика различных конструкций языка Python. Формальное определение грамматики приведено в приложении B.

10.1 Структура строк программы

10.1.1 Логические и физические строки

Программа на языке Python состоит из *логических строк*. Инструкции не могут быть разделены на несколько логических строк, за исключением составных инструкций. Логическая строка составляется из одной или нескольких *физических строк*, следуя правилам явного или подразумеваемого объединения строк. Отступ в физической строке, являющейся продолжением логической строки, не имеет значения.

Физическая строка заканчивается символом конца строки, принятым для данной платформы: ASCII символ LF для UNIX, последовательность CR LF для DOS и Windows, CR для Macintosh.

Комментарий начинается символом '#', не являющимся частью строкового литерала, и заканчивается в конце физической строки. Комментарии лексическим анализатором игнорируются.

Две или более физических строки могут быть объединены в логическую строку явно с помощью символа обратной косой черты ('\'): если физическая строка заканчивается символом обратной косой черты, который не является частью строкового литерала или комментария, она объединяется со следующей физической строкой, образуя одну логическую строку. Например:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:
        # Похоже на правильную дату
        return 1
```

Строка, заканчивающаяся символом обратной косой черты, не может содержать комментария. Лексемы не могут быть разделены на несколько строк, за исключением

строковых литералов. Использование символа обратной косой черты в любом другом месте, кроме как в конце строки, в строковом литерале и в комментарии, не допускается.

Выражения в обратных кавычках и скобках (круглых, квадратных, фигурных) могут быть разделены на несколько физических строк без использования символа обратной косой черты, например:

```
month_names = ['Январь',   'февраль',  # Названия
               'Март',     'Апрель',   # месяцев года
               'Май',      'Июнь',     # по-русски
               'Июль',     'Август',
               'Сентябрь', 'Октябрь',
               'Ноябрь',   'Декабрь']
```

Такие физические строки, объединяемые в одну логическую строку по правилам подразумеваемого объединения, могут содержать комментарии. Строковые литералы, записанные с использованием трех кавычек, также могут быть разделены на несколько физических строк, которые, однако, не могут содержать комментарии.

Логические строки, содержащие только символы пропуска (whitespace) и, возможно, комментарий, игнорируются. В интерактивном режиме пустая строка (без символов пропуска и комментариев) используется для обозначения конца составной инструкции.

10.1.2 Отступы

Символы пробела и табуляции в начале логических строк используются для определения уровня отступа строки. Уровень отступа строк используется для объединения инструкций в группы.

Сначала каждый символ табуляции (слева направо) заменяется на от одного до восьми пробелов таким образом, чтобы общее количество символов до этого места, включая замененные символы, было кратным восьми. Затем общее количество пробелов в начале логической строки используется для определения уровня отступа. Отступ не может быть разделен на несколько физических строк: для определения уровня отступа будут использованы символы пропуска до первого символа обратной косой черты. Только логические строки, имеющие одинаковое количество пробелов (после преобразования символов табуляции) в начале строки, считаются имеющими одинаковый уровень отступа.

Приведенный ниже пример имеет правильные, хотя и сбивающие с толку, отступы:

```
def perm(l):
        # Создает список всех возможных перестановок в l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
```

```
            s = l[:i] + l[i+1:]
            p = perm(s)
            for x in p:
             r.append(l[i:i+1] + x)
    return r
```

А следующий пример показывает возможные ошибки в отступах:

```
# ошибка: отступ в первой строке
     def perm(l):
# ошибка: нет отступа
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
# ошибка: неожиданный отступ
           p = perm(l[:i] + l[i+1:])
           for x in p:
                   r.append(l[i:i+1] + x)
# ошибка: при уменьшении отступа должна существовать
# строка с точно таким же отступом
               return r
```

Общепринято добавлять четыре пробела для каждого следующего уровня отступа. Использование символов табуляции не желательно, так как их интерпретация зависит от используемого редактора и его настроек. Большинство редакторов позволяют настроить автоматическую поддержку отступов. Если Вы работаете с редактором **vim**, достаточно добавить строчку '`autocmd FileType python set softtabstop=4 shiftwidth=4 expandtab autoindent`' в конфигурационный файл редактора '.vimrc' ('_vimrc' под Windows).

Символ подачи страницы (FF) может присутствовать в начале строки и игнорируется при определении отступа. В других местах отступа символ подачи страницы может давать неопределенный эффект (например, обнулить счетчик пробелов).

За исключением начала логической строки и строковых литералов, символы пробела, табуляции и подачи страницы могут быть взаимозаменяемо использованы для разделения лексем.

10.2 Выражения

Выражения состоят из атомов ("неделимых" частей), объединенных различными операторами. Описание операторов разделено на несколько подразделов, расположенных в порядке убывания приоритета описанных в них операторов (в конце этого раздела приведена сводная таблица приоритетов). Перед применением бинарных операторов операнды приводятся к общему типу по правилам, описанным в разделе 11.6.3. Назначение большинства операторов зависит от типа операндов и описано в соответствующих разделах главы 11.

10.2.1 Атомы

Идентификаторы

Идентификаторы являются ссылками на локальные, глобальные или встроенные имена. Если к идентификатору привязан объект в текущем блоке кода и в этом блоке не было явно указано с помощью инструкции `global`, что имя является глобальным, то идентификатор ссылается на локальное имя для этого блока. Если идентификатору в текущем блоке кода не был привязан объект, идентификатор ссылается на глобальное (если оно существует) или встроенное имя. Если же идентификатор явно указан в инструкции `global` в текущем блоке кода, то он всегда ссылается на глобальное имя.

Если к идентификатору привязан объект, его вычисление как атома в выражении дает этот объект. При попытке использования в выражении имени, не привязанного к объекту, будет сгенерировано исключение `NameError`.

Идентификаторы (имена) должны начинаться с буквы латинского алфавита или символа подчеркивания и содержать только буквы латинского алфавита, цифры и символы подчеркивания. Интерпретатор языка Python различает регистры букв: идентификаторы, например, `spam` и `Spam` считаются разными. Длина идентификаторов не ограничена.

Следующие идентификаторы являются зарезервированными (или ключевыми) словами языка и не могут быть использованы как обычные имена:

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass
def       finally   in        print
```

Помимо ключевых слов, некоторые идентификаторы имеют специальное значение (символ '`*`' обозначает "любое количество любых допустимых символов"; имена относятся к следующему классу, только если они не относятся к предыдущему):

`__*__`
Системные имена (особых модулей, специальных атрибутов и методов).

`__*` (только в определении класса)
Если идентификатор такого вида встречается в определении класса, то он считается частным атрибутом класса и подвергается специальному преобразованию. Например, имя '`__spam`', определенное в классе с именем '`Ham`', преобразуется к '`_Ham_spam`'. Преобразование производится независимо от контекста, в котором идентификатор используется. Если имя класса состоит только из символов подчеркивания, преобразование не производится.

_*

Частные объекты модуля, не импортируются инструкцией '`from` *`module`* `import *`'. В интерактивном режиме идентификатор с именем '`_`' используется для хранения результата вычисления последней инструкции-выражения, сохраняется в пространстве встроенных имен. Не в интерактивном режиме имя '`_`' не имеет специального значения (часто используется для выбора сообщений в соответствии с текущими национальными установками).

Литеральные выражения

Литеральные выражения являются записью значений некоторых встроенных типов: чисел различного типа (см. раздел 11.1), обычных строк (раздел 11.2.1) и строк Unicode (раздел 11.2.2).

"Замкнутые" выражения

Формы, заключенные в обратные кавычки и различные скобки также распознаются синтаксическим анализатором как атомы.

Круглые скобки представляют содержащееся в нем выражение и используются для контроля порядка вычисления. Обратите внимание, что кортеж (см. раздел 11.2.3) образуется не с помощью круглых скобок, а путем перечисления выражений через запятую. Скобки необходимы только для образования пустого кортежа, а также в тех случаях, когда перечисление через запятую используется синтаксисом для других целей (список аргументов функции или инструкции).

Квадратные и фигурные скобки используются для представления списков и словарей соответственно (см. разделы 11.2.6 и 11.3).

Обратные кавычки используются для получения строкового представления объекта, являющегося результатом вычисления заключенного в них выражения. Тот же результат Вы можете получить, используя встроенную функцию `repr()`. Смотрите также описание функции `repr()` (раздел 12) и специального метода `__repr__()` (раздел 11.6.3).

10.2.2 Первичные выражения

Первичное выражение может быть атомом или выражением, представляющим применение к первичному выражению (*`primary`*) следующих операций, имеющих наибольший приоритет:

`primary.attr`

Операция получения атрибута. Первичное выражение *`primary`* должно представ-

лять объект, поддерживающий доступ к атрибутам. Если атрибут *attr* объекта *primary* не доступен, генерируется исключение `AttributeError`.

primary[*key*]
: Операция получения элемента по индексу/ключу, характерная для последовательностей и отображений (см. разделы 11.2 и 11.3).

primary[*slice_list*]
: Операция получения подпоследовательности по срезу. Последовательности встроенного типа поддерживают лишь простую запись среза (*slice_list*) в виде '[*start*]:[*stop*]' (где *start* — нижняя и *stop* — верхняя границы). Расширенная запись (используется, например, в модулях расширения Numerical Python) позволяет указать через запятую несколько срезов вида '[*start*]:[*stop*][:[*step*]]' (*step* — шаг) или использовать троеточие ('...'). Например: 'x[1:10:2]', 'x[:10, ...]', 'x[5:, ::2]'. Расширенная запись среза представляется с помощью специальных объектов `slice` и `ellipsis`, описанных в разделе 11.8.3.

primary(*arg_list*)
: Операция вызова. Встроенные типы, поддерживающие вызов, перечислены в разделе 11.4. *arg_list* — список (через запятую) аргументов, которые могут быть переданы следующими способами (Вы можете использовать сразу несколько способов, однако порядок их следования в списке аргументов должен быть таким, в котором они здесь перечислены (*exprN* — произвольные выражения):

  expr1 [, ...]
  : Простой способ передачи позиционных аргументов. Благодаря автоматической упаковке/распаковке кортежей, Вы можете использовать (только при таком способе передачи аргументов) подсписки аргументов:

```
>>> def print_vector(s, (x, y, z)):
>>>     print s+':'
>>>     print 'x =', x
>>>     print 'y =', y
>>>     print 'z =', z
...
>>> velocity = 0.0, 1.0, 1.0
>>> print_vector('Вектор скорости', velocity)
Вектор скорости:
x = 0.0
y = 1.0
z = 1.0
```

  ***name1* = *expr1* [, ...]**
  : Передача именованных аргументов. *nameN* — идентификаторы (имена) аргументов, *exprN* — их значения. Порядок следования именованных аргументов не имеет значения.

  seq
  : Передача списка (*seq*) позиционных аргументов. Позиционные аргументы, переданные обычным способом, дополняются элементами последовательности (любого типа) *seq*. Такой способ передачи позиционных аргументов доступен, начиная с версии 1.6.

*******dict*

Передача словаря (*dict*) именованных аргументов. Именованные аргументы, переданные обычным способом, дополняются записями из словаря *dict*. Такой способ передачи именованных аргументов доступен, начиная с версии 1.6.

Операция вызова всегда возвращает значение (возможно `None`), если выполнение не было прервано (например, сгенерировано исключение).

10.2.3 Арифметические и битовые операторы

Ниже приведены все арифметические и битовые операторы в порядке уменьшения их приоритета. Операторы с одинаковым приоритетом объединены в группы.

x ****** *y*

Оператор возведения в степень имеет больший приоритет, чем унарный арифметический или битовый оператор слева, но меньший, чем унарный арифметический или битовый оператор справа (во избежание нелепых ошибок использование оператора `not` справа от любого арифметического или битового оператора не допускается). Так, например, выражение '`-x ** -y`' будет вычисляться справа налево, то есть эквивалентно выражению '`-(x ** (-y))`'.

+*x*, **-***x*, **~***x*

x ***** *y*, *x* **/** *y*, *x*%*y*

x **+** *y*, *x* **-** *y*

x **<<** *y*, *x* **>>** *y*

x **&** *y*

x **^** *y*

x **|** *y*

10.2.4 Условные операторы

Все условные операторы (операторы сравнения, проверки идентичности и вхождения) имеют одинаковый приоритет (больший, чем у логических операторов). Кроме того, условные операторы могут быть записаны в цепь, например, выражение '`x < y <= z`' интерпретируется так же, как это принято в математике, и эквивалентно '`x < y and y <= z`', за исключением того, что *y* вычисляется только один раз (но в обоих случаях *z* не вычисляется, если выражение '`x < y`' ложно).

Заметим, что конструкции типа '`a < b > c`' вполне допустимы, однако выглядят довольно неприятно (сравнение между *a* и *c* не производится). Операторы `!=` и `<>` полностью эквивалентны, использование первого предпочтительнее.

Сравнение

x **<** *y*
: 1, если *x* меньше *y*, иначе 0.

x **<=** *y*
: 1, если *x* меньше или равен *y*, иначе 0.

x **>** *y*
: 1, если *x* больше *y*, иначе 0.

x **>=** *y*
: 1, если *x* больше или равен *y*, иначе 0.

x **==** *y*
: 1, если *x* равен *y*, иначе 0.

x **<>** *y*

x **!=** *y*
: 0, если *x* равен *y*, иначе 1. <> и != — две альтернативные формы записи одного оператора, вторая форма предпочтительнее.

Операторы сравнения работают со всеми типами данных. Вы можете сравнивать объекты разного типа. Переменные численных типов равны, если равны их значения. Если Вы сравниваете обычную строку и строку Unicode, к обычной строке сначала применяется встроенная функция `unicode()` (считается, что строка имеет кодировку ASCII, если обычная строка содержит символы с кодом больше 127, генерируется исключение `UnicodeError`). Для экземпляров классов может быть определен специальный метод `__cmp__()` (иногда также будет полезен метод `__rcmp__()`, см. раздел 11.6.3), который будет использоваться для реализации операторов <, <=, >, >=, ==, != (<>), в том числе, и с объектами других типов (следует помнить, что перед выполнением операций сравнения производится попытка привести операнды к общему типу). Во всех остальных случаях производится проверка идентичности: объекты равны, только если они являются одним и тем же объектом. Порядок следования при сортировке объектов, для которых не определена операция сравнения, определяется типом объектов и их идентификаторами[1].

Идентичность

Операторы `is` и `is not` проверяют идентичность объектов: являются ли операнды на самом деле одним и тем же объектом (сравниваются идентификаторы объектов; см. также описание функции `id()` в разделе 12).

x **is** *y*
: 1, если *x* и *y* ссылаются на один и тот же объект, иначе 0.

[1]Возможны ситуации, при которых *obj1 < obj2* и *obj2 < obj3*, но *obj1 > obj3*, например, если один из объектов является экземпляром, для которого не определен метод `__cmp__()`.

x **is not** *y*
: `0`, если *x* и *y* ссылаются на один и тот же объект, иначе `1`.

Вхождение

Еще два условных оператора — `in` и `not in` — предназначены для определения, является ли объект элементом последовательности (или любого другого контейнера) и подробно описаны в разделе 11.2.

x **in** *y*
: `1`, если *y* содержит элемент, который при сравнении (оператор `==`) оказывается равным *x*, иначе `0`.

x **not in** *y*
: `1`, если *y* не содержит элемента равного *x*, иначе `0`.

10.2.5 Истинность

Для любого объекта можно проверить его истинность при использовании в инструкциях `if` и `while` или в качестве операнда логических операторов, описанных ниже. Следующие значения считаются ложью:

- `None`;
- ноль любого числового типа, например, `0`, `0L`, `0.0`, `0j`;
- пустая последовательность, например, `''`, `()`, `[]`;
- пустое отображение (словарь) — `{}`;
- экземпляр класса, для которого определен метод `__nonzero__()` или `__len__()`, если он возвращает ноль. Дополнительную информацию об этих специальных методах Вы можете получить в разделе 11.6.3.

Все остальные значения считаются истинными — так что многие объекты всегда истинны.

10.2.6 Логические операторы

Ниже приведены логические операторы в порядке уменьшения приоритета. Заметим, что оператор `not` имеет меньший приоритет, чем арифметические и условные операторы. То есть '`not` *a* `==` *b*' интерпретируется как '`not (`*a* `==` *b*`)`', а выражение '*a* `== not` *b*' является синтаксической ошибкой.

Операторы `or` и `and` всегда возвращают один из своих операндов. Причем второй операнд операторов `or` и `and` вычисляется, только если это необходимо для получения результата.

not *x*
: Если *x* ложно, то 1, иначе 0.

x **and** *y*
: Если *x* ложно, то *x*, иначе *y*.

x **or** *y*
: Если *x* ложно, то *y*, иначе *x*.

10.2.7 Оператор `lambda`

lambda *param_list*: *expression*
: Короткая форма создания (безымянной) функции, возвращающей значение выражения *expression*. Ее поведение аналогично поведению функции, созданной с помощью инструкции '`def` *name*(*param_list*): `return` *expression*'. Синтаксис списка параметров описан в разделе 10.4.5.

Обратите внимание, что функция, создаваемая с помощью оператора `lambda`, не может содержать инструкций. Кроме того, `lambda`-функция (как и функция, определенная с помощью инструкции `def`) не имеет доступа к локальному пространству имен блока, в котором она определена. Однако Вы можете обойти это ограничение, используя аргументы, имеющие значения по умолчанию, например:

```
def make_incrementor(increment):
    return lambda x, n=increment: x+n
```

`lambda` имеет наименьший приоритет среди операторов, однако его приоритет больше чем у конструкции, образующей список выражений. Так, например, выражение '`lambda: 1, 2`' эквивалентно выражению '`(lambda: 1), 2`' и образует кортеж из двух элементов: функции, возвращающей `1`, и числа `2`.

10.2.8 Списки выражений

Список выражений (аргументов) образуется путем перечисления выражений (идентификаторов) через запятую. Списки аргументов функции и выражений, образующий кортеж, может содержать завершающую запятую, однако она не имеет никакого значения, за исключением одного случая: завершающая запятая необходима для образования кортежа, содержащего один элемент (см. раздел 11.2.3). Использование завершающей запятой в списке аргументов инструкций, кроме инструкции `print`, не допускается. Завершающая запятая в списке аргументов инструкции `print` указывает на то, что не нужно выводить символ перехода на новую строку (см. раздел 10.3.5).

10.2.9 Сводная таблица приоритетов

Ниже перечислены все конструкции и операторы, которые могут быть использованы для создания выражений, в порядке уменьшения приоритета. Конструкции, перечисленные в одном пункте, имеют одинаковый приоритет.

1. Атомы: идентификаторы, литеральные и "замкнутые" (‘(*expression* ...)’, ‘[*expression* ...]’, ‘{*key*: *value* ...}’, ‘`` `*expression*` ``’) выражения.
2. Первичные выражения: `x.attribute`, `x[key]`, `x[slice_list]`, `x(arg_list)`.
3. `**` (см. замечания в разделе 10.2.3).
4. Унарные операторы `+`, `-` и `~`.
5. `*`, `/`, `%`.
6. Бинарные операторы `+` и `-`.
7. `<<`, `>>`.
8. `&`.
9. `^`.
10. `|`.
11. Условные операторы: `<`, `<=`, `>`, `>=`, `==`, `!=`, `<>`, `is [not]`, `[not] in`.
12. `not`.
13. `and`.
14. `or`.
15. `lambda`.
16. *expr1*, ... (образование списка выражений).

10.3 Простые инструкции

Простые инструкции записываются в одну логическую строку. Логическая строка может содержать несколько простых инструкций разделенных точкой с запятой.

10.3.1 Инструкции-выражения

Инструкции-выражения обычно используются для вычисления и вывода значения выражения (в интерактивном режиме) или для вызова "процедур" — функций, возвращающих незначимый результат (`None`) и состоят из одного выражения (список выражений также является выражением).

В интерактивном режиме значение выражения, если оно не равно `None`, преобразуется в строку аналогично тому, как это делает встроенная функция `repr()` и выводится на стандартный поток вывода. Значение `None` не выводится — таким образом вызов процедур не дает никакого вывода.

10.3.2 Присваивание

Присваивание ('*lvalue = expression*') используется для того, чтобы связать идентификатор (существующий или новый) с объектом, для создания и изменения атрибутов объектов, изменения элементов изменяемых последовательностей, добавления и изменения записей в отображениях. Присваивание может производиться (выражения, которые могут быть использованы слева от знака равенства, обычно называют *lvalue*):

name **=** *expression*
: Идентификатору.

x.*attribute* **=** *expression*
: Атрибуту. Обычно имя *attribute* в пространстве имен объекта *x* связывается с объектом, являющимся результатом вычисления выражения *expression*. Вы можете контролировать поведение при присваивании атрибуту экземпляра класса, определив для класса специальный метод `__setattr__()`.

x[*key*] **=** *expression*
: Элементу последовательности или отображения. Изменение элементов экземпляров классов реализуется специальным методом `__setitem__()`.

x[*slice_list*] **=** *expression*
: Подпоследовательности (синтаксис списка срезов *slice_list* описан в разделе 10.2.2). Выражение *expression* должно быть последовательностью. Присваивание срезу экземпляра контролируется с помощью специального метода `__setslice__()` (если используется простая запись среза) или `__setitem__()` (если используется расширенная запись срезов).

lvalue, ... **=** *expression*
(*lvalue*, ...**)** **=** *expression*
[*lvalue*, ...**]** **=** *expression*
: Конструкции, используемой для создания кортежей и списков (в том числе, произвольным образом вложенных) из всех перечисленных выражений. Операция присваивания выполняется путем распаковки последовательностей. Результат вычисления выражения *expression* должен быть последовательностью, содержащей

такое же количество элементов, какое указано слева от знака равенства. Каждому вложенному списку слева в результате должна соответствовать подпоследовательность с таким же количеством элементов. При несоответствии количества элементов генерируется исключение `ValueError`. Приведем несколько примеров:

```
>>> x = range(1)
>>> y, = range(1)
>>> print x, y
[0] 0
>>> class c:
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):
...         return 'c('+`self.val`+')'
...
>>> o = c(1)
>>> l = range(3)
>>> l[1], [(x, y), o.val] = [10, (xrange(2, 4), 5)]
>>> print x, y, l, o
2 3 [0, 10, 2] c(5)
```

Значение выражения справа от = (включая упаковку) вычисляется до того, как начинается присваивание. Таким образом, инструкция '`a, b = b, a`' обменивает значения переменных `a` и `b`. Однако следующий пример выведет '`[0, 2]`':

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

Начиная с версии 2.0, возможно одновременное выполнение арифметической или битовой операции и присваивания:

```
lvalue += expression
lvalue -= expression
lvalue *= expression
lvalue /= expression
lvalue %= expression
lvalue **= expression
lvalue &= expression
lvalue |= expression
lvalue ^= expression
lvalue >>= expression
lvalue <<= expression
```

Если *lvalue* является неизменяемым объектом, инструкция '*lvalue op= expression*' всегда эквивалентна последовательному применению оператора и присва-

иванию результата (нового объекта) *lvalue* ('*lvalue* = *lvalue op expression*'). Здесь, в отличие от обычного присваивания, *lvalue* не может быть кортежем.

Заметим, что формально присваивание в языке Python не является оператором, то есть, не возвращает значения и не может быть использовано в выражениях. Так сделано, например, чтобы избежать ошибочного использования = вместо == в инструкции `if`.

10.3.3 Инструкция `del`

Семантика инструкции '`del` *lvalue*' во многом схожа с семантикой присваивания (синтаксис *lvalue* описан в разделе 10.3.2). Удаление идентификатора удаляет привязку имени (которое должно существовать) в локальном или глобальном пространстве имен (см. раздел 10.5) к объекту. При удалении атрибута обычно удаляется соответствующее имя из пространства имен объекта. Вы можете контролировать поведение при удалении атрибутов экземпляров, определив для класса специальный метод `__delattr__()`. Удаление элементов (записей) из экземпляров-последовательностей (экземпляров-отображений) реализуется специальным методом `__delitem__()`, подпоследовательности — специальным методом `__delslice__()` (если используется простая запись среза) или `__delitem__()` (при использовании расширенной записи срезов).

10.3.4 Пустая инструкция

Инструкция `pass` ничего не делает и может быть полезна там, где инструкция требуется синтаксисом, однако нет необходимости в выполнении каких либо действий, например:

```
# функция, которая ничего не делает
def do_nothing(arg): pass

# пустой класс
class record: pass
```

10.3.5 Инструкция `print`

Инструкция '`print` [*expr_list*]' для каждого выражения в списке выражений *expr_list* вычисляет его значение и выводит строковое представление значения выражения на стандартный поток вывода, разделяя их пробелами (необходимость вывода пробела контролируется атрибутом `softspace` файлового объекта). Строковое представление получается аналогично тому, как это делает встроенная функция `str()` (см. раздел 12).

Если список выражений не заканчивается запятой, в конце выводится символ перехода на новую строку. Это единственное действие, которое выполняется, если ин-

струкция содержит только ключевое слово `print`.

В качестве стандартного потока вывода используется файловый объект, на который ссылается переменная `stdout` в модуле `sys`. Если имя `sys.stdout` было удалено, генерируется исключение `RuntimeError`.

Начиная с версии 2.0, инструкция `print` может быть использована для вывода в любой файл. Для этого после ключевого слова `print` необходимо поставить >> и первое выражение в списке *expr_list* должно давать файловый объект или `None` (в этом случае для вывода будет использоваться `sys.stdout`). Например:

```
print >> sys.stderr, 'Ошибочка вышла!'
```

Инструкция `print` предъявляет следующие требования к файловому объекту, в который производится вывод: объект должен иметь метод `write()` и доступный для записи атрибут `softspace` (последнее всегда верно для экземпляров классов, не имеющих специальных методов `__getattr__()` и `__setattr__()`).

10.3.6 Инструкция break

Инструкция `break` может быть использована только (синтаксически) внутри циклов `for` и `while`, но не в определении функции или класса, находящегося в цикле. Прерывает выполнение самого вложенного цикла (пропуская ветвь `else` цикла, если она присутствует). При прерывании цикла `for` переменная (или переменные), в которой хранится значение текущего элемента, остается неизменной.

Если для выхода из цикла необходимо покинуть блок инструкции `try` с ветвью `finally`, перед выходом выполняется ветвь `finally`.

10.3.7 Инструкция continue

Инструкция `continue` может быть использована только (синтаксически) внутри циклов `for` и `while`, но не в определении функции или класса, находящегося в цикле. В текущих версиях инструкция `continue` также не может быть использована внутри основного блока инструкции `try` (расположенного синтаксически внутри цикла) — только в ветвях `except` и `else`[2]. Инструкция продолжает выполнение наиболее вложенного цикла со следующего прохода.

[2]Это ограничение, возможно, будет снято в будущих версиях языка.

10.3.8 Инструкция `return`

Инструкция '`return` [*`expression`*]' (может быть использована синтаксически только в определении функции) прерывает выполнение функции (при ее вызове) используя значение выражения *`expression`* (`None`, если выражение опущено) в качестве возвращаемого функцией значения.

Если для возврата из функции необходимо покинуть блок инструкции `try` с ветвью `finally`, перед возвратом выполняется ветвь `finally`.

10.3.9 Инструкция `global`

Инструкция '`global` *`identifier_list`*' указывает на то, что идентификаторы *`identifier_list`* (список через запятую идентификаторов) во всем текущем блоке ссылаются на глобальные переменные. В то время как использование в выражениях глобальных имен производится автоматически, если они не определены (нигде в текущем блоке) в локальном пространстве имен, присваивание глобальным именам без использования инструкции `global` невозможно.

Если имена где-либо в данном блоке кода указаны в инструкции `global`, их следует использовать только после этой инструкции. Такие имена не должны использоваться как формальные параметры циклов `for`, определений классов и функций и не должны быть среди имен, импортируемых инструкцией `import`. В текущих версиях инструкции `for`, `class` и `def` изменяют глобальную переменную, а инструкция `import` не импортирует такие имена, однако в будущих версиях поведение может измениться.

Инструкция `global` является директивой синтаксическому анализатору команд и применяется только к единовременно компилируемому блоку кода. В частности, инструкция `global`, содержащаяся в инструкции `exec` не оказывает влияния на блок кода, в котором содержится эта инструкция `exec`. И наоборот, инструкция `global` в блоке кода, содержащем также инструкцию `exec`, не оказывает влияние на код в этой инструкции `exec`. То же самое верно и для встроенных функций `eval()`, `execfile()` и `compile()`.

10.3.10 Инструкция `import`

Существует три формы записи инструкции `import`:

`import` *`module_list`*

Импортирует указанные модули. *`module_list`* — список через запятую полных имен модулей (если модуль находится в пакете — используется запись через точку, например, *`пакет.подпакет.модуль`*) с возможным указанием имен, к которым будут привязаны объекты-модули в локальном пространстве имен (см. ниже).

`from` *`module`* `import` *`obj_list`*
: Импортирует объекты из модуля *module*. *obj_list* — список через запятую идентификаторов объектов (идентификатор объекта может ссылаться, в том числе, и на объект-модуль, но, в отличие от имени модуля, не может использовать запись через точку) с возможным указанием имен, к которым будут привязаны объекты в локальном пространстве имен (см. ниже).

`from` *`module`* `import *`
: Импортирует все имена из модуля *module*, кроме имен, начинающихся с символа подчеркивания.

Инструкция `import` выполняется в два этапа:

1. Интерпретатор находит модуль (или модули) и, если это необходимо, инициализирует его.
2. Связывает имена в текущем локальном пространстве имен с объектами-модулями (первая форма записи инструкции) или объектами, на которые ссылаются соответствующие имена в модуле.

Начиная с версии 2.0, Вы имеете возможность указать имя, к которому будет привязан объект в локальном пространстве имен, используя запись '*`object`* `as` *`local_name`*' в списке модулей (*`module_list`*) или объектов (*`obj_list`*). Так, например, инструкция '`import operator`' привязывает импортированный объект-модуль к локальному имени `operator`, в то время как '`import operator as op`' привязывает тот же объект-модуль к локальному имени `op`. Во избежание недоразумений не допускается импортирование под другим именем объектов, для которых используется точечная запись: вместо '`import xml.sax as s`' необходимо использовать запись '`from xml import sax as s`'.

Под инициализацией модуля, реализованного на языке Python, подразумевается создание пустого объекта-модуля, добавление его в таблицу `sys.modules` и выполнение в контексте этого модуля инструкций из файла '***module*.py**' (или '**__init__.py**' в каталоге '***module***', если модуль является пакетом). Точнее, выполняется его байт-компилированная версия — файл '***module*.pyc**' или '***module*.pyo**' в зависимости от того, используется оптимизация или нет. Если соответствующий байт-компилированный файл отсутствует или не соответствует исходному '**.py**'-файлу, интерпретатор компилирует исходный файл (если он содержит синтаксические ошибки, генерируется исключение `SyntaxError` и объект-модуль в `sys.modules` не добавляется), пытается записать байт-компилированную версию и выполняет его (независимо от успешности записи байт-компилированной версии). Не перехваченные исключения во время инициализации прерывают процесс импортирования модуля. Однако частично инициализированный модуль остается в таблице `sys.modules`. Инициализация динамически подгружаемого или встроенного в интерпретатор модуля выполняется путем вызова функции `init`*`module`*.

Система запоминает модули, которые были инициализированы в словаре (индексированном именами модулей) `sys.modules`. При импортировании, если имя модуля найдено в `sys.modules`, его инициализация пропускается. Если Вы хотите заново

инициализировать модуль (например, чтобы не покидать интерпретатор после каждого внесения изменений в модуль при отладке), воспользуйтесь встроенной функцией `reload()`.

Поиск модулей производится сначала среди встроенных в интерпретатор, затем в путях, хранящихся в списке `sys.path`. `sys.path` при запуске интерпретатора содержит текущий каталог, список каталогов из переменной окружения PYTHONPATH и зависящие от платформы пути по умолчанию. Если модуль с указанным именем не найден, генерируется исключение `ImportError`.

Импортируемые имена не должны использоваться в инструкции `global` в той же области видимости. В текущих реализациях такие имена просто не будут импортированы, однако в будущих версиях поведение может измениться.

Запись '`from ... import *`' рекомендуется использовать только в глобальной области видимости.

Вы можете изменить поведение инструкции `import`, переопределив встроенную функцию `__import__()`. При этом будут полезны средства, определенные в модуле `imp`.

10.3.11 Инструкция `exec`

Инструкция '`exec` *`expression`* [`in` *`globals`*, [*`locals`*]]' предназначена для динамического выполнения кода. Выражение *`expression`* должно давать объект типа `code`, `string` или `file` (открытый для чтения). Если *`expression`* является объектом кода, инструкция `exec` просто выполняет его. Строка или файл считаются содержащими инструкции языка Python, которые должны быть выполнены (если не содержат синтаксических ошибок).

Выражения *`globals`* и *`locals`* должны давать словари, которые будут использованы как глобальное и локальное пространства имен. Если аргументы опущены, код выполняется в текущей области видимости. Если указан только аргумент *`globals`*, он используется в качестве глобального и локального пространств имен. В текущих реализациях в эти словари автоматически добавляется запись (если такой нет) с ключом `'__builtins__'`, ссылающаяся на словарь пространства встроенных имен (то есть, модуля `__builtin__`).

Динамическое вычисление выражений осуществляется с помощью встроенной функции `eval()`. Также могут быть полезны встроенные функции `globals()` и `locals()`, которые возвращают словари, соответствующие текущим глобальному и локальному пространствам имен.

В текущих реализациях многострочные составные инструкции должны заканчиваться переходом на новую строку. Так, '`exec 'for i in (1,2):\n\tprint i\n'`' будет работать нормально, но при попытке выполнить '`exec 'for i in (1,2):\n\tprint i'`' будет сгенерировано исключение `SyntaxError`.

10.3.12 Отладочные утверждения

Общепринятый способ включения в программу отладочных утверждений — инструкция '`assert` *expression* [, *exc_arg*]'. Если выражение *expression* ложно, генерируется исключение `AssertionError`. Если задан необязательный второй аргумент *exc_arg*, то он используется в качестве аргумента, передаваемого конструктору исключения. При компиляции с оптимизацией инструкции `assert` игнорируются.

Простая форма '`assert` *expression*' эквивалентна конструкции (предполагается, что идентификаторы `__debug__` и `AssertionError` ссылаются на встроенные переменные с такими именами; встроенная переменная `__debug__` равна `1` при отключенной оптимизации)

```
if __debug__:
    if not expression: raise AssertionError()
```

Расширенная форма '`assert` *expression*, *exc_arg*' эквивалентна конструкции

```
if __debug__:
    if not expression: raise AssertionError(exc_arg)
```

В качестве аргумента *exc_arg* можно, например, использовать строку, поясняющую возможную причину невыполнения утверждения. Заметим, что нет необходимости включать в нее исходный текст выражения *expression*, так как он в любом случае будет включен в выводимую (если исключение не перехвачено) отладочную информацию.

10.3.13 Генерация исключений

Инструкция '`raise` [*expr1* [, *expr2* [, *expr3*]]]' следует использовать в случае возникновения ошибок и других исключительных ситуаций. Без аргументов инструкция `raise` повторно генерирует последнее исключение, сгенерированное в текущей области видимости.

Иначе вычисляется значение выражения *expr1*, которое должно быть строкой, классом или экземпляром класса. Если задан второй аргумент (выражение *expr2*), вычисляется его значение, в противном случае подставляется `None`. Если первый аргумент является объектом-классом и второй аргумент является экземпляром этого или производного от него класса, *expr2* используется в качестве генерируемого исключения. Если же второй аргумент не является таким экземпляром, он используется для инициализации класса *expr1* следующим образом: если выражение *expr2* является кортежем, то оно используется в качестве списка аргументов; если *expr2* равно `None`, класс инициализируется с пустым списком аргументов; в остальных случаях *expr2* используется как единственный аргумент при инициализации. Если же первый аргумент инструкции

raise является экземпляром класса, он используется в качестве исключения (второй аргумент должен быть равен None или опущен).

Поддержка строк в качестве исключений существует лишь для совместимости со старыми версиями и может быть удалена в будущих версиях языка. И, тем не менее, если первый аргумент является строкой, он используется в качестве исключения. В этом случае второй аргумент (или None) является ассоциированным с исключением значением.

Третий аргумент, если задан и не равен None, должен быть объектом traceback (см. раздел 11.9.3). В этом случае он используется вместо объекта, создаваемого для текущего положения, для указания на место возникновения исключительной ситуации.

10.4 Составные инструкции

Составные инструкции содержат другие инструкции и каким-либо образом управляют их выполнением. Обычно составные инструкции записываются в несколько логических строк с использованием отступов, хотя, если составная инструкция имеет только одну ветвь, тело которой содержит только простые инструкции, она может быть записаны в одну логическую строку.

Инструкции if, while и for реализуют традиционные средства управления логикой программы. Инструкция try устанавливает обработчик исключений. Определения функций и классов также являются составными инструкциями.

Составная инструкция состоит из одной или нескольких ветвей. Каждая ветвь состоит из заголовка и тела. Заголовки ветвей одной составной инструкции должны иметь одинаковый уровень отступа. Заголовок каждой ветви начинаются уникальным ключевым словом и заканчиваются двоеточием. Тело является набором инструкций, управляемых данной ветвью. Тело может содержать одну или несколько разделенных точкой с запятой простых инструкций, записанных в одной логической строке с заголовком (сразу после двоеточия), или одну или несколько произвольных инструкций на непосредственно следующих за заголовком стоках, имеющих по сравнению с заголовком дополнительный отступ. Только вторая форма записи тела ветви может содержать составные инструкции. Следующий пример является недопустимым, в первую очередь потому, что не очевидно, к какой из инструкций if должна была бы относиться ветвь else:

```
if test1: if test2: print x
```

Логическая строка не может содержать несколько инструкций, если хотя бы одна из них является составной. В следующем примере все инструкции print входят в тело основной ветви инструкции if:

```
if x < y < z: print x; print y; print z
```

Обратите внимание, что каждая ветвь любой составной инструкции всегда заканчиваются переходом на новую строку. Кроме того, во избежание неоднозначностей, дополнительные ветви всегда начинаются с ключевого слова, которое никогда не используется в начале инструкций. Характерная для таких языков, как C, проблема отнесения ветви `else` решается в языке Python требованием правильного использования отступов во вложенных составных инструкциях.

10.4.1 Инструкция `if`

Инструкция `if` используется для выполнения различных действий в зависимости от условий:

```
if expr1: suite1
[elif expr2: suite2...]
[else: suite0]
```

Инструкция `if` вычисляет одно за другим выражения *exprN* в основной ветви и необязательных ветвях `elif` до тех пор, пока не будет получена истина, и выполняет соответствующий ему блок кода (вся оставшаяся часть инструкции `if` пропускается). Если все выражения дают ложь, выполняется блок кода *suite0* необязательной ветви `else`.

10.4.2 Цикл `while`

Цикл `while` используется для многократного выполнения действий, пока выполняется условие:

```
while expr: suite1
[else: suite2]
```

Инструкция `while` выполняется в следующем порядке:

1. Вычисляется выражение *expr* и, если оно дает ложь, переходит к пункту 3.

2. Выполняется блок кода *suite1*. При этом выполнение в блоке инструкции `break` немедленно прерывает выполнение цикла (пункт 3 не выполняется), а выполнение инструкции `continue` прерывает выполнение блока кода, после чего выполнение цикла продолжается с пункта 1. После окончания выполнения блока *suite1*, выполнение цикла продолжается с пункта 1.

3. Выполняется блок кода *suite2* (ветвь `else`). Инструкции `break` и `continue` в этом блоке считаются относящимися к внешнему циклу.

10.4.3 Цикл for

Инструкция for используется для перебора элементов последовательности:

```
for lvalue in sequence: suite1
[else: suite2]
```

Выражение *sequence* вычисляется один раз и должно давать последовательность (объект, для которого определена операция получения элемента по индексу; если индекс выходит за пределы диапазона, должно генерироваться исключение IndexError). Для каждого элемента последовательности *sequence* в порядке возрастания индексов, начиная с 0, выполняется присваивание *lvalue* элемента (на элемент *item* и параметр *lvalue* распространяются обычные правила, действующие при выполнении инструкции присваивания '*lvalue = item*'; см. раздел 10.3.2) и выполняется блок кода *suite1*. После того как последовательность исчерпана (определяется по сгенерированному исключению IndexError), если присутствует ветвь else, выполняется блок *suite2*.

Выполнение инструкции break в блоке *suite1* немедленно прерывает выполнение цикла (ветвь else игнорируется). При выполнении инструкции continue в блоке *suite1* пропускается остаток блока и выполнение цикла продолжается после присваивания *lvalue* следующего элемента последовательности *sequence* или выполняется ветвь else, если в последовательности нет следующего элемента.

Вы можете присваивать переменной или переменным, входящим в *lvalue*, — это никак не отразится на следующем элементе, который будет присвоен *lvalue*. Переменные, входящие в *lvalue* не удаляются после окончания выполнения цикла и сохраняют последние присвоенные им значения. Однако, если последовательность *sequence* пустая, *lvalue* не будет присвоено никакое значение (таким образом, используя эти переменные без дополнительной проверки, Вы имеете шанс получить ошибку NameError). См. также описания встроенных функций range() и xrange() в главе 12.

Будьте внимательны, изменяя последовательность, элементы которой Вы перебираете. Текущий элемент последовательности определяется значением внутреннего счетчика. Конец последовательности определяется по исключению IndexError при попытке получить очередной элемент. Таким образом, если в блоке *suite1* Вы удалите текущий или один из пройденных элементов последовательности, следующий элемент будет пропущен. Аналогично, если вы добавите в последовательность новый элемент перед текущим, тело цикла для текущего элемента будет выполнено еще раз. Такое поведение может привести к неприятным ошибкам, которых обычно можно избежать, если перебирать элементы копии исходной последовательности, например:

```
for x in a[:]:
    if x < 0: a.remove(x)
```

10.4.4 Инструкция `try`

Существует две формы инструкции `try`. Их нельзя смешивать (то есть, инструкция `try` не может содержать одновременно ветви `except` и `finally`), однако могут быть вложенными друг в друга.

С помощью первой формы Вы определяете обработчики исключений:

```
try: suite1
except [exc_class1 [, lvalue1]]: suite2
[...]
[else: suite0]
```

Сначала выполняется тело основной ветви — *suite1*, и если код не генерирует исключение, выполняется блок *suite0* (необязательная ветвь `else`). Если же код в блоке *suite1* генерирует исключение, последовательно проверяются (в порядке их следования) ветви `except` до тех пор, пока не будет найдена ветвь `except`, удовлетворяющая сгенерированному исключению. Ветвь `except` без аргументов, если присутствует, должна быть последней ветвью `except` (в противном случае генерируется исключение `SyntaxError`) — удовлетворяет всем исключениям. Если указано выражение *exc_classN*, оно вычисляется и ветвь считается удовлетворяющей исключение, если полученный объект совместим с исключением. Объект *exc_classN* является совместимым с исключением, если исключение является экземпляром класса *exc_classN* или производного от него класса или является строкой равной *exc_classN*. Объект *exc_classN* также является совместимым с исключением, если он является кортежем, содержащим объект, совместимый с исключением (при этом кортеж может содержать вложенные кортежи).

Если ни одна из ветвей `except` не удовлетворяет исключению, поиск обработчика продолжается во внешнем блоке `try` (если такой имеется) и т. д. Если исключение генерируется при вычислении выражения *exc_classN*, обработка исходного исключения прекращается и производится поиск обработчика нового исключения во внешнем блоке (как будто вся инструкция `try` сгенерировала исключение).

Если найдена ветвь `except`, удовлетворяющая исключению, исключение (если оно является экземпляром класса) или ассоциированное с ним значение (если исключение является строкой) присваивается параметру *lvalueN* (любое выражение, которому допускается присваивание — см. раздел 10.3.2), если он задан, выполняется блок кода ветви (*suiteN*) и выполнение продолжается инструкций следующих за инструкцией `try` (исключение считается обработанным). Таким образом, если имеется две вложенных инструкции `try` и исключительная ситуация возникла в блоке `try` внутренней инструкции, обработчик внешней инструкции не будет обрабатывать исключение.

Перед выполнением блока ветви `except` информация об исключении сохраняется и может быть получена с помощью функции `sys.exc_info()` — функция возвращает кортеж, состоящий из типа исключения (класс или строка), самого исключения (если

оно является экземпляром класса) или ассоциированного значения (если исключение является строкой) и объекта `traceback` (см. раздел 11.9.3). Эти же значения сохраняются в `sys.exc_type`, `sys.exc_value` и `sys.exc_traceback` соответственно. Однако их не рекомендуется использовать, так как это небезопасно в многопоточных программах.

Необязательная ветвь `else` выполняется, если в блоке основной ветви не было сгенерировано исключение. Если при выполнении ветви `except` или `else` генерируется исключение, поиск обработчика производится во внешней инструкции `try` (если такая имеется), а не в текущей.

Вторая форма используется для определения “страховочного” кода:

```
try: suite1
finally: suite2
```

Сначала выполняется ветвь `try` (блок *suite1*). Если при этом не возникает исключительной ситуации, далее просто выполняется ветвь `finally` (блок *suite2*). Если же в блоке *suite1* генерируется исключение, то оно временно сохраняется, выполняется ветвь `finally` и сохраненное исключение генерируется снова (поиск обработчика производится во внешней инструкции `try`, если такая имеется). Если Вы прерываете выполнение блока *suite2* инструкцией `return`, `break` или `continue` или генерируя новое исключение, сохраненное исключение будет утрачено (то есть не будет сгенерировано снова).

Ветвь `finally` выполняется и в том случае, если выход из блока *suite1* происходит по инструкции `return` или `break`. В текущих реализациях использование инструкции `continue` в основной ветви инструкции `try` не допускается (ситуация, возможно, изменится в будущих версиях).

10.4.5 Определение функций

С помощью инструкции `def` Вы можете определить объект-функцию (`function`, см. раздел 11.4.1):

```
def func_name(param_list): suite
```

param_list — список параметров (через запятую; если параметры записаны с использованием только первой и второй форм, список может содержать завершающую запятую, которая игнорируется) в следующих формах (различные формы можно смешивать, но каждая из них может использоваться только один раз и порядок их следования должен быть именно таким, в котором они перечислены):

param1 [, *param2* ...]

Список простых позиционных параметров. Каждый из параметров может

быть идентификатором или подсписком параметров в виде '(*subparam1* [, *subparam2* ...])' (подсписки аргументов также могут содержать вложенные подсписки).

param1 = *expr1* [, *param2* = *expr2*...]
: Список позиционных аргументов *paramN*, имеющих значение по умолчанию, получаемое вычислением выражения *exprN*. Форма записи параметров такая же, как и для первой формы записи аргументов. Выражения *exprN* вычисляются один раз — при выполнении определения функции, в области видимости, которое содержит определение функции.

**indentifier*
: Получение произвольного числа позиционных аргументов. Переменной *indentifier* присваивается кортеж всех позиционных аргументов, которым не соответствует параметры, заданные с использованием первых двух форм.

***indentifier*
: Получение произвольного количества именованных аргументов. Переменной *indentifier* присваивается словарь всех именованных аргументов, которым не соответствуют параметры, заданные с использованием первых двух форм.

При каждом выполнении определения функции (инструкции `def`) создается объект-функция и привязывается к имени *func_name* (идентификатор) в локальном пространстве имен (не рекомендуется использовать в качестве *func_name* идентификатор, который указан в этом блоке в инструкции `global` — см. раздел 10.3.9). Создаваемый объект-функция содержит ссылку на текущее глобальное пространство имен, которое будет использоваться в качестве глобального пространства имен при вызове функции.

Определение функции не выполняет тело функции (*suite*) — оно выполняется только при вызове функции. Аргументы, соответствующие параметрам со значением по умолчанию, могут быть опущены. В этом случае подставляется значение по умолчанию.

Синтаксис и семантика вызова функций описана в разделе 10.2.2. При вызове значения присваиваются всем параметрам, указанным в списке: это может быть значение позиционного или именованного аргумента или значение по умолчанию. Параметры, записанные в виде **identifier* и ***identifier*, если присутствуют, инициализируются пустыми кортежем и словарем соответственно.

Для создания простой анонимной функции вы можете воспользоваться оператором `lambda` (см. раздел 10.2.7).

Функция (в том числе, и созданная с помощью оператора `lambda`) не имеет доступа к локальному пространству имен блока, в котором она определена. Чтобы обойти это ограничение, часто используют значения аргументов по умолчанию:

```
# Возвращает функцию, которая возвращает аргумент,
# увеличенный на n
def make_incrementer(n):
```

```
    def increment(x, n=n):
        return x+n
    return increment

add1 = make_incrementer(1)
print add1(3)  # Выведет '4'
```

Как мы уже упоминали, значения по умолчанию вычисляются только один раз. Это особенно важно, если значение по умолчанию является изменяемым объектом (например, списком или словарем): если функция изменяет объект (например, добавляя элемент в список), изменяется значение по умолчанию. Обычно это не то, что Вы ожидаете. Один из способов обойти такое поведение — использовать в качестве значения параметра по умолчанию объект `None` и явно проверять его в теле функции:

```
def add_ostrich_notice(notices=None):
    if notices is None:
        notices = []
    notices.append(
        "Просьба страусов не пугать - пол бетонный")
    return notices
```

Однако использовать такое поведение для определения статических переменных нежелательно — лучше использовать глобальные переменные модуля или определить класс со специальным методом `__call__()`, например:

```
# Для вспомогательных объектов используем частные
# имена, которые не будут импортироваться с помощью
# 'from ... import *'
_MAX_KICKS_ALLOWED = 10  # Число разрешенных пинков

class _dog_kicker:
    def __init__(self):
        # ведет себя как статическая переменная в
        # "функции" kick_a_dog()
        self.num_of_kicks = 0
    def __call__(self, num=1):
        if self.num_of_kicks >= _MAX_KICKS_ALLOWED:
            print "А Вам собаку не жалко?"
            return
        num = min(_MAX_KICKS_ALLOWED-self.num_of_kicks,
                  num)
        self.num_of_kicks = self.num_of_kicks+num
        print "Вы пнули собаку", num, "раз(а)"

# Этот экземпляр можно использовать аналогично функции
kick_a_dog = _dog_kicker()
```

10.4.6 Определение класса

Определение класса создает объект-класс (class, см. раздел 11.6.1):

```
class class_name[(base_classes)]: suite
```

Здесь *base_classes* — список выражений (см. раздел 10.2.8), каждое из которых должно давать объект-класс. Для класса создается новое локальное пространство имен, глобальное пространство имен используется текущее (то есть того блока кода, в котором находится определение класса) и в этом окружении выполняется блок *suite* (обычно содержит только определения функций). После выполнения блока *suite* локальное пространство имен класса (используется для создания словаря атрибутов класса) и список базовых классов (значения выражений, входящих в *base_classes*) сохраняется в созданном объекте-классе и объект-класс привязывается к имени *class_name* в текущем локальном пространстве имен.

Имена, определенные в пространстве имен класса являются атрибутами класса — общими для всех экземпляров класса. Для того, чтобы определить атрибут экземпляра класса (имя в пространстве имен экземпляра класса), необходимо присвоить ему значение в одном из методов (например, при инициализации экземпляра — в методе `__init__()`). И атрибуты класса, и атрибуты экземпляра доступны с использованием записи через точку (*instance.name*) — атрибуты экземпляра "прячут" атрибуты класса с таким же именем. Атрибуты класса, значение которых равно неизменяемому объекту, могут служить как значения по умолчанию для атрибутов экземпляров этого класса.

10.5 Пространства имен

Пространство имен — отображение имен (идентификаторов) к объектам. По функциональности пространства имен эквивалентны словарям и реализуются в виде словарей.

В языке Python всегда присутствуют три пространства имен: локальное, глобальное и пространство встроенных имен. Поиск локальных имен всегда производится в локальном пространстве имен, глобальных — сначала в глобальном, затем в пространстве встроенных имен[3].

Является ли имя локальным или глобальным определяется в момент компиляции: в отсутствии инструкции global, имя, добавляемое где-либо в блоке кода, является локальным во всем блоке; все остальные имена считаются глобальными. Инструкция global позволяет заставить интерпретатор считать указанные имена глобальными.

[3]Если блок программы содержит инструкцию exec или 'from ... import *', семантика локальных имен изменяется: сначала производится их поиск в локальном пространстве имен, затем в глобальном и пространстве встроенных имен.

Имена могут быть добавлены (только в локальное пространство имен) следующими способами:

- передача формальных аргументов функции,
- использование инструкции import,
- определение класса или функции (добавляет в локальное пространство имен имя класса или функции),
- использование оператора присваивания,
- использование цикла for (в заголовке указывается новое имя),
- указывая имя во второй позиции после ключевого слова except.

Если глобальное имя не найдено в глобальном пространстве имен, его поиск производится в пространстве встроенных имен, которое, на самом деле, является пространством имен модуля __builtin__. Этот модуль (или словарь определенных в нем имен) доступен под глобальным именем текущего блока __builtins__. Если же имя не найдено в пространстве встроенных имен, генерируется исключение NameError.

Ниже приведены значения локального и глобального пространств имен для различных типов блоков кода. Пространство имен модуля автоматически создается, когда модуль импортируется первый раз.

Тип блока	**Глобальное пространство имен**	**Локальное пространство имен**
Модуль (программа и интерактивные команды фактически являются модулем __main__)	пространство имен модуля	то же, что и глобальное
Класс	пространство имен модуля, в котором находится определение класса	новое пространство имен (пространство имен класса), создается в момент выполнения определения
Функция (если объект кода, представляющий тело функции, выполняется с помощью инструкции exec, Вы можете сами определить, какие пространства имен использовать)	пространство имен модуля, в котором находится определение функции	новое пространство имен (пространство имен функции), создается каждый раз при вызове функции
Выражение, вычисляемое с помощью функции input()	глобальное пространство имен блока, из которого вызывается функция	локальное пространство имен блока, из которого вызывается функция

Тип блока	Глобальное пространство имен	Локальное пространство имен
Строка, файл или объект кода, выполняемые с помощью инструкции `exec`, или функций `eval()` или `execfile()` (используя необязательные аргументы, Вы можете сами определить, какие пространства имен использовать)	по умолчанию используется глобальное пространство имен блока, в котором выполняется инструкция или из которого вызывается функция	по умолчанию используется локальное пространство имен блока, в котором выполняется инструкция или из которого вызывается функция

Встроенные функции `globals()` и `locals()` возвращают словарь, представляющий глобальное и локальное пространства имен соответственно (см. главу 12).

Глава 11

Встроенные типы данных

В этом разделе описываются стандартные типы данных, поддержка которых встроена в интерпретатор: числа, последовательности и другие, включая сами типы. В языке Python нет явного логического типа — используйте целые числа.

Некоторые операции поддерживаются всеми объектами; в частности, любые объекты можно сравнивать, проверять их истинность (см. разделы 10.2.4 и 10.2.5) и для любых объектов можно получить строковое представление с помощью встроенных функций `repr()` (или используя запись в обратных кавычках: `` `expression` ``) и `str()`.

Объекты встроенных типов могут иметь следующие специальные атрибуты:

`__methods__`
Список имен статических методов объекта. Атрибут определен только, если объект имеет хотя бы один такой метод.

`__members__`
Список имен статических атрибутов данных объекта. Атрибут определен только, если объект имеет хотя бы один такой атрибут.

При каждом обращении к атрибутам `__methods__` и `__members__` создается новая копия списка.

11.1 Числовые типы

В языке Python есть четыре типа чисел: целые (`int`), длинные целые (`long int`), с плавающей точкой (вещественные; `float`) и комплексные (`complex`).

Числа создаются с использованием литералов или как возвращаемый результат встроенной функции или оператора. Числовые литералы в чистом виде (включая шестнадцатеричные и восьмеричные) дают простые целые числа.

Python полностью поддерживает смешанную арифметику: если бинарный арифметический оператор имеет операнды разного типа, операнд “меньшего” типа приводится к типу другого операнда (`int < long int < float < complex`). Это же правило используется для сравнения чисел разного типа[1]. Для приведения чисел к нужному типу

[1] В результате, например, списки `[1, 2]` и `[1.0, 2.0]` считаются равными.

Вы можете использовать встроенные функции `int()`, `long()`, `float()` и `complex()`. При приведении числа с плавающей точкой к целому (простому или длинному) типу, число округляется в сторону нуля (см. также функции `floor()` и `ceil()` в модуле `math`).

11.1.1 Целые и длинные целые числа

Целые числа реализованы с использованием типа `long` в C, поддерживающего числа в диапазоне не менее чем от $-2\,147\,483\,647$ до $2\,147\,483\,647$. Длинные целые имеют неограниченную точность.

Литералы целых чисел могут быть записаны в десятичном, восьмеричном и шестнадцатеричном виде. Десятичная запись состоит из идущих подряд десятичных цифр (`0`–`9`), причем первая цифра не может быть нулем. Восьмеричная запись образуется из нуля и следующих за ним восьмеричных цифр (`0`–`7`). Шестнадцатеричная запись образуется из приставки '`0x`' или '`0X`' и следующих за ней шестнадцатеричных цифр (`0`–`9`, `a`–`z`, `A`–`Z`). Знак '`-`' или '`+`' не является частью литерала, а лишь унарным оператором, применяемым к объекту после его создания. Если литерал представляет число, выходящее за пределы диапазона допустимых чисел, генерируется исключение `OverflowError`.

Целые литералы с суффиксом '`L`' или '`l`' дают длинные целые (лучше использовать '`L`', из-за схожести символа '`l`' с единицей).

Приведем несколько примеров литералов для целых и длинных целых чисел:

```
7   2147483647                      0177   0x80000000
3L  79228162514264337593543950336L  0377L  0x100000000L
```

11.1.2 Вещественные числа

Вещественные числа (числа с плавающей точкой) реализованы с использованием типа `double` в C (не менее 10 значащих цифр, наибольшее представимое число не меньше 10^{37}).

Запись литералов вещественных чисел состоит из десятичных цифр (первая цифра не может быть нулем) и содержит десятичную точку и/или экспоненциальную часть. Экспоненциальная часть начинается с символа '`e`' или '`E`', далее следует необязательный знак ('`+`' или '`-`') и одна или несколько десятичных цифр.

Приведем несколько примеров литералов для чисел с плавающей точкой:

```
3.14    10.    .001    1e100    3.14e-10
```

Поведение в случаях, когда результат вычисления выражения не может быть представлен в рамках типа `float`, зависит от поведения лежащей в основе библиотеки языка C. Обычно существуют значения, представляющие (отрицательные и положительные)

бесконечно малые и бесконечно большие числа. Строковое представление таких чисел зависит от версии интерпретатора и используемой для реализации библиотеки языка C, например, они могут выводиться как `1.` или `-1.` с добавлением комментария `#INF`, `#IND` или `#QNAN`.

11.1.3 Комплексные числа

Комплексные числа имеют мнимую и вещественную части, реализованные с использованием типа `double` в C. Литералы мнимого числа создаются путем добавления суффикса '`j`' или '`J`' к десятичной записи целого или вещественного числа. Например:

```
3.14j   10.j    10j     .001j   1e100j  3.14e-10j
```

Комплексные числа с ненулевой вещественной частью создаются сложением вещественной и мнимой частей или с помощью встроенной функции `complex()` (см. главу 12).

Комплексные числа имеют два атрибута данных и один метод:

real
Действительная часть комплексного числа.

imag
Мнимая часть комплексного числа.

conjugate()
Возвращает комплексное число, сопряженное с данным.

Преобразование комплексных чисел к другим числовым типам неоднозначно: для получения действительной части используйте атрибут `real`, а абсолютной величины — встроенной функцией `abs()`.

11.1.4 Арифметические операторы

Ниже приведены арифметические операторы, применимые ко всем числам:

x **+** *y*
Сумма *x* и *y*.

x **-** *y*
Разница между *x* и *y*.

x ***** *y*
Произведение *x* на *y*.

x **/** *y*
: Частное *x* и *y*. Если оба операнда целые (простые или длинные), то результат тоже будет целым. Результат всегда округляется в меньшую сторону: `1/2` и `-1/-2` даст `0`, а `-1/2` и `1/-2` даст `-1`.

x **%** *y*
: Остаток от деления *x* на *y*.

-*x*
: *x* с противоположным знаком.

+*x*
: *x* (без изменений).

x ****** *y*
: *x* в степени *y*.

Кроме того, для работы с числами Python предоставляет встроенные функции `abs()`, `divmod()` и `pow()` (см. главу 12), а также более сложные функции, определенные в модуле `math`.

11.1.5 Битовые операции над целыми числами

Для простых и длинных целых чисел в языке Python определены битовые операторы. Приоритет у всех бинарных битовых операторов ниже, чем у арифметических, и выше, чем у операторов сравнения. Унарный оператор ~ имеет такой же приоритет, как и унарные арифметические операторы (+ и −).

Ниже приведены все битовые операторы:

x **|** *y*
: Битовое ИЛИ *x* и *y*.

x **^** *y*
: Битовое исключающее ИЛИ *x* и *y*.

x **&** *y*
: Битовое И *x* и *y*.

x **<<** *n*
: Биты *x*, сдвинутые на *n* влево, эквивалентно умножению на `2**`*n* без контроля переполнения. Сдвиг на отрицательное число бит не допускается (генерируется исключение `ValueError`).

x **>>** *n*
: Биты *x*, сдвинутые на *n* вправо, эквивалентно (целочисленному) делению на `2**`*n* без контроля переполнения. Сдвиг на отрицательное число бит не допускается (генерируется исключение `ValueError`).

~*x*
: Биты *x* инвертированные.

11.2 Последовательности

В языке Python есть пять встроенных типов последовательностей: `string`, `unicode`, `tuple`, `xrange`, `buffer` и `list`.

Ниже приведены операции, определенные для всех последовательностей. Операторы `in` и `not in` имеют такой же приоритет, как и операторы сравнения, а операторы + (бинарный) and `*` — такие же, как и соответствующие арифметические операторы[2]. Выражения *s* и *t* — последовательности одинакового[3] типа, *n*, *i* и *j* — целые.

len(*s*)
: Возвращает число элементов последовательности *s*.

x **in** *s*
: `1`, если элемент *x* содержится в последовательности *s*, иначе — `0`.

x **not in** *s*
: `0`, если элемент *x* содержится в последовательности *s*, иначе — `1`

s **+** *t*
: Объединение последовательностей *s* и *t*.

s ***** *n*
n ***** *s*
: *n* копий последовательности *s* объединенных вместе. Отрицательные значения *n* воспринимаются как `0` (что дает пустую последовательность такого же типа, как и *s*).

s **+=** *t*
s ***=** *t*
: Эквивалентно '*s* = *s* + *t*' и '*s* = *s* * *n*' соответственно, но, если *s* является изменяемой последовательностью, то выполнение операции будет производиться без создания нового объекта, то есть переменная *s* будет ссылаться на один и тот же объект до и после операции.

s[*i*]
: *i*-й элемент последовательности *s*, отсчет начинается с `0`. Если индекс *i* имеет отрицательное значение, отсчет ведется с конца, то есть подставляется `len(`*s*`) + `*i*. Однако `-0` означает `0`. Если индекс *i* указывает за пределы последовательности, генерируется исключение `IndexError`.

s[[*i*]:[*j*]]
: Срез последовательности *s* от *i* до *j* — подпоследовательность, содержащая элементы последовательности *s* с такими индексами *k*, которые удовлетворяют неравенству *i* <= *k* < *j*. Если *i* или *j* имеет отрицательное значение, отсчет ведется с конца, то есть подставляется `len(`*s*`) + `*i* или `len(`*s*`) + `*j*. Индексы *i* и *j* могут быть опущены: *i* по умолчанию равен `0`, *j* — `sys.maxint`. Если (после всех преобразований) первый индекс больше или равен второму, возвращается пустая последовательность.

[2]Это необходимо, так как синтаксический анализатор не может определить тип операндов.

[3] Последовательности, определяемые пользователем, могут не иметь этого ограничения.

Для работы с последовательностями также могут быть полезны встроенные функции `min()` и `max()` (см. главу 12).

11.2.1 Строки

В этом разделе описаны свойства обычных строк. Однако они в основном характерны и для строк Unicode.

Строковые литералы

Строки могут быть заключены в одинарные (‘'’) или двойные (‘"’) кавычки (открывающая и закрывающая кавычки должны быть одинаковыми) — так называемые короткие строки. Для длинных строк более удобна другая запись — строка, заключенная в группы из трех одинарных или трех двойных кавычек (открывающая и закрывающая группы должны быть одинаковыми). Короткие строки могут содержать управляющие последовательности и любые символы, кроме обратной косой черты (‘\’), символов перехода на новую строку и кавычек, в которые строка заключена. Длинные строки дополнительно могут содержать символы перехода на новую строку и любые кавычки, хотя и не могут содержать группу из трех кавычек, в которые строка заключена.

В строках можно использовать следующие управляющие последовательности:

Последовательность	**Представляемый символ**
`\`*`newline`*	Игнорируется (*`newline`* — символ новой строки).
`\\`	Символ обратной косой черты (‘\’).
`\'`	Одинарная кавычка (‘'’).
`\"`	Двойная кавычка (‘"’).
`\a`	Символ оповещения (BEL).
`\b`	Символ возврата на одну позицию (backspace, BS).
`\f`	Символ подачи страницы (formfeed, FF).
`\n`	Символ перехода на новую строку (linefeed, LF).
`\r`	Символ возврата каретки (CR).
`\t`	Символ горизонтальной табуляции (TAB).
`\v`	Символ вертикальной табуляции (VT).
`\`*`ooo`*	ASCII символ с восьмеричным кодом, равным *`ooo`*.
`\x`*`hh...`*	ASCII символ с шестнадцатеричным кодом, равным *`hh...`*.

В соответствии со стандартом C, в последних двух управляющих последовательностях воспринимается до трех восьмеричных цифр и неограниченное количество шестна-

дцатеричных цифр соответственно (на платформах с 8-битным типом `char` используются только младшие 8 бит). В отличие от стандарта C, все нераспознанные управляющие последовательности остаются в строке (включая символ обратной косой черты). Такое поведение очень удобно при отладке.

Перед открывающей кавычкой (или группой из трех кавычек) в строковом литерале может быть указана буква ‘r’ или ‘R’. Такие строки называют “необрабатываемыми”, для них используются другие правила обработки управляющих последовательностей: строка всегда будет состоять из тех символов, которые записаны между открывающей и закрывающей кавычками (или группами кавычек). Так, `r"\n"` даст строку, состоящую из двух символов — обратной косой черты и латинской буквы ‘n’, `r"\""` также является правильным строковым литералом и представляет строку, состоящую из символов обратной косой черты и двойной кавычки. Обратите внимание, что строка не может заканчиваться на нечетное количество символов обратной косой черты: последний из них образует с закрывающей кавычкой управляющую последовательность. Например, `r"\"` не является правильным строковым литералом.

Несколько следующих друг за другом строковых литералов, возможно разделенных символами пропуска (whitespace), использующие различную запись (заключены в различные кавычки; обычные строки и необрабатываемые), автоматически объединяются. Таким образом, строка `'Привет' "всем"` является эквивалентом строки `'Приветвсем'`. Строки объединяются даже в том случае, если одна из них является обычной строкой, а другая строкой Unicode (`'a' u'b' == u'ab'`). Эта особенность позволяет уменьшить количество используемых символов обратной косой черты при записи длинного строкового литерала в несколько строк программного кода и даже снабдить части строк комментариями:

```
re.compile("[A-Za-z_]"       # буква или символ
                             # подчеркивания
           "[A-Za-z0-9_]*"   # буква, цифра или символ
                             # подчеркивания
          )
```

Заметим, что объединение строковых литералов выполняется в момент компиляции, в то время как оператор + объединяет строки во время выполнения программы.

Оператор форматирования

Оператор `%` с объектом-строкой (`string` или `unicode`) в качестве левого аргумента имеет особое значение. Он воспринимает левый аргумент как строку формата, которую необходимо применить к правому аргументу (аналогично функции `sprintf()` в C), и возвращает полученную в результате форматирования строку.

Правый аргумент должен быть кортежем, содержащим по одному элементу на каждый аргумент, требуемый строкой формата. Если строке формата необходим один аргумент, правым аргументом оператора `%` может быть также требуемый строкой фор-

мата объект, если он не является кортежем[4].

Строка форматирования может содержать обычные символы (кроме символа '%'), которые копируются без изменения, и описания формата. Каждое описание формата имеет вид: '%[*флаг*...][*ширина*][.*точность*]*символ_формата*'[5].

После символа '%' могут быть указаны следующие флаги:

Флаг	Назначение
#	Используется альтернативное представление аргумента. Для форматов 'o', 'x' и 'X' результат будет начинаться с '0', '0x' и '0X' соответственно. При использовании форматов 'f', 'g' и 'G' результат всегда будет содержать десятичную точку, даже если после нее не следует не одной цифры. Кроме того, для форматов 'g' и 'G' из результата не будут удалены завершающие нули. В остальных случаях результат остается неизменным.
0	Результат заполняется слева нулями до нужной ширины поля. Для 's'- и '%'-форматов игнорируется.
–	Результат выравнивается влево (по умолчанию результат выравнивается вправо).
пробел	Перед положительным числом вставляется пробел при использовании знаковых форматов.
+	Перед числом всегда ставится знак ('+' или '-') при использовании знаковых форматов.

Следующие два необязательных параметра — *минимальная ширина поля* и *точность*. Если представление значения содержит меньше символов, чем ширина поля, то оно будет дополнено пробелами (нулями). Точность задает минимальное количество цифр при использовании форматов 'd', 'i', 'o', 'u', 'x' и 'X' (по умолчанию 1), число цифр после десятичной точки для форматов 'e', 'E' и 'f' (по умолчанию 6), максимальное количество значащих цифр для форматов 'g' и 'G' (по умолчанию 6; 0 воспринимается как 1), максимальное количество символов из строки для формата 's' игнорируется. Вместо того, чтобы прямо указывать ширину поля и/или точность, Вы можете использовать символ '*'. В этом случае соответствующее целое значение передается в кортеже аргументов:

```
>>> import math
>>> R = 1
>>> print "Длина окружности равна %*.*f" % (
...         5,             # Ширина поля
...         2,             # Точность
...         2*math.pi*R)   # Форматируемое число
Длина окружности равна  6.28
```

[4]Кортеж в этом случае должен содержать один элемент.

[5]Перед символом формата может быть также указан модификатор длины ('h', 'l' или 'L'), однако он игнорируется.

Отрицательная ширина поля воспринимается как флаг '-' и следующее за ним положительное число. Если значение после '.' не указано или оно отрицательное, точность считается равной нулю.

Интерпретатор Python поддерживает следующие символы формата:

Символ	Назначение
d, i	Десятичное представление целого числа (знакового)
o	Восьмеричное представление целого числа без знака
u	Десятичное представление целого числа без знака
x	Шестнадцатеричное представление целого числа без знака. Используются буквы в нижнем регистре (abcdef).
X	Шестнадцатеричное представление целого числа без знака. Используются буквы в верхнем регистре (ABCDEF).
e	Экспоненциальное представление вещественного числа. Экспоненциальная часть обозначается буквой 'e'.
E	Экспоненциальное представление вещественного числа. Экспоненциальная часть обозначается буквой 'E'.
f	Представление вещественного числа. Если точность равна 0, десятичная точка не используется. Во избежание вывода бесконечных строк бессмысленных цифр, точность ограничена числом 50.
g	Если порядок вещественного числа меньше −4 или больше или равен точности, используется 'e'-формат, в противном случае используется 'f'-формат. Завершающие нули из дробной части удаляются.
G	Если порядок вещественного числа меньше −4 или больше или равен точности, используется 'E'-формат, в противном случае используется 'f'-формат. Завершающие нули из дробной части удаляются.
r	Строковое представление объекта, полученное аналогично применению встроенной функции repr(). Поддерживается, начиная с версии 1.6.
s	Вставка строки или строкового представления объекта, полученного аналогично применению встроенной функции str().
%	Символ '%'.

Если тип значения для ширины поля или точности не является int (при использовании '*'), генерируется исключение TypeError. Остальные аргументы, не являющиеся строками и тип которых не соответствует используемому формату, интерпретатор пытается привести к необходимому типу с помощью соответствующих встроенных функций. Если это не удается, то генерируется исключение TypeError (для встроенных типов) или AttributeError (для объектов типа instance).

В качестве правого аргумента оператора % можно использовать словарь (или любое другое отображение). В этом случае описания формата сразу после символа '%' должны

содержать заключенный в скобки ключ, соответствующий необходимому значению в словаре. Например:

```
>>> count = 2
>>> language = 'Python'
>>> print 'В языке %(language)s %(count)03d типа ' \
          'кавычек.' % vars()
В языке Python 002 типа кавычек.
```

Однако при такой записи Вы не можете передать ширину поля или точность в качестве аргумента (то есть, использовать '*' в описании формата).

Методы строк

Начиная с версии 1.6, строки (обычные и Unicode) имеют набор методов для работы с ними. В более ранних версиях Вы можете воспользоваться функциями, определенными в стандартном модуле `string`. Далее *s* — строка, к которой применяется метод.

Форматирование

center(*width*)
: Возвращает строку длиной *width*, в центре которой расположена исходная строка (центрирует строку в поле заданной ширины). Строка дополняется до нужной длины пробелами. Если *width* меньше длины исходной строки, она возвращается без изменений.

ljust(*width*)
: Возвращает копию исходной строки, дополненную справа пробелами (выравнивает строку влево в поле заданной ширины). Если *width* меньше длины исходной строки, она возвращается без изменений.

rjust(*width*)
: Возвращает копию исходной строки, дополненную слева пробелами (выравнивает строку вправо в поле заданной ширины). Если *width* меньше длины исходной строки, она возвращается без изменений.

Поиск вхождений

count(*sub* [, *start* [, *end*]])
: Возвращает количество вхождений подстроки *sub* в `s[`*start*`:`*end*`]`. Необязательные аргументы *start* и *end* интерпретируются так же, как и в операции среза (раздел 11.2).

endswith(*suffix* [, *start* [, *end*]])
Возвращает 1, если строка s[*start*:*end*] заканчивается на *suffix*, иначе возвращает 0.

find(*sub* [, *start* [, *end*]])
Возвращает наименьший индекс в исходной строке *s* начала вхождения подстроки *sub* в s[*start*:*end*]. Необязательные аргументы *start* и *end* интерпретируются так же, как и в операции среза (раздел 11.2). Если подстрока не найдена, возвращает -1.

index(*sub* [, *start* [, *end*]])
Аналог метода find(), генерирующий исключение, если подстрока не найдена.

rfind(*sub* [, *start* [, *end*]])
Возвращает наибольший индекс в исходной строке *s* начала вхождения подстроки *sub* в s[*start*:*end*]. Необязательные аргументы *start* и *end* интерпретируются так же, как и в операции среза (раздел 11.2). Если подстрока не найдена, возвращает -1.

rindex(*sub* [, *start* [, *end*]])
Аналог метода rfind(), генерирующий исключение, если подстрока не найдена.

startswith(*prefix* [, *start* [, *end*]])
Возвращает 1, если строка s[*start*:*end*] начинается с *prefix*, иначе возвращает 0.

Преобразование символов и фрагментов строк

expandtabs([*tabsize*])
Возвращает копию исходной строки, в которой все символы табуляции заменены одним или несколькими пробелами в зависимости от текущей позиции размера табуляции. По умолчанию размер табуляции *tabsize* равен 8 символам.

lstrip()
Возвращает копию строки, с удаленными идущими в начале строки символами пропуска (см. метод isspace()).

replace(*old*, *new* [, *maxcount*])
Возвращает копию строки, в которой все вхождения подстроки *old* заменены на *new*. Если задан необязательный аргумент *maxcount*, заменяются только первые *maxcount* вхождений.

rstrip()
Возвращает копию строки, с удаленными идущими в конце строки символами пропуска (см. метод isspace()).

strip()
Возвращает копию строки, с удаленными идущими в начале и конце строки символами пропуска (см. метод isspace()).

translate(*table* [, *delchars*])

Возвращает копию строки, в которой все символы, указанные в строке *delchars* удалены, а для оставшихся символов произведена заменена $c \Rightarrow$ *table*[ord(*c*)]. Аргумент *table* должен быть строкой из 256 символов. **Замечание:** у объектов unicode этот метод имеет другой синтаксис (см. раздел 11.2.2).

encode([*encoding* [, *errors*]])

Возвращает представление строки в кодировке *encoding* (по умолчанию ASCII). Аргумент *errors* (строка) указывает способ обработки ошибок. По умолчанию используется 'strict' — если символ не может быть представлен в данной кодировке, генерируется исключение UnicodeError (класс, производный от ValueError). Другие возможные значения — 'ignore' (отсутствующие в кодировке символы удаляются) и 'replace' (отсутствующие в кодировке символы заменяются, обычно на символ '?'). (Метод encode() в версии 1.6 определен только для строк Unicode.)

Разбиение и объединение

join(*seq*)

Возвращает объединение строк-элементов последовательности *seq*, используя строку *s* в качестве разделителя. Если последовательность содержит элементы, которые не являются строками, генерирует исключение ValueError.

split([*sep* [, *maxcount*]])

Возвращает список слов, содержащихся в исходной строке. В качестве разделителя слов используется строка *sep*, если она не задана или равна None, разделителем слов считаются символы пропуска. Если задан аргумент *maxcount* и *maxcount* >= 0, возвращается список из *maxcount* первых слов и остатка (таким образом список будет содержать не более *maxcount*-1 элементов).

splitlines([*keepends*])

Аналог метода split(), использующий в качестве разделителя переход на новую строку. Символы перехода на новую строку включаются в результат, только если задан и является истинным необязательный аргумент *keepends*.

Методы, зависящие от национальных установок

Поведение следующих методов зависит от текущих национальных установок. Вы можете изменить их с помощью функции locale.setlocale() (см. описание стандартного модуля locale). Применительно к строкам Unicode все описанные здесь методы работают со всеми символами Unicode, для которых соответствующее преобразование однозначно.

capitalize()

Возвращает копию строки, в которой первая буква заменена на прописную.

isdigit()
Возвращает 1, если строка содержит только цифры, иначе возвращает 0.

islower()
Возвращает 1, если строка содержит хотя бы один символ, который может быть записан в верхнем и нижнем регистре, и все такие символы в строке находятся в нижнем регистре (строчные), иначе возвращает 0.

isspace()
Возвращает 1, если строка содержит только символы пропуска (whitespace) — пробел, табуляция, перевод на новую строку и т. д., иначе возвращает 0.

istitle()
Возвращает 1, если регистр букв в строке соответствует заголовку (строчные буквы следуют только после символов, которые могут быть записаны в верхнем и нижнем регистре, а прописные только после символов, для которых нет понятия регистра, и с начала строки), иначе возвращает 0[6].

isupper()
Возвращает 1, если строка содержит хотя бы один символ, который может быть записан в верхнем и нижнем регистре, и все такие символы в строке находятся в верхнем регистре (прописные), иначе возвращает 0.

lower()
Возвращает копию строки, все символы которой приведены к нижнему регистру.

swapcase()
Возвращает копию строки, в которой регистр букв изменен с верхнего на нижний и наоборот.

title()
Возвращает копию строки, в которой регистр букв соответствует заголовку (строчные буквы следуют только после символов, которые могут быть записаны в верхнем и нижнем регистре, а прописные только после символов, для которых нет понятия регистра, и с начала строки)[6].

upper()
Возвращает копию строки, все символы которой приведены к верхнему регистру.

11.2.2 Строки Unicode

Литералы для строк Unicode образуются из строковых литералов (любого вида) при добавлении в начало символа 'u' или 'U': u'abc', ur"""abc""" (но не ru'abc'). Кроме того, литералы строк Unicode могут содержать управляющие последовательности вида '\u*hhhh*', которые интерпретируются так же, как и '\x*hh...*', но всегда должны содержать ровно четыре шестнадцатеричные цифры, что гораздо удобнее. Обратите

[6] Несколько упрощенное правило написания заголовков в некоторых западных языках — каждое слово заголовка начинается с прописной буквы. Обычно используется более сложное правило, по которому предлоги пишутся полностью строчными буквами.

внимание, что управляющее последовательности вида '`\x`*hh*`...`' и '`\u`*hhhh*' позволяют представить все символы Unicode, а '`\`*ooo*' — только первые 256.

Синтаксис метода `translate` объектов `unicode` отличается от одноименного метода объектов `string`:

translate(*table*)
: Возвращает копию строки, в которой все символы *c*, для которых в отображении *table* содержится запись с ключом `ord(`*c*`)`, заменены на `unichr(`*table*`[ord(`*c*`)])` или удалены, если *table*`[ord(`*c*`)] == None`. *table* может быть отображением любого типа.

Помимо описанных в предыдущем разделе, строки Unicode имеют еще несколько методов:

isdecimal()
: Возвращает `1`, если строка не содержит ничего, кроме десятичных цифр, иначе возвращает `0`. Под символами десятичных цифр помимо традиционных ASCII символов подразумеваются также символы, используемые в различных языках, надстрочные и подстрочные десятичные цифры (метод `isdigit` применительно к строкам Unicode охватывает также и символы с точкой, в скобках, в круге и т. д.).

isnumeric()
: Возвращает `1`, если строка не содержит ничего, кроме символов-чисел, иначе возвращает `0`. Помимо символов цифр, символы чисел включают символы римских чисел, дробные и т. п.

11.2.3 Кортежи

Кортежи конструируются простым перечислением элементов через запятую, заключенным в круглые скобки. Обычно (если это не конфликтует с другими элементами синтаксиса языка) скобки можно опустить, однако пустой кортеж всегда должен записываться в скобках. Кортеж с одним элементом должен содержать завершающую запятую: '`a, b, c`', '`()`', '`(d,)`'. Последовательности другого типа могут быть преобразованы в кортеж с помощью встроенной функции `tuple()` (см. главу 12). Кортежи не поддерживают никаких дополнительных операций кроме тех, которые характерны для всех последовательностей.

11.2.4 Объекты `xrange`

Объект `xrange` создается встроенной функцией `xrange()` (см. главу 12) и является псевдопоследовательностью, представляющей ряд целых чисел (объект не хранит свои элементы, а создает каждый элемент при обращении к нему). Достоинство объектов

xrange состоит в том, что они всегда занимают одинаковое количество памяти, независимо от размера представляемого диапазона. Использование объектов xrange вместо списков не дает существенного выигрыша в производительности.

Помимо операций, определенных для всех последовательностей, объекты xrange имеют один метод:

tolist()
: Преобразует объект, к которому метод применяется, в объект-список и возвращает его.

11.2.5 Объекты buffer

Объекты buffer создаются с помощь встроенной функции buffer() (см. главу 12) из различных объектов, поддерживающих соответствующий интерфейс: встроенных типов string, unicode, buffer, а также некоторых типов, определенных в различных модулях (например, array из модуля array). Объекты buffer позволяют обращаться с различными структурами как с массивами байтов. При этом все изменения в исходный объект будут немедленно отражаться в (ранее созданном) объекте buffer. Встроенная функция str() преобразует объект buffer в строку.

11.2.6 Изменяемые последовательности

Все описанные выше последовательности являются неизменяемыми — после создания такого объекта Вы не можете внести в него изменения. Объекты-списки (list) поддерживают дополнительные операции, позволяющие изменять объект[7]. Списки конструируются путем перечисления в квадратных скобках через запятую его элементов: '[a, b, c]'. Начиная с версии 2.0, списки могут быть также сконструированы с помощью расширенной записи '[*expression iter_list*]', где *iter_list* состоит из одного или нескольких (разделенных символами пропуска) выражений вида 'for *lvalue* in *expression*' и, возможно, 'if *test*'. При этом для каждой комбинации всех *lvalue* (см. раздел 10.3.2), для которой все выражения *test* являются истинными.

Ниже приведены дополнительные операции, позволяющие добавлять, удалять и заменять элементы списка (*x* — произвольный объект, *s* и *t* — последовательности одинакового[3] типа):

s[*i*] = *x*
: Заменяет элемент последовательности, на который указывает индекс *i* на *x*.

del *s*[*i*]
: Удаляет элемент последовательности, на который указывает индекс *i*.

[7]В дальнейшем, возможно, в язык будут добавлены другие типы изменяемых последовательностей. К ним также будут применимы операции, описанные в этом разделе.

s[[*i*]:[*j*]] = *t*
: Заменяет срез последовательности от *i* до *j* на *t* (удаляет из последовательности элементы входящие в срез и вставляет элементы из *t*).

del *s*[[*i*]:[*j*]]
: Удаляет из последовательности элементы, входящие в срез (эквивалентно '`s[i:j] = []`').

Кроме того, Вы можете вносить изменения в изменяемые последовательности с помощью методов (*x* — произвольный объект, *s* — последовательность, к которой применяется метод, и *t* — последовательность того же[3] типа). Для экономии памяти при работе с большими списками все методы вносят изменения в уже существующий список, а не создают новый. Во избежание возможных ошибок, они (если не указано иного) не возвращают результат.

append(*x*)
: Добавляет объект в конец последовательности (для списков это эквивалентно '`s[len(s):len(s)] = [x]`').

extend(*t*)
: Добавляет в конец последовательности элементы последовательности *t* (эквивалентно '`s[len(s):len(s)] = x`').

count(*x*)
: Возвращает число вхождений элемента *x* в последовательность (число элементов последовательности *s*, равных *x*).

index(*x*)
: Возвращает наименьшее *i*, для которого `s[i] == x`. Генерирует исключение `ValueError`, если последовательность не содержит элемента, равного *x*.

insert(*i*, *x*)
: Вставляет в последовательность элемент *x* перед *i*-м элементом (для списков это эквивалентно '`s[i:i] = [x]`', если `i >= 0`, иначе элемент вставляется в начало). Если значение индекса меньше нуля, элемент вставляется в начало последовательности, если больше длины последовательности — в ее конец.

pop([*i*])
: Возвращает *i*-й элемент последовательности, одновременно удаляя его из списка. Если индекс не указан, подразумевается последний элемент списка. Положение элемента можно отсчитывать с конца, указывая отрицательный индекс. Если индекс выходит за пределы диапазона, генерируется исключение `IndexError`.

remove(*x*)
: Удаляет из списка первый элемент со значением *x* (эквивалентно `del s[s.index(x)]`). Если такового в списке нет, то генерируется исключение `ValueError`.

reverse()
: Располагает элементы последовательности в обратном порядке.

sort([*cmpfunc*])

Располагает элементы последовательности в порядке возрастания. В качестве необязательного параметра можно передать функцию двух переменных, определяющую сравнение (по умолчанию сравнение производится так же, как и встроенной функцией `cmp()`). Однако использование пользовательской функции сравнения значительно замедляет процесс сортировки. Так, например, последовательное применение методов `sort()` и `reverse()` работает значительно быстрее, чем сортировка с использованием функции сравнения '`lambda x, y: cmp(y, x)`'.

11.3 Отображения

Отображения ставят в соответствие объектам (ключам) другие объекты. Отображения являются изменяемыми объектами. В настоящее время есть только один стандартный тип отображения — словарь (`dictionary`). В качестве ключа в словаре не могут использоваться изменяемые объекты, сравнение которых производится по значению вместо простой проверки идентичности (см. раздел 10.2.4), например списки и словари. Если два объекта равны (в том числе, если они имеют разный тип), например `1` и `1.0`, при использовании их в качестве ключа Вы получите доступ к одной и той же записи словаря.

Создаются словари путем перечисления в фигурных скобках через запятую пар *key*`:` *value*, например, '`{'jack': 4098, 'sjoerd': 4127}`' или '`{4098: 'jack', 4127: 'sjoerd'}`'. Интерпретатор не обнаруживает конфликты, связанные с наличием нескольких записей с одинаковыми ключами, — сохраняется только запись, идущая последней.

Для отображений определены следующие операции (*m* — отображение, *k* — ключ, *v* — произвольный объект):

len(*m*)

Возвращает количество записей в *m*.

m[*k*]

Значение, соответствующий ключу *k* в *m*. Если отображение не содержит запись с ключом *k*, генерируется исключение `KeyError`.

m[*k*] = *v*

Устанавливает значение, соответствующее ключу *k* в *v*.

del *m*[*k*]

Удаляет из *m* запись, соответствующую ключу *k*. Если отображение не содержит запись с ключом *k*, генерируется исключение `KeyError`.

Кроме того, отображения имеют следующие методы (*m* — отображение, к которому применяется метод, *n* — отображение того же[8] типа, *k* — ключ, *v* — произвольный объект):

[8]Отображения, определяемые пользователем, могут не иметь этого ограничения.

clear()
: Удаляет все записи из отображения.

copy()
: Создает и возвращает поверхностную (т. е. ключи и значения нового отображения являются теми же объектами, которые были в исходном, а не их копиями) копию исходного отображения.

has_key(*k*)
: Возвращает 1, если отображение содержит запись с ключом *k*, иначе возвращает 0.

items()
: Возвращает список записей отображения в виде '(*key*, *value*)'

keys()
: Возвращает список ключей отображения.

update(*m*)
: Эквивалентно выполнению 'for *k* in *b*.keys(): *m*[*k*] = *b*[*k*]'.

values()
: Возвращает список значений, содержащихся в отображении.

get(*k* [, *v*])
: Возвращает *m*[*k*], если отображение содержит запись с ключом *k*, иначе *v* (по умолчанию None).

setdefault(*k* [, *v*])
: Если отображение содержит запись с ключом *k*, возвращает *m*[*k*], в противном случае возвращает значение *v*, предварительно добавив его в отображение с ключом *k*. Если аргумент *v* опущен, он считается равным None. Поведение этого метода похоже на поведение метода get(), за исключением того, что если запись с заданным ключом отсутствует, значение по умолчанию не только возвращается, но и добавляется в отображение.

Порядок следования элементов в списке не определен. Однако, если вызвать методы keys() и values(), не внося между их вызовами изменений в отображение, то порядок следования элементов в возвращаемых списках будет одинаковый. Это позволяет создавать список пар (*value*, *key*), используя встроенную функцию map(): 'pairs = map(None, *m*.values(), *m*.keys())'.

11.4 Объекты, поддерживающие вызов

В этом разделе описаны типы объектов, для которых определена операция вызова (*obj*(*arglist*)). Вы можете узнать, поддерживает ли объект операцию вызова, с помощью встроенной функции callable().

11.4.1 Функции, определенные пользователем

Объекты-функции (`function`) создаются при определении функции с помощью инструкции `def` или оператора `lambda`. Единственная операция, определенная для функций — их вызов: '*func(arglist)*'.

Объекты-функции имеют следующие атрибуты:

func_doc

__doc__

Строка документации или `None`, если она не определена.

func_name

__name__

Имя функции.

func_defaults

Значения аргументов по умолчанию или `None`, если ни один аргумент не имеет значения по умолчанию.

func_code

Объект кода (`code`), представляющий собой байт-компилированное тело функции.

func_globals

Ссылается на словарь, представляющий глобальное пространство имен модуля, в котором определена функция (имена, являющиеся для функции глобальными).

Атрибутам `func_doc` (или `__doc__`), `func_defaults` и `func_code` можно присвоить новые значения, остальные — доступны только для чтения[9].

11.4.2 Методы, определенные пользователем

Объект-метод (`instance method`) создается при обращении к атрибуту класса или экземпляра класса, определенному как функция-атрибут класса. Если же Вы обратитесь к нему напрямую через специальный атрибут класса `__dict__`, Вы получите объект-функцию в неизменном виде. Метод может быть привязанным к экземпляру класса (если Вы обращаетесь к нему как к атрибуту экземпляра класса) и не привязанным (если Вы обращаетесь к нему как к атрибуту класса). Важно понимать, что преобразование атрибута-функции в атрибут-метод производится, только если функция является атрибутом класса. Функции, являющиеся атрибутами экземпляра класса в метод не преобразуются.

[9] `func_globals` является объектом изменяемого типа — Вы можете внести в него изменения, например, используя методы, но не можете присвоить новое значение. Следует понимать, что при этом будет изменено глобальное пространство имен модуля, в котором определена функция, и изменения отразятся на других объектах, определенных в этом модуле.

Объекты-методы имеют следующие атрибуты:

im_func
Объект-функция, реализующая метод. Атрибут доступен только для чтения.

im_self
Экземпляр класса, к которому применяется метод (None, если метод не привязан). Атрибут доступен только для чтения.

im_class
Класс, в котором метод определен. Атрибут доступен только для чтения.

__name__
Имя метода (то же самое, что и im_func.__name__).

__doc__
Строка документации (то же самое, что и im_func.__doc__).

При вызове метода, не привязанного к экземпляру класса, вызывается лежащая в его основе функция (im_func) с одной лишь разницей: если первый аргумент не является экземпляром класса im_class или производного от него класса, генерируется исключение TypeError. Если же метод привязан к определенному экземпляру класса, функция im_func вызывается со списком аргументов, в начало которого вставлен экземпляр im_self. То есть, если класс C содержит определение функции f и x является экземпляром класса C, то вызов x.f(1) эквивалентен вызову C.f(x, 1).

Заметим, что преобразование объекта-функции в (привязанный или не привязанный к экземпляру класса) метод происходит при каждом обращении к атрибуту класса или его экземпляра. В некоторых случаях Вы можете немного ускорить работу программы, присвоив метод локальной переменной и затем используя ее для многократного вызова метода.

11.4.3 Встроенные функции и методы

Встроенные функции и методы представляются в языке Python объектами одного типа — builtin_function_or_method. Количество и тип воспринимаемых ими аргументов определяется реализацией. Объекты, представляющие встроенные функции и методы, являются неизменяемыми и не содержат ссылок на изменяемые объекты.

Объекты builtin_function_or_method имеют следующие атрибуты:

__doc__
Строка документации (None, если строка документации не определена).

__name__
Имя функции/метода.

__self__
Ссылка на объект, к которому должен применяться метод или `None` для встроенных функций.

11.4.4 Классы

Когда вызывается объект-класс, создается и возвращается новый экземпляр этого класса. При этом подразумевается вызов специального метода класса `__init__` (если он определен) с аргументами, с которыми вызывается объект-класс. Если метод `__init__` не определен, объект-класс должен вызываться без аргументов. Свойства объектов-классов описаны в разделе 11.6.1.

11.4.5 Экземпляры классов

Если для класса определить метод `__call__()` (см. раздел 11.6.3), то его экземпляры будут поддерживать вызов: вызов экземпляра (`x(arglist)`) преобразуется в вызов метода `__call__()` с теми же аргументами (`x.__call__(arglist)`). Свойства экземпляров классов описаны в разделе 11.6.2.

11.5 Модули

Единственная операция, поддерживаемая объектами-модулями — доступ к их атрибутам: `m.name`, где *m* — модуль и *name* — имя, определенное в пространстве имен модуля *m* (Вы можете как получить, так и присвоить новое значение атрибуту модуля). Для создания объекта модуля предназначена инструкция `import` (см. раздел 10.3.10).

Объекты модули имеют следующие атрибуты:

__dict__
Словарь, являющийся таблицей имен модуля. Внося в него изменения, Вы меняете таблицу имен модуля, однако присваивание атрибуту `__dict__` невозможно (то есть Вы можете написать `m.__dict__['a'] = 1`, тем самым определив `m.a` равным `1`, но не можете написать `m.__dict__ = {}`).

__name__
Имя модуля.

__doc__
Строка документации модуля.

__file__
Полное имя файла, содержащего определение модуля (у модулей, статически встроенных в интерпретатор, атрибут `__file__` отсутствует).

11.6 Классы и экземпляры классов

11.6.1 Классы

Объект-класс создается с помощью определения класса (см. раздел 10.4.6). Пространство имен класса реализовано в виде словаря, доступного через специальный атрибут `__dict__`. Ссылка на атрибут класса преобразуется в поиск соответствующего имени в словаре, например, `C.x` преобразуется в `C.__dict__['x']`. Если атрибут не найден, поиск продолжается в базовых классах. Поиск производится сначала в глубину, затем слева направо, в порядке следования их в списке базовых классов. Если атрибуту соответствует объект `function`, то объект преобразуется в не привязанный к экземпляру объект-метод (см. раздел 11.4.2). Атрибут `im_class` этого объекта-метода указывает на класс, в котором найден соответствующий ему объект-функция (он может быть одним из базовых классов).

Присваивание атрибуту класса вносит изменения в словарь *этого* класса, но никогда не изменяет словари базовых классов.

Объект-класс может быть вызван, для получения экземпляра класса (см. раздел 11.4.4).

Объекты-классы имеют следующие атрибуты:

`__name__`
Имя класса.

`__module__`
Имя модуля, в котором класс определен.

`__dict__`
Словарь атрибутов класса. Вы можете изменять его содержимое, тем самым добавлять и удалять атрибуты класса и изменять их значения. Вы даже можете присвоить `__dict__` новое значение, но это может спровоцировать довольно странные ошибки.

`__bases__`
Кортеж базовых классов в порядке их следования в списке базовых классов. Вы можете изменить значение этого атрибута, однако вряд ли найдется достаточно веская причина для подобных манипуляций.

`__doc__`
Строка документации класса или `None`, если она не определена.

11.6.2 Экземпляры классов

Объект-экземпляр класса возвращается при вызове объекта-класса (см. раздел 11.4.4). Пространство имен объекта-экземпляра реализовано в виде словаря, в котором поиск

атрибутов производится в первую очередь. Если атрибут не найден, поиск продолжается среди атрибутов класса (см. раздел 11.6.1). При этом атрибут класса, являющийся методом, становится привязанным к данному экземпляру класса (см. раздел 11.4.2). Если нет атрибута класса с таким именем, вызывается специальный метод `__getattr__()`. Если же метод `__getattr__()` не определен, генерируется исключение `AttributeError`.

Присваивание атрибуту и его удаление вносят изменения в словарь экземпляра класса, но никогда не изменяют словарь класса. Если определены специальные методы `__setattr__()` и `__delattr__()`, то вызываются эти методы вместо того, чтобы вносить изменения в словарь экземпляра напрямую.

Объекты-экземпляры имеют следующие атрибуты:

`__dict__`
: Словарь атрибутов экземпляра класса. Вы можете изменять его содержимое, тем самым добавлять и удалять атрибуты экземпляра и изменять их значения. Вы даже можете присвоить `__dict__` новое значение, однако это может спровоцировать довольно странные ошибки (часто программы на языке Python рассчитывают на возможность добавить атрибут к экземпляру класса, присваивание же `__dict__` нового значения может привести к мистическим исчезновениям таких атрибутов).

`__class__`
: Объект-класс, экземпляром которого объект является. Вы можете изменить значение этого атрибута, однако вряд ли найдется достаточно веская причина для подобных манипуляций.

11.6.3 Специальные методы

Определив для класса методы со специальными именами, Вы можете реализовать над его экземплярами определенные операции, которые вызываются с помощью специального синтаксиса.

Инициализация и очистка

`__init__`(*`self`* [, *`args...`*])
: Вызывается при создании экземпляра класса (конструктор). *`args...`* — аргументы, с которыми вызывается объект-класс. Если базовый класс имеет метод `__init__()`, метод `__init__()` производного класса должен явно вызвать его, для инициализации части экземпляра, определенной в базовом классе.

`__del__`(*`self`*)
: Вызывается перед уничтожением экземпляра (деструктор). Если базовый класс имеет метод `__del__()`, метод `__del__()` производного класса должен явно вызвать его, для очистки части экземпляра, определенной в базовом классе. Заметим, что можно (но не рекомендуется) предотвратить уничтожение экземпляра, создав новую ссылку на него. Метод `__del__()` будет вызван снова — при удалении

этой ссылки. Если во время выхода из интерпретатора экземпляр все еще присутствует, вызов деструктора для него не гарантируется.

Замечания:

- Инструкция 'del x' не вызывает напрямую x.__del__(), а лишь удаляет имя 'x' из локального пространства имен, тем самым уменьшая число ссылок на объект на единицу. Когда число ссылок на объект достигает нуля, для него вызывается метод __del__(). Наиболее часто встречающаяся причина, по которой количество ссылок на объект не достигает нуля — существование циклических ссылок между объектами. Такая ситуация возникает, например, при использовании двоесвязных списков или графов. В этом случае единственное средство — явно разорвать циклы. Циклические ссылки также создаются, если Вы сохраняете объект traceback, полученный с помощью функции sys.exc_info() (см. описание модуля sys). В интерактивном режиме объект traceback, относящийся к последнему не перехваченному исключению, хранится в sys.last_traceback. Он содержит ссылку на кадр стека функции, в которой возникла исключительная ситуация, и, соответственно, на все переменные функции. Чтобы разорвать эту связь достаточно присвоить None переменной sys.last_traceback.
- В связи с тем, что условия, при которых вызывается метод __del__(), зависят от многих обстоятельств, не перехваченные исключения, возникающие во время его выполнения, игнорируются и на sys.stderr выводится соответствующее предупреждение. Кроме того, если объект удаляется при удалении объекта-модуля, в котором он определен (например, по окончании выполнения программы), другие глобальные объекты этого модуля уже могут быть удалены. По этой причине метод __del__() должен выполнять минимум операций поддержки внешних ресурсов. Гарантируется, что глобальные объекты, не являющиеся объектами-модулями, имя которых начинается с одного символа подчеркивания, удаляются раньше других.
- "Сборщик мусора" (см. описание модуля gc) автоматически обрабатывает только объекты, не имеющие деструктора. Наличие в цикле ссылающихся друг на друга, но недоступных извне объектов экземпляра с методом __del__() приведет к тому, что все объекты цикла останутся не уничтоженными.

Истинность и сравнение

__cmp__(*self*, *other*)

Вызывается (для первого или второго операнда) при выполнении всех операций сравнения, кроме случаев, когда оба операнда являются экземплярами классов и при этом для первого операнда не определен метод __cmp__(). Должен возвращать целое число: отрицательное, если self < other, ноль, если self == other, и положительное целое число, если self > other (см. также описание функции cmp() в главе 12). Если метод __cmp__() (или __rcmp__()) не определен, сравнение экземпляров производится по идентификаторам.

__rcmp__(*self*, *other*)

Вызывается для второго операнда, если оба операнда являются экземплярами

классов и при этом для первого операнда не определен метод `__cmp__()`. Должен возвращать целое число: отрицательное, если `self > other`, ноль, если `self == other`, и положительное целое число, если `self < other`. Нет смысла определять `cmp(y, x)` отличным от `-cmp(x, y)`, поэтому метод `__rcmp__()` обычно определяют в виде:

```
def __rcmp__(self, other):
    return -cmp(self, other)
```

`__hash__`(*self*)
Метод предназначен для получения хэш-значения объекта. Вызывается для объектов, используемых в качестве ключа при операциях над словарями, а также при использовании встроенной функции `hash()`. Должен возвращать 32-битное целое число. Объекты, которые при сравнении считаются равными, должны иметь равные хэш-значения. Рекомендуется использовать своего рода смесь (используя, например, исключающее ИЛИ) хэш-значений компонентов объекта, играющих роль при сравнении. Если для класса не определен метод `__cmp__()`, то по умолчанию в качестве хэш-значения для его экземпляров будет использоваться идентификатор объекта, определение же метода `__hash__()` лишь замедлит выполнение программы. Если же у класса есть метод `__cmp__()`, но нет метода `__hash__()`, его экземпляры не могут быть использованы в качестве ключа в операциях над словарями. Хэш-значение должно быть постоянно на протяжении всего времени жизни объекта, поэтому изменяемые объекты не должны иметь метода `__hash__()`.

`__nonzero__`(*self*)
Вызывается для определения истинности объекта, должен возвращать `0` или `1`. Если этот метод не определен, используется метод `__len__()` (см. раздел 11.6.3); если же для класса не определен ни один из этих методов, все его экземпляры будут истинными.

Доступ к атрибутам

Вы можете изменить действия, выполняемые при доступе к атрибутам: получении значения, присваивании и удалении. Для повышения производительности эти методы кэшируются в объекте-классе в момент выполнения определения класса. Поэтому их изменение после выполнения определения класса не даст желаемого результата.

`__getattr__`(*self*, *name*)
Вызывается, если атрибут не доступен обычным способом (то есть экземпляр, класс, экземпляром которого он является, и все его родительские классы не имеют атрибута с именем *name*). Метод должен возвращать значение (вычисленное) атрибута или генерировать исключение `AttributeError`.

Обратите внимание, что в отличие от `__setattr__()` и `__delattr__()`, метод `__getattr__()` не вызывается, если атрибут найден обычным способом. Такой вариант эффективней и позволяет получить доступ из этих методов к другим атрибутам.

__setattr__(*self*, *name*, *value*)
Вызывается при попытке присваивания атрибуту вместо обычного механизма, который сохраняет значение в словаре `__dict__` экземпляра. *name* — имя атрибута, *value* — присваиваемое значение.

Обратите внимание, что попытка присвоить значение атрибуту экземпляра (например, '`setattr(self, name, value)`') внутри метода `__setattr__()` приведет к его рекурсивному вызову. Вместо этого следует добавить соответствующую запись в словарь атрибутов экземпляра: '`self.__dict__[name] = value`'.

__delattr__(*self*, *name*)
Вызывается при попытке удаления атрибута. Этот метод следует определять только в том случае, когда для объектов имеет смысл инструкция '`del self.name`'.

Операция вызова

__call__(*self* [, *args*...])
Вызывается при вызове экземпляра класса, то есть '`x(arg1, arg2, ...)`' является короткой записью для '`x.__call__(arg1, arg2, ...)`'.

Эмуляция последовательностей и отображений

Для того, чтобы определенные Вами объекты вели себя так же, как и встроенные типы `string`, `tuple`, `list` и `dictionary`, для них необходимо определить методы, описанные в этом разделе. Разница между последовательностями и отображениями заключается в том, что для последовательностей допустимыми ключами (индексами) являются целые числа k, такие что `0 <= k < N`, где N — длина последовательности. Для отображений же ключом может являться объект любого типа[10]. Кроме того, при применении к последовательности операции среза, в качестве ключа выступает объект, представляющий один (объект типа `slice` или `ellipsis`) или несколько (кортеж из объектов типа `slice` и `ellipsis`) срезов (см. раздел 11.8.3).

Кроме описанных здесь методов, для отображений рекомендуется реализовать методы `keys()`, `values()`, `items()`, `has_key()`, `get()`, `clear()`, `copy()`, и `update()`, ведущие себя так же, как и для объектов `dictionary` (см. раздел 11.3); для изменяемых последовательностей следует реализовать методы, характерные для объектов `list`: `append()`, `count()`, `index()`, `insert()`, `pop()`, `remove()`, `reverse()` и `sort()` (см. раздел 11.2.6). И, наконец, специальные методы `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()` и `__imul__()`, реализующие характерные для последовательностей операции сложения (объединение последовательностей) и умножения (размножение последовательностей), описаны в разделе 11.6.3. Для совместимости с предыдущими версиями, могут быть определены специальные ме-

[10]Тип `dictionary` налагает дополнительные ограничения: для его ключа должна быть определена операция получения хэш-значения (см. описание встроенной функции `hash()` в главе 12).

тоды `__getslice__()`, `__setslice__()` и `__delslice__()` (см. ниже), которые будут обрабатывать операции с простой (но не расширенной) записью среза.

`__len__`(*self*)
Вызывается для реализации встроенной функции `len()`. Возвращаемое значение должно быть целым числом, большим или равным нулю. Кроме того, если для объекта не определен метод `__nonzero__()`, истинность объекта определяется по значению, возвращаемому методом `__len__()`.

`__getitem__`(*self*, *key*)
Вызывается для реализации получения элемента по индексу или ключу (*self*[*key*]). Интерпретация отрицательных индексов (если Вы реализуете последовательность) должна производиться самим методом `__getitem__()`. Если *key* имеет несоответствующий тип, следует генерировать исключение `TypeError`. Если Вы эмулируете последовательность, то, в случае выхода индекса за пределы допустимых значений (с учетом преобразований отрицательных индексов), метод должен генерировать исключение `IndexError` (таким образом при переборе элементов последовательности определяется ее конец). Если отображение не содержит записи с ключом *key*, принято использовать исключение `KeyError`.

`__setitem__`(*self*, *key*, *value*)
Вызывается для реализации присваивания элементу с заданным ключом или индексом ('*self*[*key*] = *value*'). Этот метод необходим только для отображений, допускающих изменение значения, соответствующего определенному ключу, или добавление записи с новым ключом, а также для последовательностей, допускающих изменение элементов. Если Вы эмулируете последовательность, то, в случае выхода индекса за пределы допустимых значений (с учетом преобразований отрицательных индексов), следует генерировать исключение `IndexError`.

`__delitem__`(*self*, *key*)
Вызывается для реализации удаления элемента с заданным индексом или ключом ('del *self*[*key*]'). Этот метод необходим только для последовательностей и отображений, допускающих удаление элементов. Соглашения об использовании исключений такие же, как и для метода `__getitem__`.

До версии 2.0, для того, чтобы последовательность поддерживала операцию с простой записью среза, необходимо было определить следующие три метода (для неизменяемых последовательностей нужен только метод `__getslice__()`). Для всех трех методов опущенные значения индексов *i* и *j* заменяются на ноль и `sys.maxint` соответственно. Если в записи среза используются отрицательные индексы, к ним добавляется длина последовательности. Если же метод `__len__()` не реализован, генерируется исключение `AttributeError`. Заметим, что нет никакой гарантии того, что после добавления длины последовательности индекс не будет отрицательным. Индексы, превышающие длину последовательности, остаются неизменными. Обратите внимание, что эти методы используются только для реализации операций с простой записью среза.

__getslice__(*self*, *i*, *j*)
: Вызывается для реализации получения подпоследовательности (*self*[*i*:*j*]).

__setslice__(*self*, *i*, *j*, *sequence*)
: Вызывается для реализации присваивания срезу, то есть замены подпоследовательности ('*self*[*i*:*j*] = *sequence*').

__delslice__(*self*, *i*, *j*)
: Вызывается для реализации удаления подпоследовательности ('del *self*[*i*:*j*]').

Для реализации операций, касающихся расширенной записи срезов, используются методы `__getitem__()`, `__setitem__()` и `__delitem__()`. В этом случае в качестве ключа выступает объект типа `slice`, объект `Ellipsis` или кортеж из них (см. раздел 11.8.3). Если вы реализовали поддержку расширенной записи среза, то, начиная с версии 2.0, нет необходимости реализовывать методы `__getslice__()`, `__setslice__()` и `__delslice__()`: при их отсутствии автоматически будет вызван метод method**__**getitem**__**(), `__setitem__()` или `__delitem__()`.

Следующий пример показывает, как Вы можете сохранить возможность работы Вашей программы (модуля) со старыми версиями интерпретатора (предполагается, что методы method**__**getitem**__**(), `__setitem__()` или `__delitem__()` поддерживают объекты, представляющие срез, в качестве аргумента):

```
class MyClass:
    ...
    def __getitem__(self, item):
        ...
    def __setitem__(self, item, value):
        ...
    def __delitem__(self, item):
        ...
    if sys.hexversion<0x020000F0:
        # Эти методы не будут определены, если
        # используется версия 2.0 (финальная) или новее
        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Обратите внимание на использование функции `max()`. В то время как методы `__*item__()` всегда получают индексы без изменений, перед вызовом методов `__*slice__()` к отрицательным индексам добавляется длина последовательности. После такого преобразования индекс может остаться отрицательным и быть неверно интерпретирован метами `__*item__()`. Использование `max(0, i)` гарантирует, что индексы всегда будут интерпретироваться корректно.

Оператор `in` реализуется путем перебора элементов последовательности и сравнения их с искомым значением. Однако для некоторых последовательностей его можно реализовать более эффективно. Для этих целей в версии 1.6 добавлен еще один специальный метод:

__contains__(*self*, *item*)
: Вызывается для реализации операторов `in` и `not in` (‘*item* `in` *self*’, ‘*item* `not in` *self*’). До версии 1.6, а также если метод `__contains__` не определен, для реализации этих операторов используется последовательный перебор элементов последовательности и сравнение их с искомым значением. Операторы `in` и `not in` всегда возвращают `0` или `1`, определяющие истинность возвращаемого методом `__contains__` объекта (см. раздел 10.2.5).

Приведение объектов к базовым типам

В этом разделе описаны специальные методы, реализующие приведение объектов к строковому и числовым типам. Для того, чтобы объект можно было привести к встроенным типам последовательностей, достаточно определить специальные методы `__len__()` и `__getattr__()` (см. раздел 11.6.3).

Строковое представление

__repr__(*self*)
: Вызывается встроенной функцией `repr()` и при использовании обратных кавычек (`` `expr` ``) (а также функцией `str()` и инструкцией `print`, если метод `__str__()` не определен) для получения формального строкового представления объекта. Обычно такое представление должно выглядеть как выражение языка Python, пригодное для создания объекта с тем же самым значением. По традиции, объекты, для которых нельзя получить такое строковое представление достаточно простым способом, дают строку вида `'<полезная информация>'` (представление такого вида будет получено для экземпляра класса, если он не имеет метода `__repr__()`).

__str__(*self*)
: Вызывается встроенной функцией `str()`, инструкцией `print` и оператором форматирования с символом формата ‘`s`’ для получения строкового представления, наиболее пригодного для вывода.

__oct__(*self*)
__hex__(*self*)
: Вызываются для реализации встроенных функций `oct()` и `hex()` соответственно. Должны возвращать строковое значение.

Приведение к числовым типам

__complex__(*self*)
__int__(*self*)
__long__(*self*)
__float__(*self*)

Вызывается при использовании встроенных функций complex(), int(), long(), и float(), а также оператора форматирования с символами формата, отличными от 's'. Должны возвращать значение соответствующего типа.

Арифметические и битовые операции

Определение описанных в этом разделе методов позволит производить над объектами соответствующие арифметические и битовые операции привычным способом, то есть, используя операторы и встроенные функции.

Унарные операции

__neg__(*self*)
__pos__(*self*)
__invert__(*self*)
__abs__(*self*)

Вызывается для реализации унарных операторов -, +, ~ и встроенной функции abs().

Бинарные операции

Для того, чтобы вычислить выражение *x op y*, предпринимаются следующие шаги (__*op*__() и __r*op*__() — методы, реализующие оператор *op*, например, __add__() и __radd__() реализуют оператор +):

1. Если *x* является строкой и *op* является оператором форматирования (%), выполняется форматирование и все остальные шаги пропускаются.
2. Если *x* является экземпляром класса:
   (a) Если для *x* определен метод __coerce__(), *x* и *y* заменяются значениями, содержащимися в кортеже *x*.__coerce__(*y*); если же метод __coerce__() возвращает None, перейти к пункту 3.
   (b) Если после преобразований ни *x*, ни *y* не является экземпляром, перейти к пункту 4.
   (c) Если для *x* определен метод __*op*__, вернуть *x*.__*op*__(*y*), в противном случае восстановить значения *x* и *y*, которые были перед выполнением пункта 2a.

3. Если *y* является экземпляром класса:

   (a) Если для *y* определен метод `__coerce__()`, *x* и *y* заменяются значениями, содержащимися в кортеже `x.__coerce__(y)`; если же метод `__coerce__()` возвращает `None`, перейти к пункту 4.

   (b) Если после преобразований ни *x*, ни *y* не является экземпляром, перейти к пункту 4.

   (c) Если для *y* определен метод `__rop__`, вернуть `y.__rop__(x)`, в противном случае восстановить значения *x* и *y*, которые были перед выполнением пункта 3a.

4. (a) Если *x* является последовательностью и *op* — + или *, выполняется объединение и размножение соответственно.

   (b) В противном случае оба операнда должны быть числами. По возможности они приводятся к общему типу, и для этого типа выполняется соответствующая операция.

`__coerce__`(*self*, *other*)

Вызывается для приведения операндов к общему[11] типу. Должен возвращать кортеж, содержащий *self* и *other*, приведенные к общему типу, или `None`, если осуществить преобразование невозможно. В тех случаях, когда общий тип будет типом второго аргумента (*other*), эффективнее вернуть `None`, так как интерпретатор также будет пытаться вызвать метод `__coerce__` второго аргумента (иногда, однако, реализация другого типа не может быть изменена, в этом случае полезно осуществить приведение к типу второго аргумента здесь).

`__add__`(*self*, *other*)
`__sub__`(*self*, *other*)
`__mul__`(*self*, *other*)
`__div__`(*self*, *other*)
`__mod__`(*self*, *other*)
`__divmod__`(*self*, *other*)
`__pow__`(*self*, *other* [, *modulo*])
`__lshift__`(*self*, *other*)
`__rshift__`(*self*, *other*)
`__and__`(*self*, *other*)
`__xor__`(*self*, *other*)
`__or__`(*self*, *other*)

Вызываются (для правого операнда) для реализации бинарных операторов `+`, `-`, `*`, `/`, `%`, встроенной функции `divmod()`, оператора `**` и встроенной функции `pow()`, операторов `<<`, `>>`, `&`, `^` и `|` соответственно. Например, для того чтобы вычислить выражение *x*+*y*, где *x* является экземпляром класса, имеющим метод `__add__`,

[11] Иногда полезно возвратить значения разного типа, например список и целое число.

вызывается `x.__add__(y)`. Обратите внимание, что, если должен поддерживаться вызов встроенной функции `pow()` с тремя аргументами, метод `__pow__()` должен быть определен с третьим необязательным аргументом.

__radd__(*self*, *other*)
__rsub__(*self*, *other*)
__rmul__(*self*, *other*)
__rdiv__(*self*, *other*)
__rmod__(*self*, *other*)
__rdivmod__(*self*, *other*)
__rpow__(*self*, *other*)
__rlshift__(*self*, *other*)
__rrshift__(*self*, *other*)
__rand__(*self*, *other*)
__rxor__(*self*, *other*)
__ror__(*self*, *other*)
Вызываются (для левого операнда) для реализации перечисленных выше операций, если правый операнд не имеет соответствующего метода (без буквы '`r`'). Например, для того чтобы вычислить выражение *x*-*y*, где *y* является экземпляром класса, имеющим метод `__rsub__`, вызывается `y.__rsub__(x)`. Обратите внимание, что функция `pow()`, вызванная с тремя аргументами, не будет пытаться использовать метод `__rpow__()` (правила приведения типов были бы слишком сложными).

__iadd__(*self*, *other*)
__isub__(*self*, *other*)
__imul__(*self*, *other*)
__idiv__(*self*, *other*)
__imod__(*self*, *other*)
__ipow__(*self*, *other*)
__ilshift__(*self*, *other*)
__irshift__(*self*, *other*)
__iand__(*self*, *other*)
__ixor__(*self*, *other*)
__ior__(*self*, *other*)
Вызываются для реализации операций с присваиванием (изменением исходного объекта) `+=`, `-=`, `*=`, `/=`, `%=`, `**=` `<<=`, `>>=`, `&=`, `^=` и `|=` соответственно. Эти методы должны не возвращать результат, а модифицировать объект, к которому метод применяется. Если соответствующий метод не определен, операция '*x op*= *y*' выполняется как '*x* = *x op y*'.

11.7 Файловые объекты

Объекты типа `file` создаются встроенной функцией `open()` (см. главу 12), а также с помощью функций `os.popen()` и `os.fdopen()` (см. описание модуля `os`). Кроме того, различные модули предоставляют пути создания объектов с аналогичным интерфейсом (например, метод `makefile()` объектов, представляющих сетевое соединение). Переменные `sys.stdin`, `sys.stdout` и `sys.stderr` (см. описание модуля `sys`) инициализируются файловыми объектами, соответствующими стандартным потокам ввода, вывода и ошибок.

Если при выполнении операции над объектом-файлом возникает ошибка ввода-вывода, включая ситуации, при которых по каким-либо причинам операция не определена (например, метод `seek()` файлового объекта, представляющего устройство tty, или запись в файл, открытый для чтения), генерируется исключение `IOError`.

Объекты `file` имеют следующие методы:

close()
Закрывает файл. При попытке произвести какую-либо операцию над закрытым файлом генерируется исключение `ValueError`. Допускается вызов метода `close()` более одного раза.

flush()
Сбрасывает содержимое внутреннего буфера.

isatty()
Возвращает истину, если файл соответствует устройству tty, иначе возвращает `0` (ложь). Если объект с интерфейсом, характерным для файловых объектов, не ассоциирован с реальным файлом, для него этот метод не должен быть реализован.

fileno()
Возвращает дескриптор файла (целое число), используемый для запросов к операционной системе для выполнения операций ввода/вывода. Может быть полезен для выполнения операций низкого уровня, использующих дескрипторы (например, определенные в модулях `os` и `fcntl`). Если объект с интерфейсом, характерным для файловых объектов, не ассоциирован с реальным файлом, для него этот метод не должен быть реализован.

read([*size*])
Считывает не более *size* байт из файла (меньше, если конец файла достигается раньше, чем прочитано *size* байт). Если аргумент *size* опущен или отрицательный, считывается все содержимое файла. Считанные байты возвращаются в виде строкового объекта. Для некоторых файлов (например, устройства tty) вполне осмысленно продолжение чтения после достижения конца файла.

readline([*size*])
Считывает и возвращает строку из файла, включая завершающий символ новой

строки[12] (в последней строке файла он может отсутствовать). Если задан неотрицательный аргумент *size*, считывается не более *size* байт (учитывая завершающий символ новой строки). В этом случае возвращаемая строка может быть неполной.

readlines([*sizehint*])

Считывает все содержимое файла и возвращает список строк. Если задан неотрицательный аргумент *sizehint*, считывает (по сумме длин строк) примерно *sizehint* байт (*sizehint* округляется в большую сторону таким образом, чтобы значение было кратным размеру внутреннего буфера). Объекты с интерфейсом, характерным для файловых объектов, могут игнорировать значение *sizehint*.

seek(*offset* [, *whence*])

Устанавливает текущую позицию в файле. Необязательный аргумент *whence* указывает на точку отсчета: `0` (по умолчанию) — начало файла, `1` — текущая позиция и `2` — конец файла.

tell()

Возвращает текущую позицию в файле.

truncate([*size*])

Усекает файл до заданного (*size*) размера. По умолчанию размер равен текущей позиции. Наличие этого метода зависит от операционной системы.

write(*s*)

Записывает строку *s* в файл. Заметим, что из-за использования буферизации строка на самом деле может не быть записана в файл до тех пор, пока не будет вызван метод `flush()` или `close()`.

writelines(*slist*)

Записывает строки из списка *slist* в файл. Метод `writelines()` не добавляет разделители строк. Начиная с версии 1.6, *slist* может быть последовательностью любого типа, не обязательно `list`.

Файловые объекты также имеют следующие атрибуты:

closed

Текущее состояние файлового объекта: истина, если файл закрыт, иначе ложь. Доступен только для чтения, значение изменяется при использовании метода `close()`. Этого атрибута может не быть у объектов, аналогичных файловым.

mode

Режим ввода/вывода. Если файловый объект был создан с помощью встроенной функции `open()`, равен переданному ей аргументу *mode*. Доступен только для чтения. Этого атрибута может не быть у объектов, аналогичных файловым.

[12]Основным достоинством того, что завершающий символ строки остается в возвращаемой строке, является возможность использования пустой строки для обозначения достижения конца файла.

name

Если файловый объект был создан с помощью встроенной функции `open()`, равен имени файла. В противном случае, равен строке, которая указывает на происхождение файлового объекта, в виде '<*происхождение*>'. Доступен только для чтения.

softspace

Показывает, должен ли выводиться пробел перед выводом следующего значения с помощью инструкции `print` (истина) или нет (ложь). При создании собственных типов файловых объектов следует позаботиться о наличии доступного для записи атрибута `softspace`, который должен быть инициализирован нулем. Для классов, реализованных на языке Python, это происходит автоматически.

11.8 Вспомогательные объекты

11.8.1 Пустой объект

Тип `None` принимает единственное значение, и существует единственный объект этого типа, который доступен через встроенное имя `None`. Объект `None` используется для указания на отсутствие значения во многих ситуациях (например, если функция не возвращает явно никакого значения). Значение `None` всегда ложно.

11.8.2 Объекты типа

Объекты типа (`type`) представляют типы различных объектов. Вы можете получить объект типа для любого объекта (в том числе и для самого объекта типа) с помощью встроенной функции `type()`. В стандартном модуле `types` определены объекты типа для всех встроенных типов.

Объекты типа имеют следующие атрибуты, доступные только для чтения:

__doc__

Строка документации, описывающая назначение объектов данного типа, или `None`.

__name__

Имя типа. Именно эти имена используются в настоящей книге: Вы можете найти описание любого встроенного типа, воспользовавшись предметным указателем.

Строковое представление объектов типа выводится (например, инструкцией `print`) в виде `<type '`*имя типа*`'>`.

11.8.3 Представление расширенной записи среза

Эллипсис

Тип `ellipsis` принимает единственное значение, и существует единственный объект этого типа, который доступен через встроенное имя `Ellipsis`. Объект `Ellipsis` используется для указания на использование эллипсиса (‘`...`’) в расширенной записи среза. Значение объекта `Ellipsis` всегда истинно.

Срез

Объекты типа `slice` используются для представления расширенной записи срезов, то есть, если срез записывается с использованием двух символов двоеточия (*obj*[*i*:*j*:*s*]), указывается несколько срезов или используется эллипсис (*obj*[*i*:*j*, *k*:*l*], *obj*[..., *i*:*j*]). Кроме того, Вы можете создать объект `slice`, используя встроенную функцию `slice()`.

Объекты среза имеют следующие атрибуты, доступные только для чтения (каждый из них может быть любого типа и равен `None`, если соответствующий аргумент в записи среза ‘*start*:*stop*:*step*’ опущен):

start
Нижняя граница.

stop
Верхняя граница.

step
Шаг.

Фактически объект среза представляет множество индексов ‘`range(`*start*`,` *stop*`,` *step*`)`’.

11.9 Детали реализации

11.9.1 Объекты кода

Объекты кода (`code`) используются для представления байт-компилированного исполняемого кода, такого как тело функции. Разница между объектом-кодом и объектом-функцией состоит в том, что объект-функция содержит явную ссылку на словарь, представляющий глобальное пространство имен модуля, в котором определена функция, в то время как объект-код “не знает” контекста, в котором он будет выполняться. Кроме того,

значения аргументов по умолчанию сохраняются в объекте-функции, так как они вычисляются во время выполнения. Объект-код, в отличие от объекта-функции, неизменяем и не содержит ссылок (прямо или косвенно) на изменяемые объекты.

Объект-код создается с помощью встроенной функции `compile()`, а также может быть извлечен из объекта-функции используя его атрибут `func_code`. Объект-код можно выполнить с помощью инструкции `exec` или вычислить представляемое им выражение с помощью встроенной функции `eval()`.

Объекты кода имеют следующие атрибуты, доступные только для чтения:

co_name
: Содержит имя функции (или `'?'`, если объект-код создан с помощью функции `compile()`).

co_argcount
: Число позиционных аргументов функции (включая аргументы, имеющие значения по умолчанию).

co_nlocals
: Число локальных переменных (включая аргументы функции).

co_varnames
: Кортеж, содержащий имена локальных переменных.

co_code
: Строка, представляющая последовательность байт-компилированных инструкций.

co_consts
: Кортеж, содержащий значения используемых литеральных выражений.

co_names
: Кортеж всех используемых в байт-коде имен.

co_filename
: Имя файла, из которого объект-код был скомпилирован.

co_firstlineno
: Номер первой строки определения функции в файле.

co_lnotab
: Строка, представляющая отображение смещения в байт-коде в номера строк (пустая строка, если включена оптимизация).

co_stacksize
: Необходимый для выполнения размер стека (включая локальные переменные).

co_flags
: Число, представляющее различные флаги интерпретатора. Для `co_flags` определены следующие биты (наиболее важные): бит `0x04` установлен, если функция

использует запись '*args' (воспринимает произвольное число позиционных аргументов); бит 0x08 установлен, если функция использует запись '**kwds' (воспринимает произвольное число именованных аргументов).

Если объект кода представляет функцию, то первый элемент кортежа co_consts всегда является строкой документации (или None, если строка документации не определена).

11.9.2 Кадр стека

Объект frame представляет кадр стека функции, "окружение", в котором выполняются инструкции кода. Вы можете получить объект кадра стека через атрибут tb_frame объекта traceback (см. раздел 11.9.3). Объекты frame имеют следующие атрибуты (атрибутам f_trace, f_exc_type, f_exc_value и f_exc_traceback можно присвоить новое значение, остальные доступны только для чтения):

f_back
: Ссылка на предыдущий кадр стека (то есть, если мы рассматриваем кадр стека функции *func*, то этот атрибут ссылается на кадр стека функции, вызвавшей функцию *func*) или None, если рассматриваемый кадр стека является последним (то есть относится не к функции, а к коду, исполняемому в глобальной области видимости).

f_code
: Объект кода (см. раздел 11.9.1), который выполняется в рассматриваемом кадре стека.

f_locals
: Словарь, используемый для поиска локальных имен.

f_globals
: Словарь, используемый для поиска глобальных имен.

f_builtins
: Словарь, используемый для поиска встроенных имен.

f_restricted
: Флаг, указывающий на то, что функция выполняется в защищенном режиме.

f_lineno
: Номер последней выполненной в рассматриваемом кадре стека строки.

f_lasti
: Номер последней выполненной в рассматриваемом кадре стека инструкции байт-кода (может использоваться в качестве индекса для строки байт-кода объекта code).

f_trace
Функция, которая будет вызываться в начале каждой строки исходного кода, или None (используется отладчиком).

f_exc_type
f_exc_value
f_exc_traceback
Эти атрибуты представляют последнее перехваченное в рассматриваемом кадре исключение.

11.9.3 Объекты traceback

Объект traceback представляет путь, пройденный с момента генерации исключения до его перехвата. Таким образом, он создается при возникновении исключительной ситуации. Вы можете получить объект traceback в качестве третьего элемента кортежа, возвращаемого функцией sys.exc_info() (см. описание модуля sys). Если программа не содержит подходящего обработчика исключения, информация, хранящаяся в объекте traceback, выводится в стандартный поток ошибок. В интерактивном режиме объект traceback для последнего не перехваченного исключения доступен через переменную sys.last_traceback.

Объекты traceback имеют следующие атрибуты, доступные только для чтения:

tb_next
Ссылка на объект traceback, представляющий следующий уровень на пути в сторону кадра стека, в котором возникла исключительная ситуация, или None, если рассматриваемый объект представляет последний уровень.

tb_frame
Ссылка на кадр стека, представляющего текущий уровень.

tb_lineno
Номер строки (на текущем уровне), в которой возникла исключительная ситуация.

tb_lasti
Номер (на текущем уровне) инструкции байт-кода, которая привела к возникновению исключительной ситуации (может использоваться в качестве индекса для строки байт-кода объекта code).

Значения атрибутов tb_lineno и tb_lasti объекта traceback могут отличаться от значений атрибутов f_lineno и f_lasti объекта frame, если исключительная ситуация возникла в теле инструкции try без подходящей ветви except или имеющей ветвь finally.

Глава 12

Встроенные функции

Существует некоторый набор функций, встроенных в интерпретатор и, таким образом, всегда доступных для использования. Ниже приведены все встроенные функции в алфавитном порядке.

`__import__`(*name* [, globals [, locals [fromlist]]])

Эта функция вызывается инструкцией `import`. Вы можете заменить ее другой функцией, имеющей совместимый с `__import__()` интерфейс для того, чтобы изменить семантику инструкции `import`. Чтобы получить представление как и зачем это следует делать, смотрите описание стандартных модулей `ihooks` и `rexec`. Обратите также внимание на встроенный модуль `imp`, в котором определены некоторые операции, полезные для написания собственной функции `__import__`.

Например, инструкция '`import spam`' приводит к выполнению '`__import__('spam', globals(), locals(), [])`', а инструкция '`from spam.ham import eggs`' — к выполнению '`__import__('spam.ham', globals(), locals(), ['eggs'])`'. Несмотря на то, что `locals()` и `['eggs']` передаются в качестве аргументов, сама функция `__import__()` не устанавливает локальную переменную `eggs`. На самом деле, стандартная функция `__import__()` вообще не использует аргумент *locals*, а аргумент *globals* использует только для определения контекста инструкции `import`.

Если аргумент *name* задан в виде '*пакет.модуль*' и список *fromlist* пустой, функция `__import__()` возвращает объект-модуль, соответствующий имени '*пакет*' (часть имени *name* до первой точки), а не объект-модуль с именем *name*. Однако, если задан не пустой аргумент *fromlist*, функция `__import__()` возвращает объект-модуль, соответствующий имени *name*. Это сделано для совместимости байт-кода, сгенерированного для разных вариантов инструкции `import`: при использовании '`import spam.ham.eggs`', объект модуль `spam` должен быть помещен в пространство имен, соответствующее области видимости, в котором выполняется инструкция `import`, в то время как при использовании '`from spam.ham import eggs`', объект-модуль `spam.ham` должен быть использован для поиска объекта `eggs`. Для получения объекта Вы можете воспользоваться встроенной функцией `getattr()`, например:

```
import string

def my_import(name):
```

```
        mod = __import__(name)
        components = string.split(name, '.')
        for comp in components[1:]:
            mod = getattr(mod, comp)
        return mod
```

abs(*x*)

Возвращает абсолютное значение аргумента. Для комплексных чисел возвращает `x*x.conjugate()`. Поведение функции `abs()` применительно к экземплярам определяется специальным методом `__abs__()` (см. раздел 11.6.3).

apply(*object*, *args* [, *keywords*])

Аргумент *object* должен быть объектом, поддерживающим вызов (см. раздел 11.4) и аргумент *args* должен быть последовательностью (если *args* не является кортежем, последовательность сначала преобразуется в кортеж). Необязательный аргумент *keywords* должен быть словарем, ключи которого являются строками. Возвращает результат вызова функции *object* с позиционными аргументами *args* и именованными аргументами *keywords*. Начиная с версии 1.6, тот же результат Вы можете получить, используя расширенную запись вызова: '`object(*args, **keywords)`'.

buffer(*object* [, *offset*[, *size*]])

Создает и возвращает объект `buffer`, который ссылается на *object* (см. раздел 11.2.5). *object* должен быть объектом, поддерживающим соответствующий интерфейс (`string`, `unicode`, `array`, `buffer`). Объект `buffer` представляет срез от *offset* (или от начала, если аргумент *offset* опущен) длиной *size* (до конца объекта *object*, если аргумент *size* опущен).

callable(*object*)

Возвращает истину, если аргумент *object* поддерживает вызов, иначе возвращает ложь (см. также раздел 11.4).

chr(*i*)

Возвращает строку, состоящую из одного символа, код которого равен *i*. Например, `chr(97)` возвращает строку `'a'`. Аргумент *i* должен быть целым числом от 0 до 255 включительно, если *i* выходит за пределы указанного диапазона, генерируется исключение `ValueError`. Обратная операция выполняется функцией `ord()`. См. также описание функции `unichr()`.

cmp(*x*, *y*)

Сравнивает объекты *x* и *y* и возвращает целое число: отрицательное, если `x < y`, ноль, если `x == y`, и положительное, если `x > y`.

coerce(*x*, *y*)

Пытается привести объекты *x* и *y* к общему типу используя такие же правила, как и для арифметических операций, и возвращает результат в виде кортежа из двух объектов. Если оба аргумента являются экземплярами класса, специальный метод `__coerce__()` вызывается только для первого.

compile(*string*, *filename*, *kind*)

Компилирует строку *string* и возвращает объект кода (см. раздел 11.9.1). Аргумент *filename* указывает имя файла, из которого создается объект кода. Используйте строку вида '<*источник*>', если исходный текст берется не из файла (например, '<string>'). Аргумент *kind* указывает какого рода объект кода должен быть создан: используйте 'exec', если *string* состоит из последовательности инструкций, 'eval', если *string* содержит единственное выражение, или 'single', если *string* содержит одну инструкцию для интерактивного режима (в последнем случае результат выполнения инструкции-выражения отличный от None будет выведен в стандартный поток вывода).

complex(*real* [, *imag*])

Преобразует аргумент числового типа *real* к комплексному типу, добавляет к нему *imag**j (аргумент *imag* должен быть числового типа, по умолчанию равен нулю) или преобразует строку *real* в комплексное число (аргумент *imag* при этом игнорируется) и возвращает результат.

delattr(*object*, *name*)

Удаляет атрибут с именем *name* (строка) у объекта *object*, если объект позволяет это сделать. Например, вызов 'delattr(*x*, 'foobar')' эквивалентен инструкции 'del *x*.foobar'.

dir([*object*])

Без аргументов, возвращает список имен, определенных в локальной области видимости (с точностью до порядка следования, эквивалентно locals().keys()). С заданным аргументом, возвращает список атрибутов объекта *object*. Эта информация собирается из атрибутов объекта __dict__, __methods__ и __members__, если они определены. Заметим, что функция dir() выдает список лишь тех атрибутов, которые определены непосредственно для объекта *object*. Так, например, применительно к классу, не включаются атрибуты, определенные для базовых классов, а применительно к экземплярам, не включаются атрибуты класса. Имена в возвращаемом списке располагаются в алфавитном порядке.

divmod(*a*, *b*)

Возвращает кортеж из двух чисел[1]: целой части и остатка при делении *a* на *b*. К операндам применяются обычные правила приведения к общему типу, как и при выполнении других арифметических операций. Если оба операнда целые (int или long int), результат будет равен '(*a* / *b*, *a* % *b*)'. Для чисел с плавающей точкой результат будет равен '(*q*, *a* % *b*)', где *q* (типа float) обычно равно 'math.floor(*a* / *b*)', но может быть на единицу меньше. В любом случае, значение выражения '*q* * *b* + *a* % *b*' очень близко к *a*, а '*a* % *b*' (если не равно нулю) имеет тот же знак, что и *b*. Кроме того, всегда выполняется условие '0 <= abs(*a* % *b*) < abs(*b*)'.

eval(*expression* [, *globals* [, *locals*]])

Строка *expression* разбирается и вычисляется ее значение как выражения на

[1]Могут быть произвольного типа, если *a* или *b* является экземпляром класса с методом __divmod__ и __rdivmod__ соответственно.

языке Python, используя словари *globals* и *locals* в качестве глобального и локального пространств имен. Если опущен аргумент *locals*, он считается равным *globals*. Если опущены оба необязательных аргумента, выражение вычисляется в том окружении, из которого была вызвана функция `eval()`. Возвращает результат вычисления выражения. Например:

```
>>> x = 1
>>> eval('x+1')
2
```

Функция может быть также использована для вычисления байт-компилированного с помощью функции `compile()` (аргумент *kind* должен быть равен `'eval'`) выражения. В этом случае аргумент *expression* должен быть объектом кода.

См. также описание инструкции `exec` (раздел 10.3) и функций `execfile()`, `globals()` и `locals()`.

execfile(*file* [, *globals* [, *locals*]])

Эта функция является аналогом инструкции `exec`, но обрабатывает файл вместо строки. От инструкции `import` она отличается тем, что не использует управление модулями — считывает файл независимо от каких-либо условий, не создает (и не использует) байт-компилированную версию файла и не создает объект-модуль.

Файл с именем *file* считывается, разбирается и выполняется как последовательность инструкций языка Python (аналогично модулям), используя словари *globals* и *locals* в качестве глобального и локального пространств имен. Если опущен аргумент *locals*, он считается равным *globals*. Если опущены оба необязательных аргумента, выражение вычисляется в том окружении, из которого была вызвана функция `execfile()`.

filter(*function*, *list*)

Возвращает список из тех элементов последовательности *list*, для которых функция *function* (любой объект, поддерживающий вызов с одним аргументом, или `None`) возвращает истину. Если *list* является строкой (`string`) или кортежем, результат будет иметь тот же тип. В противном случае возвращаемое значение будет иметь тип `list`. Если аргумент *function* равен `None`, возвращаемый список будет содержать элементы последовательности *list*, являющиеся истиной.

float(*x*)

Преобразует строку или число (кроме комплексного) к вещественному типу и возвращает результат. Если аргумент является строкой, он должен содержать только десятичную запись целого или вещественного числа, заключенную, возможно, в символы пропуска (whitespace). Заметим, что ограничения, накладываемые на аргумент-строку, зависят от реализации библиотеки языка C. Экземпляры классов также могут быть преобразованы к вещественному типу, если для них определен специальный метод `__float__`.

getattr(*object*, *name* [, *default*])

Возвращает значение атрибута с именем *name* объекта *object*. Если *object*

не имеет атрибута с именем *name* (строка), возвращается значение необязательного аргумента *default* (если он задан) или генерируется исключение `AttributeError`.

globals()

Возвращает словарь, представляющий глобальное пространство имен текущего модуля (внутри функции — это глобальное пространство имен модуля, в котором определена функция, а не тот, в котором она используется).

hasattr(*object*, *name*)

Возвращает `1`, если объект *object* имеет атрибут с именем *name* (строка), иначе возвращает `0`. Функция реализована следующим образом: производится попытка получить значение атрибута с указанным именем и, если генерируется (любое) исключение, то считается, что объект не имеет атрибута с таким именем.

hash(*object*)

Возвращает хэш-значение объекта *object*. Если для объекта не может быть получено хэш-значение, генерирует исключение `TypeError`. Хэш-значения используются для быстрого сравнения ключей при поиске в словарях. Равные объекты (в том числе и разного типа, например, `1` и `1.0`) должны иметь равные хэш-значения.

hex(*x*)

Возвращает шестнадцатеричное представление целочисленного (`int` или `long int`) аргумента. Результат является правильным литеральным выражением языка Python (для длинного целого в конце будет добавлен суффикс '`L`'). Заметим, что результат всегда беззнаковый, например, на 32-разрядных платформах `hex(-1)` дает `'0xffffffff'`. При использовании результата в качестве литерального выражения на платформе с таким же размером слова, Вы получите `-1`. На платформах с другим размером слова Вы можете получить большое положительное число или будет сгенерировано исключение `OverflowError`.

id(*object*)

Возвращает идентификатор объекта *object* — целое число (типа `int` или `long int`). Гарантируется уникальность идентификатора на всем протяжении жизни объекта. Два объекта, промежутки времени существования которых не пересекаются, могут иметь одинаковые идентификаторы. В настоящее время идентификатор является адресом объекта.

input([*prompt*])

Эквивалентно `eval(raw_input(`*prompt*`))` (запрашивает выражение, вычисляет и возвращает его значение).

int(*x* [, *radix*])

Преобразует строку или число (любого типа, кроме комплексного) *x* в простое целое (`int`). Если аргумент *x* является строкой, он должен содержать только десятичную запись целого числа, заключенную, возможно, в символы пропуска (whitespace). В этом случае, начиная с версии 1.6, Вы можете задать другое основание системы счисления *radix* (в более ранних версиях, Вы можете воспользоваться функцией `string.atoi`) от 2 до 36 включительно[2] или `0`. Если аргумент

[2] 10 цифр плюс 26 букв латинского алфавита.

radix равен нулю, основание системы исчисления (8, 10 или 16) определяется автоматически исходя из записи числа аналогично тому, как обрабатываются числовые литеральные выражения. Если задан аргумент *radix*, а *x* не является строкой, генерирует исключение `TypeError`. Преобразование вещественных чисел зависит от реализации библиотеки языка C. Обычно число округляется в сторону нуля[3].

intern(*string*)
Добавляет строку *string* в специальную таблицу строк (если такой строки там не было) и возвращает тот же объект-строку или возвращает строку с таким же значением, ранее помещенную в эту таблицу. Это позволяет немного повысить производительность при поиске по ключу в словарях благодаря тому, что сравнение таких строк можно производить по идентификаторам (вместо обычного лексикографического сравнения). Эта процедура выполняется автоматически для всех имен, используемых в программах на языке Python, для ускорения поиска в словарях, представляющих пространства имен модулей, классов и экземпляров классов. Строки, помещенные в специальную таблицу, никогда не уничтожаются.

isinstance(*object*, *class*)
Возвращает `1`, если *object* является (прямо или косвенно) экземпляром класса *class* или является объектом типа *class*, иначе возвращает `0`. В первом случае результат будет таким же, как и у выражения '`issubclass(`*object*`.__class__, `*class*`)`', во втором — таким же, как и у выражения '`type(`*object*`) is `*class*'. Если *class* не является объектом-классом или объектом-типом, генерирует исключение `TypeError`.

issubclass(*class1*, *class2*)
Возвращает `1`, если класс *class1* является (прямо или косвенно) производным от класса *class2*. Считается, что класс также является производным от самого себя. Если какой-либо аргумент не является объектом-классом, генерируется исключение `TypeError`.

len(*object*)
Возвращает длину (количество элементов) объекта. Аргумент может быть последовательностью, отображением или экземпляром класса, для которого определен специальный метод `__len__()`.

list(*sequence*)
Возвращает список, составленный из элементов последовательности *sequence*. Порядок следования элементов сохраняется. Если последовательность *sequence* уже является списком, возвращается его копия (аналогично *sequence*`[:]`). Например, `list('abc')` возвращает `['a', 'b', 'c']`, а `list((1, 2, 3))` возвращает `[1, 2, 3]`.

locals()
Возвращает словарь, представляющий локальное пространство имен. Не следует изменять его, внесенные изменения могут не отразиться на значениях локальных переменных, используемых интерпретатором.

[3] Такое поведение неприятно — определение языка должно требовать округления в сторону нуля.

long(*x* [, *radix*])

Преобразует строку или число (любого типа, кроме комплексного) *x* в длинное целое (long int). Если аргумент *x* является строкой, он должен содержать только десятичную запись целого числа (сразу после последней цифры может присутствовать символ 'l' или 'L'), заключенную, возможно, в символы пропуска (whitespace). В этом случае, начиная с версии 1.6, Вы можете задать другое основание системы счисления *radix* (в более ранних версиях, Вы можете воспользоваться функцией string.atol) от 2 до 36 включительно или 0. Если аргумент *radix* равен нулю, основание системы исчисления (8, 10 или 16) определяется автоматически исходя из записи числа аналогично тому, как обрабатываются числовые литеральные выражения. Если задан аргумент *radix*, а *x* не является строкой, генерирует исключение TypeError. Преобразование вещественных чисел зависит от реализации библиотеки языка C. Обычно число округляется в сторону нуля. См. также описание функции int().

map(*function*, *seq* ...)

Последовательно применяет *function* (любой объект, поддерживающий вызов, или None) к каждому элементу последовательности *seq* и возвращает список результатов. Функции map() может быть передано более одной последовательности. В этом случае количество воспринимаемых объектом *function* аргументов должно быть равным количеству переданных последовательностей. Если один или несколько списков окажутся короче других, недостающие элементы будут считаться равными None. Если значение аргумента *function* равно None, подразумевается функция, возвращающая свой аргумент (для одной последовательности) или кортеж своих аргументов (для нескольких последовательностей). Аргументы-последовательности могут быть любого типа, результат всегда будет иметь тип list.

max(*seq*)
max(*v1*, *v2* ...)

С единственным аргументом *seq*, возвращает наибольший элемент последовательности. Если последовательность пустая, генерируется исключение ValueError. Если задано несколько аргументов, возвращает наибольший из них.

min(*seq*)
min(*v1*, *v2* ...)

С единственным аргументом *seq*, возвращает наименьший элемент последовательности. Если последовательность пустая, генерируется исключение ValueError. Если задано несколько аргументов, возвращает наибольший из них.

oct()

Возвращает восьмеричное представление целого (int или long int) аргумента. Результат является правильным литеральным выражением языка Python (для длинного целого в конце будет добавлен суффикс 'L'). Заметим, что результат всегда беззнаковый, например, на 32-разрядных платформах oct(-1) дает '037777777777'. См. также описание функции hex().

open(*filename* [, *mode* [, *bufsize*]])

Открывает файл и возвращает соответствующий ему файловый объект (см. раздел

11.7). *filename* — имя файла, *mode* — строка, указывающая, в каком режиме файл будет открыт. Возможные режимы: `'r'` — для чтения (файл должен существовать), `'w'` — для записи (если файл не существует, он создается, в противном случае его содержимое удаляется), `'a'` — для добавления в конец файла (если файл не существует, он создается; текущее положение в файле устанавливается указывающим на его конец; на некоторых платформах метод `seek()` игнорируется), `'r+'`, `'w+'` и `'a+'` — для чтения и записи (для открытия в режиме `'r+'` файл должен существовать, в режиме `'w+'` содержимое существующего файла удаляется). При добавлении символа '`b`' к режиму гарантируется, что файл будет открыт в двоичном режиме. На некоторых платформах файлы по умолчанию открываются в текстовом режиме, в котором операционная система автоматически заменяет некоторые символы. Во избежание возможной порчи данных лучше всегда принудительно выставлять двоичный режим (даже на платформах, где файл по умолчанию открывается в двоичном режиме — подумайте о переносимости Ваших программ). Если файл не может быть открыт, генерируется исключение `IOError`. Если аргумент *mode* опущен, он считается равным `'r'`.

Необязательный целый аргумент *bufsize* указывает желаемый размер буфера: `0` означает отсутствие буферизации, `1` — построчная буферизация, любое другое положительное число означает буфер (примерно) указанного размера. Если *bufsize* опущен или меньше нуля, используется системное значение по умолчанию[4]: обычно построчная буферизация для устройств tty и довольно большое значение для других файлов.

ord(*c*)

Возвращает код символа. Аргумент *c* должен быть строкой (`string` или `unicode`), состоящей из одного символа. Например, `ord('a')` возвращает `97`, `ord(u'\u2020')` возвращает `8224`. Обратное преобразование может быть выполнено с помощью функций `chr()` для обычных символов и `unichr()` для символов Unicode.

pow(*x*, *y* [, *z*])

С двумя аргументами возвращает *x* в степени *y* (эквивалентно '`x ** y`'). С заданным третьим аргументом, результат будет такой же, как и у выражения '`pow(x, y) % z`', однако вычисление производится более эффективно. Со смешанными типами операндов, применяются обычные правила для арифметических операций. Эффективный тип операндов также является и типом результата. Если результат не может быть выражен в рамках этого типа, генерируется соответствующее ситуации исключение. Например, `pow(2, -1)` генерирует исключение `ValueError`, а `pow(2, 35000)` генерирует исключение `OverflowError`.

range([*start*,] *stop* [, *step*])

Эта гибкая функция создает и возвращает арифметическую прогрессию от *start* (включительно; по умолчанию `0`) до *stop* (исключая само значение *stop*) с шагом *step* (по умолчанию `1`; если задан шаг, аргумент *start* становится обязательным). Все аргументы должны быть простыми целыми (`int`), если аргумент

[4]На системах, библиотека языка C которых не имеет функцию `setvbuf()`, размер буфера, независимо от *bufsize*, будет иметь значение по умолчанию.

step равен нулю, генерируется исключение ValueError. Если *step* — положительное число, последним элементом списка будет наибольшее число из ряда *start + i * step*, которое меньше, чем *stop* (*i* — натуральное число или ноль). Если *step* — отрицательное число, последним элементом списка будет наименьшее число из ряда *start + i * step*, которое больше *stop*. Несколько примеров:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

raw_input([*prompt*])

Если задан аргумент *prompt*, его значение (точнее строковое представление, полученное аналогично тому, как это делают функция str() и инструкция print) выводится на стандартный поток вывода (в отличие от инструкции print, символ новой строки в конце не добавляется). Функция считывает строку из стандартного потока ввода и возвращает ее (исключая завершающий символ перехода на новую строку). Если при считывании достигается конец файла, генерируется исключение EOFError. Например:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

Если загружен модуль readline, функция raw_input() будет использовать его, предоставляя возможности редактирования строки и использования истории введенных строк (некоторые операционные системы, например Windows NT, предоставляют аналогичные возможности автоматически).

reduce(*function*, *sequence* [, *initializer*])

Последовательно применяет *function* (объект, поддерживающий вызов с двумя аргументами) сначала к двум первым элементам последовательности *sequence*, затем к результату и третьему элементу и т. д. Например, ‘reduce(operator.__add__, [1, 2, 3, 4, 5])’ вычисляет сумму ((((1+2)+3)+4)+5). Если задан необязательный аргумент *initializer*, при вычислении он помещается перед первым элементом. Если общее количество элементов (*initializer*, если он задан, плюс элементы последовательности

sequence) равно 1, возвращается его (единственного элемента) значение, если общее количество элементов равно 0 (то есть не задан аргумент *initializer* и используется пустая последовательность), генерируется исключение TypeError.

reload(*module*)

Заново считывает и инициализирует уже импортированный модуль. Аргумент *module* должен быть объектом-модулем, представляющим ранее успешно импортированный модуль. Возвращает объект-модуль (тот же самый, что и *module*). Эта функция может быть полезна при одновременном редактировании и тестировании в интерактивном режиме (не покидая интерпретатора) модулей.

Если в исходном тексте модуля нет синтаксических ошибок, но они возникают во время его инициализации, первая инструкция import для этого модуля не создаст соответствующую ему локальную переменную, а лишь сохранит (частично инициализированный) объект-модуль в sys.modules. Для того, чтобы перезагрузить модуль, необходимо сначала повторно импортировать с помощью инструкции import (при этом создается локальная переменная, ссылающаяся на частично инициализированный объект-модуль) и лишь затем использовать функцию reload().

При перезагрузке модуля старый словарь, представляющий глобальное пространство имен модуля сохраняется. Новые определения перезаписывают старые, так что это обычно не вызывает проблем. Если новая версия не определяет какое-либо имя, для него остается прежнее значение. Эта особенность может быть использована модулем, если он поддерживает глобальную таблицу или кэш объектов: с помощью инструкции try Вы можете определить наличие таблицы и, при желании, пропустить ее инициализацию.

Перезагрузка встроенных и динамически связываемых модулей в принципе допустима (кроме модулей sys, __main__ и __builtin__), однако не настолько полезна. Некоторые дополнительные модули, однако, не приспособлены к повторной инициализации, что может привести к всевозможным ошибкам.

Если Вы импортируете объекты из модуля, используя инструкцию 'from ... import ...', перезагрузка модуля не приведет к переопределению этих объектов — после перезагрузки необходимо повторно выполнить инструкцию 'from ... import ...'.

Если Вы определили экземпляры класса, перезагрузка модуля, в котором определен класс, не изменит определения методов для экземпляров — они будут продолжать использовать старые определения. То же самое верно и для производных классов.

repr(*object*)

Возвращает строку, содержащую представление объекта *object*. Тот же эффект Вы можете получить, заключив выражение *object* в обратные кавычки (`*object*`). По возможности функция repr() возвращает строку, пригодную для восстановления объекта с тем же значением с помощью функции eval(). См. также описание функции str().

round(*x* [, *n*])

Возвращает вещественное число, полученное округлением *x* до *n* цифр после десятичной точки. По умолчанию *n* считается равным нулю. То есть возвращает

ближайшее к *x* число, кратное 10^{-n}. Из двух одинаково близких чисел выбирается то, которое находится дальше от нуля (то есть `round(0.5)` дает `1.0` и `round(-0.5)` дает `-1.0`).

setattr(*object, name, value*)
Присваивает атрибуту с именем *name* (строка) объекта *object* значение *value*. *name* может быть именем уже существующего или нового атрибута. Например, вызов '`setattr(`*x*`, '`*foobar*`', 123)`' эквивалентен инструкции '*x*`.`*foobar* `= 123`'.

slice([*start*,] *stop* [, *step*])
Возвращает объект среза (см. раздел 11.8.3). Если какой-либо необязательный аргумент опущен, используется `None` для обозначения значения по умолчанию.

str(*object*)
Возвращает строковое представление, наиболее пригодное для вывода. В отличие от `repr()`, функция `str()` даже не пытается создать строку, являющуюся правильным выражением языка Python. Точно такое представление используется для объектов при выводе их с помощью инструкции `print`.

tuple(*sequence*)
Возвращает кортеж, составленный из элементов последовательности *sequence*. Порядок следования элементов сохраняется. Если последовательность *sequence* уже является кортежем, он возвращается без изменений (нет смысла создавать его копию, так как кортежи являются неизменяемыми объектами) . Например, `tuple('abc')` возвращает `('a', 'b', 'c')`, а `tuple([1, 2, 3])` возвращает `(1, 2, 3)`.

type(*object*)
Возвращает тип объекта *object*. Возвращаемое значение является объектом типа (см. раздел 11.8.2). В стандартном модуле `types` определены имена для всех встроенных типов. Например:

```
>>> import types
>>> if type(x) == types.StringType:
...     print "Объект x является строкой"
```

unichr(*i*)
Возвращает строку Unicode, состоящую из одного символа, код которого равен *i*. Например, `unichr(97)` возвращает строку `u'a'`. Аргумент *i* должен быть целым числом от 0 до 65535 включительно, если *i* выходит за пределы указанного диапазона, генерируется исключение `ValueError`. Обратная операция выполняется функцией `ord()`. См. также описание функции `chr()`.

unicode(*string* [, *encoding* [, *errors*]])
Преобразует строку *string* из кодировки *encoding* (по умолчанию `'utf-8'`) в строку Unicode и возвращает результат. Поведение при возникновении ошибок определяется значением аргумента *errors*: `'strict'` (используется по умолчанию) означает, что при возникновении ошибки будет сгенерировано исключение

UnicodeError, 'ignore' — недопустимые символы игнорируются (удаляются), 'replace' — недопустимые символы заменяются универсальным символом замены ("REPLACEMENT CHARACTER", u'\uFFFD').

vars([*object*])
Без аргументов возвращает словарь, соответствующий локальному пространству имен. Если в качестве аргумента передан объект, представляющий модуль, класс или экземпляр класса (или любой другой объект, имеющий атрибут `__dict__`), возвращает словарь, соответствующий пространству имен объекта. Не следует вносить изменения в возвращаемый словарь: эффект на соответствующее пространство имен не определен[5].

xrange([*start*,] *stop* [, *step*])
Эта функция работает аналогично функции range(), но возвращает объект xrange (см. раздел 11.2.4) вместо списка. Этот тип последовательности позволяет получать точно такие же значения, как и соответствующий список (полученный с помощью функции range() с такими же аргументами), на самом деле не храня их. Преимущества функции xrange() над функцией range() обычно минимальны и проявляются только на очень длинных рядах чисел.

zip(*seq1* ...)
Возвращает список, каждый *i*-й элемент которого является кортежем, составленным из *i*-х элементов последовательностей *seq1*, ..., *seqN* в порядке их следования в списке аргументов. Требуется как минимум один аргумент, в противном случае, генерируется исключение TypeError. Длина возвращаемого списка равна длине самой короткой из последовательностей *seq1*, ..., *seqN*. Если все последовательности имеют одинаковую длину, функция zip ведет себя аналогично функции map с None в качестве первого аргумента. Функция доступна, начиная с версии 2.0.

[5] В текущих реализациях локальные переменные не могут быть изменены таким способом, а пространства имен объектов — могут. Ситуация может измениться в будущих версиях.

Глава 13

Встроенные классы исключений

Исключения в языке Python могут быть экземплярами классов и строками. Поддержка строк в качестве исключений оставлена для совместимости с ранними версиями и может быть удалена в будущем. Классы встроенных исключений определены в стандартном модуле `exceptions`. Нет необходимости явно импортировать этот модуль — все определенные в нем классы исключений всегда доступны аналогично другим встроенным объектам (определены в пространстве встроенных имен).

Если в ветви `except` инструкции `try` указывается объект-класс *C*, эта ветвь будет обрабатывать любое "совместимое" с указанным классом исключение, то есть если оно является экземпляром *C* или производного от него класса. Например, ветвь '`except ValueError:`' будет обрабатывать исключения `UnicodeError`[1], но '`except UnicodeError:`' не будет перехватывать исключения `ValueError`. Два класса исключений, ни один из которых не является производным (прямо или косвенно) от другого, никогда не будут эквивалентными, даже если они имеют одинаковое имя (под именем исключения подразумевается имя класса, то есть значение атрибута `__name__` объекта-класса).

Описанные ниже встроенные исключения генерируются интерпретатором, встроенными функциями и методами объектов встроенных типов. Исключения могут иметь ассоциированное с ним значение (аргумент или несколько аргументов, используемые при его инициализации), описывающее причину возникновения исключительной ситуации. Если класс исключения является производным от стандартного класса `Exception`, аргументы, используемые при инициализации, сохраняются в атрибуте `args`.

Пользовательский код также может генерировать исключения встроенных типов для тестирования обработчиков ошибок, а также, если необходимо сообщить об ошибке (или другой исключительной ситуации), для которой данный встроенный класс исключений предназначен.

Ниже приведена иерархия встроенных классов исключений.

`Exception`

Базовый класс для всех встроенных классов исключений. Рекомендуется также использовать его для всех пользовательских классов исключений. Атрибут `args` его

[1] Здесь и далее под "исключениями `SomeError`" или "исключениями типа `SomeError`" подразумеваются "исключения, являющиеся экземплярами класса `SomeError`", если речь идет о генерируемом исключении, и "исключения, являющиеся экземплярами `SomeError` и производных от него классов", если речь идет о перехватываемом исключении.

экземпляров содержит список аргументов, которые использовались при его инициализации. Встроенная функция `str()`, при применении к экземплярам класса `Exception`, возвращает строковое значение аргумента (если при инициализации использовался один аргумент), кортежа аргументов (если при инициализации использовалось более одного аргумента) или пустую строку (если конструктор вызывался без аргументов). Кроме того, Вы можете получить доступ (только для чтения) к аргументам, используя операцию получения элемента по индексу (*exc*[*i*]).

StandardError

Базовый класс исключений, предназначенных для генерации при возникновении стандартных ошибок (то есть всех ошибок, которые могут возникнуть при использовании встроенных средств языка).

ArithmeticError

Базовый класс исключений, генерируемых при возникновении арифметических ошибок.

FloatingPointError

Исключения этого класса генерируются, если возникают проблемы с выполнением операции с плавающей точкой (могут генерироваться только если Python сконфигурирован с опцией `--with-fpectl` или в файле 'config.h' определено имя `WANT_SIGFPE_HANDLER`).

OverflowError

Используется, когда результат арифметической операции слишком большой, чтобы его можно было представить в рамках используемого типа. Заметим, что этого не может произойти при использовании длинного целого типа (`long int`) — скорее будет сгенерировано исключение `MemoryError`. Из-за того, что обработка ошибок при операциях с плавающей точкой в языке C не стандартизована, большинство таких операций не проверяются. Для обычных целых чисел (`int`), все операции, которые могут привести к переполнению, проверяются, за исключением сдвига влево, так как в этом случае потеря старших бит считается предпочтительнее генерации исключения.

ZeroDivisionError

Используется, если второй операнд операторов деления и получения остатка от деления равен нулю.

AssertionError

Исключения этого класса генерируются, если не выполняется условие, указанное в инструкции `assert`.

AttributeError

Исключения этого класса генерируются в случаях, когда невозможно получить значение атрибута с определенным именем, удалить его или присвоить ему новое значение. Если объекты данного типа вообще не поддерживают используемый доступ к атрибутам (получение значения, удаление или присваивание), генерируется исключение `TypeError`.

EOFError

Исключения этого класса генерируются некоторыми встроенными функциями (`input()`, `raw_input()`), если достигается конец файла и, при этом, из него не считано ни байта информации. Обратите внимание на то,

что методы `read()` и `readline()` файловых объектов при достижении конца файла возвращают пустую строку.

EnvironmentError

Базовый класс исключений, генерируемых при ошибках, возникающих за пределами интерпретатора Python. Если при инициализации исключений этого типа используются 2 или 3 аргумента, первый из них считается кодом ошибки, второй — строкой, поясняющей причину ее возникновения, и третий (необязательный) — имя файла, с которым связана возникшая ошибка. В этом случае аргументы (помимо унаследованного атрибута `args`) сохраняются соответственно в атрибутах `errno`, `strerror` и `filename` и используются для создания строкового представления (функция `str()`) в виде `'[Errno` *`errno`*`]` *`strerror`*`'` или `'[Errno` *`errno`*`]` *`strerror`*`:` *`filename`*`'`.

IOError

Используется при ошибках ввода/вывода, например, если не найден файл с указанным именем или на носителе нет свободного места.

OSError

Исключения этого класса (доступного также как `os.error`) в основном генерируются функциями модуля `os`.

WindowsError

Используется при ошибках, специфичных для операционных систем Windows или код ошибки не соответствует значению `errno`. Значения атрибутов `errno` и `strerror` создаются из значений, возвращаемых функциями Windows API `GetLastError()` и `FormatMessage()` соответственно. Добавлен в версии 1.6.

ImportError

Исключения этого класса генерируются, если модуль с указанным в инструкции `import` именем не может быть импортирован или в модуле не найдено имя, указанное в инструкции `from ... import ...`.

KeyboardInterrupt

По умолчанию, если интерпретатор получает сигнал `SIGINT`, например, при прерывании с клавиатуры (обычно клавишами `Ctrl-C`), генерируется исключение `KeyboardInterrupt`. Вы можете изменить реакцию на сигнал `SIGINT`, воспользовавшись функцией `signal.signal()`. Прерывание в момент ожидания ввода, при использовании функций `input()` и `raw_input()`, а также методов `read()`, `readline()` и `readlines()` файлового объекта, представляющего устройство tty, также генерируется исключение `KeyboardInterrupt`[2].

LookupError

Базовый класс исключений, генерируемых, если последовательность или отображение не содержит элемента с заданным индексом или ключом. Может быть сгенерировано напрямую функцией `sys.setdefaultencoding()`.

[2] На самом деле, поведение зависит от используемой платформы. Например, под Windows NT при прерывании в момент работы функций `input()` и `raw_input()` сначала будет сгенерировано исключение `EOFError`.

IndexError

Используется в случаях, когда индекс, используемый для получения элемента последовательности, выходит за пределы диапазона. Заметим, что индексы среза в этом случае считаются указывающими на конец или начало последовательности, и ошибки не возникает. Если индекс не является обычным целым числом, генерируется исключение `TypeError`.

KeyError

Исключения этого класса генерируются, если в отображении не найдена запись с указанным ключом. В качестве аргумента при инициализации экземпляра исключения обычно используется ключ.

MemoryError

Используется, если для выполнения операции не хватает памяти, но из сложившейся ситуации еще можно выйти, удалив некоторые объекты. Экземпляр исключения может быть инициализирован строкой, указывающей род (внутренней) операции, для выполнения которой не хватило памяти. Заметим, что лежащая в основе интерпретатора архитектура управления памятью (функция `malloc()` языка C) не всегда позволяет полностью восстановиться из подобных ситуаций. В любом случае генерируется исключение `MemoryError` и сообщение об ошибке (включая информацию о месте ее возникновения) может быть выведена в стандартный поток ошибок.

NameError

Исключения этого класса генерируются, если используемое в программе локальное или глобальное имя не найдено. Это имя используется в качестве аргумента при инициализации экземпляра исключения.

UnboundLocalError

Используется в случае, если Вы неявно ссылаетесь на глобальную переменную (не использовав предварительно инструкцию `global`), а затем пытаетесь удалить ее или присвоить ей новое значение. Добавлен в версии 1.6.

RuntimeError

Используется при ошибках времени выполнения, которые не попадают ни в одну из описанных выше категорий. В качестве аргумента при инициализации используется строка с информацией об ошибке. Это исключение является пережитком прошлых версий языка и в настоящее время используется не слишком часто.

NotImplementedError

Исключения этого класса следует использовать при определении абстрактных методов пользовательских классов-интерфейсов для того, чтобы показать, что метод должен быть переопределен в производных классах.

SyntaxError

Используется синтаксическим анализатором при обнаружении синтаксических ошибок. Подобные ошибки могут возникнуть при использовании инструкций `import` и `exec`, функций `eval()` и `input()`, а также при чтении инструкций из файла или стандартного потока ввода (в том числе

в интерактивном режиме).

Экземпляры этого класса исключений имеют атрибуты: `filename` и `lineno` — имя файла и номер строки в которых обнаружена ошибка, `text` — текст строки, содержащей ошибку, `offset` — место в строке, где ошибка была обнаружена синтаксическим анализатором, и `msg` — сообщение об ошибке. Экземпляр исключения инициализируется в виде '`SyntaxError(`*msg*`, (`*filename*`, `*lineno*`, `*offset*`, `*text*`))`'. Функция `str()`, применительно к экземплярам исключения, возвращает только сообщение об ошибке (атрибут `msg`).

SystemError

Исключения этого класса генерируются в случае обнаружения внутренних ошибок. В качестве аргумента при инициализации используется строка, с подробностями (используется терминология низкого уровня). О возникновении подобных ошибок следует сообщать авторам языка.

TypeError

Исключения этого класса генерируются, какая-либо операция применяется к объекту несоответствующего типа. В качестве аргумента при инициализации исключения используется строка, указывающая на детали несоответствия типа.

ValueError

Исключения этого класса генерируются, какая-либо операция применяется к объекту правильного типа, но имеющего несоответствующее значение, и ситуация не может быть описана более точно с помощью исключения другого типа (например, `IndexError`).

UnicodeError

Используется при ошибках преобразования строки в определенной кодировке в строку Unicode и наоборот. Добавлен в версии 1.6.

SystemExit

Исключения этого класса генерируются функцией `sys.exit()`. Если исключение не обрабатывается, выполнение программы молча прерывается. В качестве аргумента при инициализации используется целое число — код ошибки, передаваемый функции `exit()` языка C. По умолчанию и, если используется `None`, код завершения считается равным `0`. Если при инициализации использован аргумент другого типа или несколько аргументов, аргумент или кортеж аргументов выводится в стандартный поток ошибок. В любом случае, аргумент или кортеж аргументов сохраняется в виде атрибута `code`. `SystemExit` является производным непосредственно от `Exception`, а не `StandardError`, так как фактически исключения `SystemExit` не являются ошибкой.

Вызов функции `sys.exit()` преобразуется в генерацию исключения для того, чтобы могла быть выполнена очистка (ветвь `finally` инструкции `try`) и отладчик мог выполнить программу без потери контроля. Если все же необходимо немедленно прервать выполнение программы (например, после вызова функции `os.fork()` в дочернем процессе), воспользуйтесь функцией `os._exit()`.

Часть III

Библиотека стандартных модулей

Описание языка Python было бы неполным без описания библиотеки — огромной коллекции модулей. Некоторые модули написаны на C и встраиваются в интерпретатор, другие написаны на языке Python и доступны в исходном виде. Некоторые модули предоставляют услуги, характерные для языка Python (например, вывод сообщений об ошибках) или определенной операционной системы (например, доступ к определенным аппаратным средствам), другие определяют интерфейсы, характерные для некоторой области применения (например, WWW).

К сожалению, невозможно в рамках однотомного издания описать даже стандартные модули, обычно поставляемые вместе с интерпретатором языка. Поэтому мы постараемся остановиться на основных, наиболее универсальных.

Глава 14

Конфигурационные модули

14.1 site — общая конфигурация

Этот модуль **автоматически импортируется при каждом запуске интерпретатора** (за исключением случаев, когда используется опция командной строки **-S**) и предназначен, в первую очередь, для добавления путей поиска модулей, характерных для данной машины. По умолчанию в пути поиска добавляются каталоги с именами 'sys.prefix + '/lib/site-packages'' и 'sys.prefix + '/site-python'' (UNIX) или sys.prefix (другие платформы). Кроме того, модуль обрабатывает конфигурационные файлы путей вида '*package*.pth' во всех указанных каталогах. Эти конфигурационные файлы должны содержать дополнительные пути (по одному каталогу в строке), которые будут включены в sys.path. Пустые строки и строки, начинающиеся с '#', игнорируются.

Например, пусть стандартные библиотеки установлены в каталоге '/usr/lib/python1.5', в котором присутствует подкаталог 'site-packages'. Пусть каталог '/usr/lib/python1.5/site-packages' в свою очередь содержит вложенные каталоги 'foo', 'bar' и 'spam' и конфигурационные файлы 'foo.pth' и 'bar.pth'. Предположим, файл 'foo.pth' содержит следующие строки:

```
# foo package configuration

foo
bar
bletch
```

и 'bar.pth' содержит:

```
# bar package configuration

bar
```

Тогда в sys.path будут добавлены каталоги '/usr/lib/python1.5/site-packages/bar' и '/usr/lib/python1.5/site-packages/foo'.

Обратите внимание, что каталоги 'bletch' и 'spam' не добавляются, так как не существует файла или каталога с именем 'bletch' и каталог 'spam' не упомянут ни в одном из конфигурационных файлов.

После выполнения перечисленных манипуляций с путями, производится попытка импортировать модуль sitecustomize, в который может выполнять дополнительные настройки. Если при импортировании генерируется исключение ImportError, оно молча игнорируется.

14.2 user — конфигурация пользователя

В целях безопасности интерпретатор не выполняет автоматически конфигурационный файл пользователя. Только в интерактивном режиме интерпретатор выполняет файл, указанный в переменной окружения PYTHONSTARTUP. Однако некоторые программы могут позволить загрузить стандартный конфигурационный файл пользователя. Программа, желающая использовать такой механизм, должна выполнить инструкцию 'import user'.

Модуль user ищет файл с именем '.pythonrc.py' в домашнем каталоге пользователя и выполняет его с помощью встроенной функции execfile() в своем собственном (модуля user) глобальном пространстве имен. Ошибки, возникающие при выполнении '.pythonrc.py' модулем user не обрабатываются. Если домашний каталог не может быть определен, выполняется файл '.pythonrc.py' в текущем каталоге.

Будьте сдержаны в том, что Вы помещаете в свой файл '.pythonrc.py'. Так как Вы не знаете, какие программы будут его использовать, изменение поведения стандартных модулей и функций вряд ли будет хорошей идеей.

Рекомендация для программистов, желающих использовать описанный здесь механизм: простейший способ устанавливать параметры — определить переменные в файле '.pythonrc.py'. Например, модуль spam может определить, насколько подробными пользователь хочет видеть его сообщения, с помощью следующих инструкций:

```
import user
try:
    # значение, установленное пользователем
    verbose = user.spam_verbose
except AttributeError:
    # значение по умолчанию
    verbose = 0
```

Программам, имеющим большое количество пользовательских параметров, следует использовать отдельный конфигурационный файл. Программы, имеющие отношение к безопасности, а также модули общего назначения, *не* должны импортировать модуль user.

Глава 15

Служебные модули

Модули, описанные в этой главе, являются служебными по отношению к интерпретатору языка Python и его взаимодействию с окружением.

sys	Доступ к характерным для системы параметрам и функциям.
atexit	Регистрирует функции и выполняет их при окончании работы программы
types	Имена для всех встроенных типов.
operator	Операторы языка Python в виде функций.
traceback	Модуль для работы с объектами `traceback`.
imp	Доступ к операциям, производимым инструкцией `import`.
pprint	Представление и вывод данных в более привлекательном виде.
repr	Альтернативная реализация функции `repr()` с ограничением размера.

15.1 `sys` — характерные для системы параметры и функции

Этот модуль предоставляет доступ к переменным, используемым или поддерживаемым интерпретатором, и функциям, которые интенсивно взаимодействуют с интерпретатором. Модуль `sys` всегда доступен.

argv

Список аргументов, переданных в командной строке программе на языке Python. `argv[0]` является именем программы (является имя полным или нет, зависит от используемой операционной системы). Если интерпретатор запущен с опцией `-c`, `argv[0]` является строкой `'-c'`. При чтении команд со стандартного потока ввода (в том числе в интерактивном режиме) `argv[0]` является строкой `'-'` или `''` в зависимости от того, использовалась опция '–' или нет.

byteorder

Строка, указывающая характерный для платформы порядок следования байтов: `'big'` (big-endian) или `'little'` (little-endian). Переменная доступна, начиная с версии 2.0.

builtin_module_names
Кортеж строк — имен всех модулей, встроенных в интерпретатор. Эта информация не может быть получена никаким другим путем (`modules.keys()` дает список имен импортированных модулей).

copyright
Строка, содержащая авторские права на интерпретатор языка Python.

dllhandle (Windows)
Целое число, дескриптор DLL интерпретатора Python.

exc_info()
Возвращает кортеж из трех значений, описывающих исключение, которое в данный момент обрабатывается (в текущем потоке). Под обработкой исключения здесь подразумевается выполнение ветви `except` (в том числе и вызванных из нее функций) инструкции `try` (см. раздел 10.4.4).

Если в настоящий момент обработка исключения не выполняется, возвращаемый кортеж содержит три объекта `None`. В противном случае, возвращается ‘(*type*, *value*, *traceback*)’, где *type* — тип исключения (класс или строка), *value* — значение исключения (экземпляр класса) или ассоциированное значение (строка) и *traceback* — объект `traceback` (см. раздел 11.9.3), представляющий пройденный путь от места возникновения исключительной ситуации.

Важное замечание: присваивание *traceback* локальной переменной в функции, обрабатывающей исключение, приведет к образованию циклических ссылок между объектами. Таким образом, удаление объекта будет возможным, только если интерпретатор собран со сборщиком мусора (см. описание модуля `gc`). Так как в большинстве случаев объект `traceback` не нужен, наилучшим решением будет использование функции в виде ‘`type, value = sys.exc_info()[:2]`’ для получения только типа и значения исключения. Если Вам необходим объект `traceback`, удостоверьтесь в том, что Вы удаляете его после использования (лучше всего это делать в ветви `finally` инструкции `try`) или используйте `exc_info()` в отдельной функции, вызванной из ветви `except`.

exc_type

exc_value

exc_traceback
Присутствуют только для совместимости со старыми версиями — используйте функцию `exc_info()`. Так как они являются глобальными переменными, содержат значения для исключения, обработка которого была начата последней. Таким образом, их использование не безопасно в многопоточных программах. Если в данный момент исключение не обрабатывается, значение `exc_type` равно `None` и переменные `exc_value` и `exc_traceback` не определены.

exec_prefix
Строка, содержащая начало пути к зависимым от платформы файлам интерпретатора Python (зависит от параметров сборки и/или установки). Например, конфигурационные файлы на платформах UNIX располагаются в каталоге ‘`sys.exec_prefix + '/lib/python' + sys.version[:3] + '/config'`’.

executable

Строка с (полным) именем исполняемого файла интерпретатора Python, если оно имеет смысл для данной системы.

exit([*exit_code*])

Прерывает выполнение программы. Реализуется путем генерации исключения `SystemExit`, то есть страховочный код (ветвь `finally` инструкции `try`) принимается во внимание и попытка выхода может быть прервана на внешнем уровне путем обработки исключения. Необязательный аргумент может быть целым числом (по умолчанию `0`, что соответствует успешному завершению; многие системы требуют, чтобы это число было от 0 до 127), которое будет использовано в качестве кода завершения, или объект любого другого типа. В последнем случае использование `None` эквивалентно `0`, для других объектов на стандартный поток ошибок выводится строковое представление (полученное аналогично тому, как это делает встроенная функция `str()`) и в качестве кода завершения используется `1`. В частности инструкция '`sys.exit('Сообщение об ошибке')`' является быстрым способом прервать выполнение программы с выводом сообщения об ошибке.

exitfunc

Эта переменная на самом деле не определена самим модулем, но Вы можете присвоить ей объект-функцию, которая будет вызвана (без аргументов) при завершении работы интерпретатора. Функция `exitfunc()` не вызывается, если процесс прерван с помощью некоторых сигналов, при возникновении фатальной ошибки и при вызове функции `os._exit()`. Мы настоятельно не рекомендуем устанавливать переменную `exitfunc` напрямую (это может привести к некорректной работе других компонент программы) — воспользуйтесь модулем `atexit`, который предоставляет возможность регистрировать несколько функций.

getdefaultencoding()

Возвращает текущую кодировку, используемую по умолчанию при преобразовании обычных строк в строки Unicode и наоборот. Функция доступна, начиная с версии 2.0.

getrefcount(*object*)

Возвращает число ссылок на объект *object*. Число будет на единицу больше, чем Вы, возможно, ожидали, так как оно включает временно созданную ссылку на объект как аргумент функции `getrefcount()`.

getrecursionlimit()

Возвращает текущее ограничение рекурсии — максимальную глубину стека интерпретатора Python. Это ограничение предотвращает переполнение стека при бесконечных рекурсиях. Ограничение рекурсии может быть установлено с помощью функции `setrecursionlimit()`. Эта функция доступна, начиная с версии 2.0.

hexversion

Номер версии в виде одного целого числа. Гарантируется, что это число будет возрастать с каждой новой версией, включая отладочные выпуски. Например, для того, чтобы убедиться, что версия интерпретатора не меньше, чем 1.5.2 (финальный выпуск), воспользуйтесь следующим кодом:

```
if sys.hexversion >= 0x010502F0:
    # Используем новые возможности.
    ...
else:
    # Используем альтернативную реализацию или
    # выводим предупреждение.
    ...
```

Функция названа 'hexversion' потому, что возвращаемое ей значение выглядит действительно осмысленным, только если смотреть на его шестнадцатеричное представление (например, полученное с помощью встроенной функции hex()). Информация о версии в более дружественном виде хранится в переменной version_info.

last_type
last_value
last_traceback

Эти три переменные не всегда определены. Им присваиваются значения в случае, если исключение не обрабатывается и интерпретатор выводит информацию об ошибке. Наличие этих переменных позволяет пользователю в интерактивном режиме импортировать модуль отладчика и сделать "вскрытие трупа" программы, исключив необходимость повторного запуска команд, приведших к возникновению ошибки. Для этого достаточно выполнить 'import pdb; pdb.pm()' (см. главу 25.1).

Значения переменных аналогичны элементам кортежа, возвращаемого функцией exc_info() (так как может быть только один интерактивный поток, использование глобальных переменных безопасно).

maxint

Наибольшее целое число, которое может быть представлено в рамках типа int. Это число не меньше, чем $2^{31} - 1$. Наименьшее представимое в рамках int целое число на большинстве платформ равно -maxint-1.

modules

Словарь, отображающий имена импортированных модулей в объекты-модули. Вы можете манипулировать им, например, для принудительной перезагрузки модулей или для того, чтобы сделать доступными для импортирования динамически созданные модули (см. описание модуля new). Заметим, что последовательное удаление модуля из modules и повторное его импортирование — это не совсем то, что делает встроенная функция reload().

path

Список строк — путей поиска модулей. Изначально эта переменная содержит каталог, в котором находится исполняемая в данный момент программа (или пустую строку, если команды считываются со стандартного потока ввода; пустая строка используется для обозначения текущего каталога), пути, указанные в переменной окружения PYTHONPATH и зависящие от платформы пути по умолчанию. Обратите внимание, что каталог, в котором находится программа, идет первым (path[0]).

platform

Строка, содержащая идентификатор платформы, например, 'sunos5', 'linux1' или 'win32'. Может использоваться, например, для добавления характерных для платформы путей в path.

prefix

Строка, содержащая начало пути к независимым от платформы файлам интерпретатора Python (зависит от параметров сборки и/или установки). Например, основная коллекция модулей на платформах UNIX устанавливается в каталог 'sys.prefix + '/lib/python' + sys.version[:3]', в то время как независимые от платформы заголовочные файлы (все, кроме 'config.h') располагаются в 'sys.prefix + '/include/python' + sys.version[:3]'.

ps1
ps2

Обычно строки, определяющие, соответственно, первичное и вторичное приглашения интерпретатора. Определены и используются, только если интерпретатор находится в интерактивном режиме. В этом случае они имеют начальные значения '>>> ' и '... '. Если значения этих переменных не являются строковыми объектами, в качестве приглашений используются их строковые значения, полученные аналогично встроенной функции str(). Строковое значение каждый раз вычисляется заново — такое поведение может быть использовано для реализации динамических приглашений.

setcheckinterval(*interval*)

Устанавливает интервал, через который интерпретатор будет проверять (контролировать) такие вещи, например, как переключение между потоками и обработчики сигналов. *interval* — целое число (по умолчанию 10), количество выполненных байт-инструкций языка между проверками. Увеличивая интервал, Вы можете повысить производительность многопоточных программ, уменьшая — ускорить реакцию. При значениях интервала меньше или равных нулю, проверка производится после выполнения каждой байт-инструкции, обеспечивая максимальную оперативность в ущерб производительности.

setdefaultencoding(*encoding*)

Устанавливает кодировку, которая будет использоваться по умолчанию при преобразовании обычных строк в строки Unicode и наоборот. Если строка *encoding* не соответствует доступной кодировке, генерирует исключение LookupError. Эта функция предназначена для использования в модуле site (или sitecustomize), после этого она обычно удаляется из пространства имен модуля sys. Функция доступна, начиная с версии 2.0.

setprofile(*profilefunc*)

Устанавливает системную функцию, реализующую замер производительности (profiling) различных компонент программы (см. главу 25.2). Функция *profilefunc* будет вызываться аналогично трассировочной функции (см. описание функции settrace()), но не для каждой выполненной строки кода, а только при вызове и возврате из функций и при генерации исключений.

setrecursionlimit(*limit*)
Устанавливает максимальную глубину стека интерпретатора Python. Это ограничение предотвращает переполнение стека при бесконечных рекурсиях. Максимальная возможная глубина стека зависит от платформы. Вам может понадобиться установить более высокое значение, если Ваша программа использует глубокие рекурсивные вызовы. Однако делать это следует с осторожностью, так как слишком большое значение может привести к аварийному завершению работы интерпретатора. Эта функция доступна, начиная с версии 2.0.

settrace(*tracefunc*)
Устанавливает системную трассировочную функцию, которая будет вызываться для каждой выполненной строки кода. Это позволяет в рамках языка Python реализовать отладчик (см. главу 25.1).

stdin
stdout
stderr
Файловые объекты, соответствующие стандартным потокам ввода, вывода и ошибок интерпретатора. stdin используется интерпретатором для операций ввода, таким как input() и raw_input(). stdout используется интерпретатором для операций вывода с помощью инструкции print, инструкций-выражений и для вывода приглашений функциями input() и raw_input(). Для вывода собственных приглашений интерпретатора в интерактивном режиме и вывода сообщений об ошибках (не перехваченных исключениях) используется stderr. Для корректной работы встроенных возможностей языка Python stdout и stderr должны быть объектами (произвольного типа), имеющими метод write(). Для корректной работы функций input() и raw_input() объект, на который ссылается stderr должен иметь метод readline(). Заметим, что изменение этих переменных не оказывает влияния на стандартные потоки ввода/вывода порождаемых (например, функциями os.popen() или os.system()) процессов.

__stdin__
__stdout__
__stderr__
Эти переменные хранят исходные значения для stdin, stderr и stdout при запуске программы и могут быть использованы для их восстановления.

tracebacklimit
Если этой переменной присвоено целое значение, она определяет максимальное число уровней информации, представляемой объектом traceback, которое будет выводиться при возникновении (необрабатываемой) исключительной ситуации. По умолчанию выводится не более 1000 уровней. Значение меньше или равное нулю запрещает вывод такой информации — выводится только тип исключения и его строковое представление.

version
Строка, содержащая информацию о версии, номере и времени сборки интерпретатора Python и компиляторе, который был использован, в виде '*version* (#*build_number*, *build_date*, *build_time*) [*compiler*]'. Первые три

символа используются для идентификации версии в именах каталогов, в которые производится установка. `version` используется в первой выводимой при запуске интерпретатора в интерактивном режиме строке: `'Python '+sys.version+' on '+platform`.

version_info

Кортеж из пяти объектов, описывающих номер версии: ‘(*major*, *minor*, *micro*, *releaselevel*, *serial*)’. Все элементы, кроме *releaselevel* являются целыми числами. *releaselevel* может иметь одно из четырех значений: `'alpha'`, `'beta'`, `'candidate'` или `'final'`. Значение `version_info`, соответствующее финальному выпуску Python 2.0, будет ‘`(2, 0, 0, 'final', 0)`’. Переменная определена, начиная с версии 1.6.

winver (Windows)

Строка с версией интерпретатора, используемая для имени ключа системного реестра на платформах Windows (обычно первые три символа переменной `version`). Переменная предоставляется в информационных целях, ее изменение не оказывает влияние на имя используемого интерпретатором ключа.

15.2 gc — управление “сборщиком мусора”

Этот модуль предоставляет интерфейс к необязательному “сборщику мусора” (garbage collector, GC): возможность отключить его, настроить частоту работы, установить отладочные параметры, а также предоставляет доступ к “потерянным”[1] объектам, найденным “сборщиком мусора”, которые, однако, не могут быть удалены. Так как “сборщик мусора” дополняет механизм подсчета ссылок, уже используемый интерпретатором Python, Вы можете отключить его, если уверены, что программа не создает циклических ссылок. Автоматический сбор может быть отключен вызовом `gc.disable()`. Для поиска утечек памяти при отладке, следует вызвать `gc.set_debug(gc.DEBUG_LEAK)`.

Модуль `gc` предоставляет следующие функции:

enable()

Включает автоматический “сбор мусора”.

disable()

Выключает автоматический “сбор мусора”.

isenabled()

Возвращает `1`, если автоматический “сбор мусора” включен, иначе возвращает `0`.

[1]“Потерянными” (или недоступными) мы будем называть объекты, на которые не ссылается ни одна переменная. Единственный способ удалить их или получить к ним доступ — воспользоваться “сборщиком мусора”.

collect()

Запускает полный “сбор мусора” и возвращает общее количество найденных недоступных объектов. Список объектов, которые не могут быть удалены “сборщиком мусора”, становится доступными через переменную `garbage`.

set_debug(*flags*)

Устанавливает параметры отладки. Вся отладочная информация будет выводиться на стандартный поток ошибок. Отладочные флаги (см. ниже) можно комбинировать с помощью оператора битового ИЛИ. Полностью отключить отладку можно с помощью инструкции `set_debug(0)`.

get_debug()

Возвращает параметры отладки, которые в настоящий момент установлены.

set_threshold(*threshold0* [, *threshold1* [, *threshold2*]])

Устанавливает пороги запуска (частоту) автоматического “сбора мусора”. Значение *threshold0* равное нулю отключает автоматический “сбор мусора”.

Различаются три поколения объектов в зависимости от того, сколько запусков сборщика они пережили. Новые объекты — самые молодые, относятся к поколению `0`. После каждого запуска сборщика объект “взрослеет”, перемещается в поколение `1`, затем в поколение `2`. Так как поколение `2` самое старшее, объекты остаются в нем после последующих запусков сборщика. Для того, чтобы решить, когда производить запуск, сборщик подсчитывает количество созданных и удаленных объектов после последнего запуска. Когда разница между ними достигает *threshold0*, запускается “сборщик мусора”. Изначально анализируется только самое молодое поколение. Если поколение `0` анализировалось более *threshold1* раз после последнего анализа поколения `1`, также анализируется и поколение `1`. Аналогично *threshold2* контролирует количество запусков сборщика для поколения `1`, перед тем как обрабатывать поколение `2`.

get_threshold()

Возвращает кортеж текущих порогов запуска “сборщика мусора” в виде ‘(*threshold0*, *threshold1*, *threshold2*)’.

Следующая переменная доступна только для чтения:

garbage

Список недоступных объектов, найденных “сборщиком мусора”, которые не могут быть удалены. Экземпляры классов, имеющие специальный метод `__del__()` и входящие в цикл ссылающихся друг на друга объектов, делают весь цикл неудаляемым. Если установлена опция отладки `DEBUG_SAVEALL`, в этой переменной сохраняются все найденные недоступные объекты (“сборщик мусора” их при этом не удаляет).

Для установки параметров отладки модуль предоставляет следующие константы-флаги:

DEBUG_STATS
Выводить статистику при запусках сборщика. Эта информация может быть использована при выборе частоты запусков.

DEBUG_COLLECTABLE
Выводить информацию о найденных недоступных объектах, которые подлежат удалению.

DEBUG_UNCOLLECTABLE
Выводить информацию о найденных недоступных объектах, которые не могут быть удалены. Эти объекты будут добавлены в список `garbage`.

DEBUG_INSTANCES
Если установлен флаг `DEBUG_COLLECTABLE` или `DEBUG_UNCOLLECTABLE`, выводить информацию о найденных экземплярах классов.

DEBUG_OBJECTS
Если установлен флаг `DEBUG_COLLECTABLE` или `DEBUG_UNCOLLECTABLE`, выводить информацию о найденных объектах, не являющихся экземплярами классов.

DEBUG_SAVEALL
При использовании этого флага все найденные недоступные объекты будут добавляться в `garbage`, в том числе и те, которые могут быть удалены.

DEBUG_LEAK
Флаги, необходимые для вывода информации, полезной при поиске утечек памяти (`DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_INSTANCES | DEBUG_OBJECTS | DEBUG_SAVEALL`).

15.3 `atexit` — выполнение действий при окончании работы программы

Доступен, начиная с версии 2.0.

Модуль `atexit` определяет единственную функцию, предназначенную для регистрации. Зарегистрированные с помощью нее функции будут автоматически вызваны при нормальном окончании работы программы. Функции не будут вызваны, если процесс программы был прерван с помощью некоторых сигналов, возникла фатальная внутренняя ошибка интерпретатора, а также при вызове функции `os._exit()`.

Модуль предоставляет альтернативный интерфейс к функциональности, предоставляемой переменной `sys.exitfunc`. Таким образом, модуль может работать некорректно при использовании кода, который устанавливает `sys.exitfunc` напрямую. В частности, стандартные модули языка Python могут использовать `atexit` не предупреждая об этом. Во избежание конфликтов, авторам, использующим `sys.exitfunc` следует внести соответствующие изменения в программы.

register(*func ...*)

Регистрирует *func* как функцию, которая должна быть выполнена при завершении работы программы. Все остальные аргументы (позиционные и именованные) будут сохранены и использованы для вызова *func*.

При нормальном завершении выполнения программы (завершение выполнения основного модуля, вызов функции sys.exit() или возникновение исключительной ситуации), все зарегистрированные функции вызываются в порядке, обратном порядку их регистрации. Предполагается, что модули низкого уровня обычно импортируются раньше модулей высокого уровня и, таким образом, вызываемые в конце функции для них выполняются позже.

Следующий пример демонстрирует, как модуль может инициализировать счетчик из файла при импортировании и сохранять его новое значение автоматически при завершении работы программы.

```
try:
    _count = int(open("/tmp/counter").read())
except IOError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    open("/tmp/counter", "w").write("%d" % _count)

import atexit
atexit.register(savecounter)
```

См. также реализацию модуля readline, демонстрирующую использование возможностей модуля atexit для чтения и записи файлов истории команд.

15.4 types — имена для всех встроенных типов

Этот модуль определяет имена для всех встроенных типов объектов, используемых интерпретатором (сами типы описаны в главе 11). Вполне безопасно использовать инструкцию ‘from types import *’ — модуль не определяет никаких имен, кроме тех, которые здесь перечислены. Все имена, которые будут добавлены в будущих версиях этого модуля, будут иметь окончание ‘Type’.

Имена для объектов-типов чаще всего используют в функциях, если действия, которые необходимо выполнить, зависят от типа аргумента. Например:

```
from types import *
def delete(list, item):
    if type(item) is IntType:
       del list[item]
    else:
       list.remove(item)
```

Модуль определяет следующие имена:

NoneType
: Объект, имеющий тип `None`. Существует только один объект этого типа — `None` (пустой объект).

TypeType
: Объект, имеющий тип `type` (тип) — тип объектов, возвращаемых встроенной функцией `type()`. Все объекты, определенные в этом модуле, являются объектами именно этого типа.

IntType
: Объект, имеющий тип `int` (целое число, например, `1`).

LongType
: Объект, имеющий тип `long int` (длинное целое число, например, `1L`).

FloatType
: Объект, имеющий тип `float` (вещественное число, например, `1.0`).

ComplexType
: Объект, имеющий тип `complex` (комплексное число, например, `1.0j`).

StringType
: Объект, имеющий тип `string` (строка, например, `'Python'`).

UnicodeType
: Объект, имеющий тип `unicode` (строка Unicode, например, `u'Python'`).

TupleType
: Объект, имеющий тип `tuple` (кортеж, например, ‘`(1, 2, 3)`’).

XRangeType
: Объект, имеющий тип `xrange`. Объекты этого типа возвращаются встроенной функцией `xrange()`.

BufferType
: Объект, имеющий тип `buffer` (буфер, создается встроенной функцией `buffer`).

ListType
: Объект, имеющий тип `list` (список, например, ‘`[1, 2, 3]`’).

DictType

DictionaryType

Два альтернативных имени для объекта, представляющего тип dictionary (словарь, например, '{'язык': 'Python', 'версия': '2.0'}').

FunctionType

LambdaType

Два альтернативных имени для объекта, представляющего тип function (функция, определенная пользователем). Такой тип имеют объекты, определенные с помощью инструкции def или оператора lambda.

CodeType

Объект, имеющий тип code (объект кода, например, возвращаемый встроенной функцией compile()).

ClassType

Объект, имеющий тип class (класс). Объект-класс создается инструкцией class.

InstanceType

Объект, имеющий тип instance (экземпляр класса). Экземпляры создаются при применении операции вызова к объекту-классу.

MethodType

UnboundMethodType

Два альтернативных имени для объекта, представляющего тип method (метод, определенный пользователем).

BuiltinFunctionType

BuiltinMethodType

Два альтернативных имени для объекта, представляющего тип builtin_function_or_method (встроенная функция или метод, например, abs()).

ModuleType

Объект, имеющий тип module (модуль).

FileType

Объект, имеющий тип file (файловый объект, например, создаваемый встроенной функцией open()).

SliceType

Объект, имеющий тип slice (срез). Объекты этого типа создаются при использовании расширенной записи среза и встроенной функцией slice().

EllipsisType

Объект, имеющий тип ellipsis. Существует только один объект этого типа — Ellipsis (эллипсис), он указывает на использование троеточия в расширенной записи среза.

TracebackType

Объект, имеющий тип traceback. Объекты этого типа предназначены для отслеживания пути между местом возникновения исключительной ситуации и местом ее обработки.

FrameType

Объект, имеющий тип frame (кадр стека). Объекты этого типа представляют окружение, в котором выполняется блок кода.

15.5 operator — операторы в виде функций

Модуль operator определяет набор функций, реализованных на языке C, соответствующих операторам языка Python. Обычно функции имеют такие же имена, как и специальные методы классов, предназначенных для выполнения соответствующих операций. Например, выражения 'operator.add(*x*, *y*)' и '*x* + *y*' эквивалентны. Для удобства также определены имена без '__' в начале и конце имени.

Модуль определяет следующие функции:

add(*x*, *y*)
__add__(*x*, *y*)

Возвращает *x* + *y*.

sub(*x*, *y*)
__sub__(*x*, *y*)

Возвращает *x* - *y*.

mul(*x*, *y*)
__mul__(*x*, *y*)

Возвращает *x* * *y*.

div(*x*, *y*)
__div__(*x*, *y*)

Возвращает *x* / *y*.

mod(*x*, *y*)
__mod__(*x*, *y*)

Возвращает *x* % *y*.

neg(*x*)
__neg__(*x*)

Возвращает -*x*.

pos(*x*, *y*)
__pos__(*x*, *y*)

Возвращает +*x*.

abs(*x*)
__abs__(*x*)
Возвращает абсолютное значение *x* (см. описание встроенной функции abs()).

inv(*x*)
__inv__(*x*)
invert(*x*)
__invert__(*x*)
Возвращает ~*x*. Имена invert() и __invert__() добавлены в версии 2.0.

lshift(*x*, *y*)
__lshift__(*x*, *y*)
Возвращает *x* << *y*.

and_(*x*, *y*)
__and__(*x*, *y*)
Возвращает *x* & *y*.

or_(*x*, *y*)
__or__(*x*, *y*)
Возвращает *x* | *y*.

xor(*x*, *y*)
__xor__(*x*, *y*)
Возвращает *x* ^ *y*.

not_(*x*)
__not__(*x*)
Возвращает not *x*.

truth(*x*)
Возвращает 1, если *x* является истиной, иначе возвращает 0.

concat(*seq1*, *seq2*)
__concat__(*seq1*, *seq2*)
Возвращает *seq1* + seq2, где *seq1* и *seq2* должны быть последовательностями встроенного типа.

repeat(*seq*, *n*)
__repeat__(*seq*, *n*)
Возвращает *seq* * *n*, где *seq* должен быть последовательностью встроенного типа, а *n* — целым числом.

contains(*x*, *y*)
__contains__(*x*, *y*)
sequenceIncludes(*x*, *y*)
Возвращает результат проверки вхождения *y* in *x*. Обратите внимание на порядок следования операндов. Имя __contains__() определено, начиная с версии

2.0, имя `sequenceIncludes()` присутствует лишь для совместимости с предыдущими версиями.

countOf(*x*, *y*)
Возвращает число вхождений *y* в *x*. Эта функция никогда не использует метод `count()` экземпляров-последовательностей, а просто последовательно перебирает и сравнивает с *y* все элементы последовательности *x*.

indexOf(*x*, *y*)
Возвращает индекс первого элемента *x*, равного *y*. Эта функция никогда не использует метод `index()` экземпляров-последовательностей, а просто последовательно перебирает и сравнивает с *y* элементы последовательности *x*. Если *x* не содержит *y*, генерируется исключение `ValueError`.

getitem(*x*, *y*)
__getitem__(*x*, *y*)
Возвращает `x[y]`.

setitem(*x*, *y*, *z*)
__setitem__(*x*, *y*, *z*)
Выполняет присваивание '`x[y] = z`'.

delitem(*x*, *y*)
__delitem__(*x*, *y*)
Эквивалентно инструкции '`del x[y]`'.

getslice(*x*, *y*, *z*)
__getslice__(*x*, *y*, *z*)
Возвращает `x[y:z]`.

setslice(*x*, *y*, *z*, *v*)
__setslice__(*x*, *y*, *z*, *v*)
Выполняет присваивание '`x[y:z] = v`'.

delslice(*x*, *y*, *z*)
__delslice__(*x*, *y*, *z*)
Эквивалентно инструкции '`del x[y:z]`'.

isCallable(*obj*)
Возвращает `1`, если объект *obj* поддерживает вызов, иначе возвращает `0`. Эта функция существует для совместимости с предыдущими версиями — используйте вместо нее встроенную функцию `callable()`.

isMappingType(*obj*)
Возвращает `1`, если объект *obj* может быть отображением, то есть является словарем или экземпляром (произвольного) класса, иначе возвращает `0`. Не существует надежного способа определить, поддерживает ли экземпляр класса все операции, характерные для отображения.

isNumberType(*obj*)
Возвращает 1, если *obj* может быть объектом числового типа, то есть является объектом встроенного числового типа или экземпляром (произвольного) класса, иначе возвращает 0. Не существует надежного способа определить, поддерживает ли экземпляр класса все операции, характерные для чисел.

isSequenceType(*obj*)
Возвращает 1, если *obj* может быть объектом числового типа, то есть является последовательностью одного из встроенных типов или экземпляром (произвольного) класса, иначе возвращает 0. Не существует надежного способа определить, поддерживает ли экземпляр класса все операции, характерные для последовательностей.

Чаще всего определенные здесь функции используются в качестве первого аргумента встроенных функций map() и reduce(). Например, самый быстрый способ подсчитать сумму элементов последовательности *seq* — вычислить выражение 'reduce(operator.add, *seq*)'.

Для удобства ниже приведена таблица соответствия операторов функциям, определенным в модуле operator.

Оператор	Функция
a (точнее not not *a*)	truth(*a*)
not *a*	not_(*a*)
-*a*	neg(*a*)
a + *b*	add(*a*, *b*)
a - *b*	sub(*a*, *b*)
a * *b*	mul(*a*, *b*)
a / *b*	div(*a*, *b*)
a % *b*	mod(*a*, *b*)
~*a*	invert(*a*)
a & *b*	and_(*a*, *b*)
a ^ *b*	xor(*a*, *b*)
a | *b*	or_(*a*, *b*)
a << *b*	lshift(*a*, *b*)
a >> *b*	rshift(*a*, *b*)
seq1 + *seq2*	concat(*seq1*, *seq2*)
seq * *i*	repeat(*seq*, *i*)
o in *seq*	contains(*seq*, *o*)
o[*k*]	getitem(*o*, *k*)
o[*k*] = *v*	setitem(*o*, *k*, *v*)
del *o*[*k*]	delitem(*o*, *k*)
seq[*i*:*j*]	getslice(*seq*, *i*, *j*)
seq[*i*:*j*] = *values*	setslice(*seq*, *i*, *j*, *values*)
del *seq*[*i*:*j*]	delslice(*seq*, *i*, *j*)

15.6 traceback — модуль для работы с объектами traceback

Этот модуль предоставляет средства для извлечения, форматирования и вывода информации, которую несут объекты traceback (см. раздел 11.9.3). Он позволяет полностью имитировать поведение интерпретатора и может быть полезен для вывода сообщений об ошибках без потери контроля над выполнением.

Модуль traceback определяет следующие функции:

print_tb(*traceback* [, *limit* [, *file*]])
Выводит до *limit* уровней информации из объекта *traceback*, начиная с места перехвата исключения, в сторону места, где оно было сгенерировано (именно так описывают путь вложенные друг в друга объекты traceback). Если аргумент *limit* опущен или равен None, выводится информация для sys.tracebacklimit (10000, если переменная sys.tracebacklimit не определена) вложенных объектов traceback. Вся информация выводится в *file* (должен быть объектом с методом write()) или стандартный поток вывода, если аргумент *file* не задан.

print_exception(*type*, *value*, *traceback* [, *limit* [, *file*]])
Работает аналогично функции print_tb(), но помимо вывода информации из объекта *traceback* выполняет следующие действия:

- Если аргумент *traceback* не равен None, выводит заголовок 'Traceback (most recent call last):'.
- Выводит информацию о возникшем исключении (*type* и *value*; должны быть объектами, описывающими тип и значение исключения).
- Если *type* is SyntaxError и *value* имеет соответствующий формат, выводит строку, в которой возникла синтаксическая ошибка с символом '^', указывающим на место обнаружения ошибки интерпретатором.

Именно таким образом выводятся сообщения об ошибке самим интерпретатором.

print_exc([*limit* [, *file*]])
Вызов этой функции эквивалентен вызову 'print_exception(*(sys.exc_info() + (*limit*, *file*)))'.

print_last([*limit* [, *file*]])
Вызов этой функции эквивалентен вызову 'print_exception(sys.last_type, sys.last_value, sys.last_traceback, *limit*, *file*)'.

print_stack([*frame* [, *limit* [, *file*]]])

Отслеживает и выводит путь по стеку от кадра стека *frame* до корня программы (модуля __main__). Если аргумент *frame* опущен или равен None, считается равным кадру стека блока, из которого функция print_stack() была вызвана. Необязательные аргументы *limit* и *file* имеют такое же значение, как и в функции print_exception().

extract_tb(*traceback* [, *limit*])

Возвращает информацию о пути, содержащуюся в объекте *traceback*, в виде списка (каждый элемент представляет кадр стека, соответствующий очередному вложенному объекту traceback) кортежей '(*filename*, *line_number*, *function_name*, *text*)', где *filename* — имя файла, *line_number* — номер строки в файле, *function_name* — имя функции и *text* — строка исходного текста с обрезанными символами пропуска (whitespace) в начале и конце или None. Необязательный аргумент *limit* имеет такое же значение, как и в функции print_exception().

extract_stack([*frame* [, *limit*]])

Возвращает информацию, описывающую путь от указанного или текущего кадра стека до корня программы. Возвращаемое значение имеет такой же формат, как и значение, возвращаемое функцией extract_tb(). Необязательные аргументы *frame* и *limit* имеют такое же значение, как и в функции print_stack().

format_list(*list*)

Возвращает список готовых к выводу (например, методом writelines файлового объекта) строк, полученных форматированием списка кортежей *list* — результата функции extract_tb() или extract_stack(). Каждая строка возвращаемого списка соответствует элементу с таким же индексом аргумента. Каждая строка заканчивается символом перехода на новую строку и может содержать несколько символов перехода на новую строку, если последний элемент соответствующего кортежа (строка исходного текста) не равен None.

format_exception_only(*type*, *value*)

Возвращает список строк, подготовленный для вывода (например, методом writelines файлового объекта) информации об исключении. Каждая строка в возвращаемом списке заканчивается символом перехода на новую строку. *type* и *value* должны быть объектами, представляющими тип и значение исключения. Обычно список содержит только одну строку. Однако для исключений SyntaxError список будет состоять из нескольких строк, содержащих, в том числе, фрагмент исходного текста программы и информацию о месте обнаружения синтаксической ошибки. Строка, описывающая исключение, всегда идет в списке последней.

format_exception(*type*, *value*, *traceback* [, *limit*])

Возвращает список строк, подготовленный для вывода (например, методом writelines файлового объекта) информации об исключении и информации, представляемой объектом *traceback*. Аргументы имеют такое же значение, как и соответствующие аргументы функции print_exception(). Каждая строка в воз-

вращаемом списке заканчивается символом перехода на новую строку и может содержать несколько символов перехода на новую строку. Вывод этих строк (например, с помощью метода `writelines` файлового объекта) дает такой же результат, как и вызов функции `print_exception()`.

format_tb(*traceback* [, *limit*])
Вызов этой функции эквивалентен вызову '`format_list(extract_tb(`*traceback*`,` *limit*`))`'.

format_stack(*frame* [, *limit*])
Вызов этой функции эквивалентен вызову '`format_list(extract_stack(`*frame*`,` *limit*`))`'.

tb_lineno(*traceback*)
Возвращает номер текущей строки, представленной объектом *traceback*. Обычно возвращаемое значение равно *traceback*`.tb_lineno`. Однако при включенной оптимизации атрибут `tb_lineno` не обновляется корректно, в то время как функция `tb_lineno()` всегда возвращает правильное значение.

Следующий пример реализует простейший интерактивный интерпретатор, аналогичный стандартному:

```
import sys, traceback

def run_user_code(envdir):
    source = raw_input(">>> ")
    try:
        exec source in envdir
    except:
        print "Исключение в пользовательском коде:"
        print '-'*60
        traceback.print_exc(file=sys.stdout)
        print '-'*60

envdir = {}
while 1:
    run_user_code(envdir)
```

Для создания более сложных интерактивных интерпретаторов лучше воспользоваться классами, определенными в модуле `code`.

Важное замечание: модуль `traceback` для считывания строк исходного кода использует модуль `linecache`, который держит в памяти содержимое *всех* файлов, к которым когда-либо были обращения. В долгоживущих программах, позволяющим пользователю каким-либо образом (например, путем ввода интерактивных команд) выполнять код из различных файлов, это может привести к нежелательному использованию большого количества памяти. Чтобы этого избежать, следует периодически очищать кэш с помощью функции `linecache.clearcache()`.

15.7 imp — доступ к операциям, производимым инструкцией import

Этот модуль предоставляет интерфейс к операциям, используемым для реализации инструкции import. Он определяет следующие функции:

get_magic()
Возвращает магическое строковое значение, используемое в самом начале байт-компилированных файлов кода для того, чтобы их можно было распознать. Это значение может быть разным для разных версий интерпретатора.

get_suffixes()
Возвращает список кортежей, каждый из которых имеет вид '(*suffix*, *mode*, *type*)' и описывает определенный тип модулей: *suffix* — строка, добавляемая к имени модуля для образования имени файла; *mode* — строка режима, которую необходимо передать встроенной функции open() для того, чтобы открыть файл на чтение ('r' для текстовых файлов и 'rb' для двоичных); *type* — тип файла, может иметь значение PY_SOURCE, PY_COMPILED или C_EXTENSION (см. ниже).

find_module(*name* [, *path*])
Пытается найти модуль с именем *name* в путях *path*. Производится поиск файла с любым из суффиксов, возвращаемых функцией get_suffixes(), по очереди в каждом из каталогов в списке *path*. Неверные имена каталогов молча игнорируются, но все элементы списка должны быть строками. Если аргумент *path* опущен или равен None, поиск производится сначала среди встроенных (C_BUILTIN) и вложенных (PY_FROZEN) модулей, среди ресурсов (PY_RESOURCE) в операционных системах Macintosh или файлов, указанных в системном реестре Windows, затем в каталогах, перечисленных в sys.path.

В случае успеха функция возвращает кортеж вида '(*file*, *pathname*, *description*)', где *file* — файл, открытый для чтения, с указателем, установленным на начало, *pathname* — полное имя файла модуля и *description* — один из кортежей в списке get_suffixes(), описывающий тип модуля. Если модуль не находится в файле, то *file* равен None, *pathname* — имени модуля, а *description* содержит пустые строки (суффикс и режим). Если модуль не найден, генерируется исключение ImportError. Другие исключения указывают на проблемы с аргументами или окружением.

Эта функция не обрабатывает иерархические имена модулей. Для того, чтобы найти *P.M*, то есть модуль *M* пакета *P*, найдите и загрузите модуль *P*, затем используйте функцию find_module() с аргументом *path*, равным *P*.__path__.

load_module(*name*, *file*, *filename*, *description*)
Загружает модуль. Если модуль до этого уже был импортирован, вызов функции load_module() эквивалентен вызову встроенной функции reload(). Аргумент *name* является полным именем модуля (то есть включает имя пакета). Аргументы

file и *filename* — файловый объект, открытый для чтения, и имя соответствующего файла (None и имя модуля, если модуль загружается не из файла). Аргумент *description* должен быть кортежем, описывающим тип модуля. Все необходимые аргументы, кроме имени модуля, возвращаются функцией find_module().

Если модуль успешно загружен, функция возвращает объект-модуль. В противном случае генерируется исключение (обычно ImportError). Заметим, что после загрузки модуля файл *file* (если он не равен None) необходимо закрыть, даже если было сгенерировано исключение. Лучше всего это делать в ветви finally инструкции try.

new_module(*name*)
Возвращает новый пустой объект-модуль с именем *name*. Этот объект *не* добавляется в список sys.modules.

Модуль определяет следующие целочисленные константы, описывающие тип модуля:

PY_SOURCE
Файл с исходным текстом на языке Python.

PY_COMPILED
Байт-компилированный файл.

C_EXTENSION
Динамически загружаемая библиотека.

PY_RESOURCE
Ресурс в операционной системе Macintosh.

PKG_DIRECTORY
Каталог пакета.

C_BUILTIN
Встроенный модуль.

PY_FROZEN
Вложенный модуль (модуль на языке Python, зашитый в интерпретатор).

Для совместимости с ранними версиями интерпретатора модуль также определяет несколько устаревших констант и функций. Хорошие примеры использования описанных здесь возможностей Вы найдете в стандартных модулях knee (этот модуль присутствует только в качестве примера — не следует считать его стандартным интерфейсом) и rexec.

15.8 pprint — представление и вывод данных в более привлекательном виде

Этот модуль предоставляет возможность получить форматированное представление произвольных объектов языка Python. Аналогично встроенной функции `repr()`, возвращаемый результат обычно можно использовать в качестве аргумента встроенной функции `eval()`. Если длина представления вложенных объектов больше разрешенного значения, представление каждого объекта будет расположено на отдельной строке.

Модуль определяет один класс:

PrettyPrinter([*keyword_list*])

Этот конструктор воспринимает несколько именованных аргументов (*keyword_list*). Поток вывода может быть установлен с помощью аргумента с именем `stream` (должен быть объектом с методом `write()`). Если поток вывода не задан, используется `sys.stdout`. Аргументы с именами `indent`, `depth` и `width` определяют формат представления. `indent` определяет отступ (количество пробелов) каждого последующего уровня вложенных объектов, по умолчанию равен `1`. `depth` определяет максимальный уровень вложенных объектов, которые будут представлены — представление объектов на следующем уровне заменяется на '`...`'. Если аргумент с таким именем не задан или его значение ложно, ограничений на количество уровней нет. Длина отдельных строк в представлении будет ограничена значением аргумента с именем `width` (по умолчанию 80 символов).

```
>>> import pprint, sys
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   [   '',
        '/usr/local/lib/python1.5',
        '/usr/local/lib/python1.5/test',
        '/usr/local/lib/python1.5/sunos5',
        '/usr/local/lib/python1.5/sharedmodules',
        '/usr/local/lib/python1.5/tkinter'],
    '',
    '/usr/local/lib/python1.5',
    '/usr/local/lib/python1.5/test',
    '/usr/local/lib/python1.5/sunos5',
    '/usr/local/lib/python1.5/sharedmodules',
    '/usr/local/lib/python1.5/tkinter']
>>>
>>> import parser
>>> suite = parser.suite(open('pprint.py').read())
>>> tup = parser.ast2tuple(suite)[1][1][1]
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
(266, (267, (307, (287, (288, (...))))))
```

Для простоты использования модуль также определяет несколько функций, использующих формат представления по умолчанию:

pformat(*object*)
Возвращает строку с форматированным представлением объекта.

pprint(*object* [, *stream*])
Выводит форматированное представление объекта *object* (с символом новой строки на конце) в файл *stream*. Если аргумент *stream* не задан, используется `sys.stdout`. Эта функция может быть использована в интерактивном режиме для контроля значений.

```
>>> stuff = sys.path[:]
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=869440>,
 '',
 '/usr/local/lib/python1.5',
 '/usr/local/lib/python1.5/test',
 '/usr/local/lib/python1.5/sunos5',
 '/usr/local/lib/python1.5/sharedmodules',
 '/usr/local/lib/python1.5/tkinter']
```

isreadable(*object*)
Возвращает `1`, если представление объекта *object* может быть использовано для его восстановления с помощью функции `eval()`, иначе возвращает `0`. Всегда возвращает `0` для рекурсивных объектов.

```
>>> pprint.isreadable(stuff)
0
```

isrecursive(*object*)
Возвращает `1`, если объект содержит рекурсивные ссылки, иначе возвращает `0`.

saferepr(*object*)
Выводит неформатированное представление объекта *object* с рекурсивными ссылками, представленными в виде '`<Recursion on` *type* `with id=`*id*`>`', где *type* и *id* — тип и идентификатор объекта.

Экземпляры класса `PrettyPrinter` имеют следующие методы, принимающие во внимание значения именованных аргументов, переданных конструктору:

pformat(*object*)
Возвращает стоку с форматированным представлением объекта *object*.

pprint(*object*)
Выводит форматированное представление объекта *object* (с символом новой строки на конце) в файл, указанный при конструировании экземпляра.

isreadable(*object*)
Возвращает `1`, если представление объекта *object* может быть использовано для

его восстановления с помощью функции `eval()`, иначе возвращает `0`. Всегда возвращает `0` для рекурсивных объектов и если объект содержит больше вложенных уровней, чем было разрешено при конструировании экземпляра.

isrecursive(*object*)
Возвращает `1`, если объект содержит рекурсивные ссылки, иначе возвращает `0`.

15.9 `repr` — альтернативная реализация функции `repr()`

Этот модуль позволяет получать представление объектов аналогично встроенной функции `repr()`, но с ограничением размера получаемой строки. Эти возможности используются, например, отладчиком языка Python.

Модуль определяет следующие имена:

Repr()
Класс, предоставляющий возможности, полезные для реализации функций, аналогичных встроенной функции `repr()`. Во избежание генерации чрезмерно больших представлений включает ограничение размеров для различных типов объектов.

aRepr
Экземпляр класса `Repr`. Используется для реализации функции `repr()`, описанной ниже. Изменение атрибутов этого объекта отразится на ограничениях размеров, используемых функцией `repr()` и, соответственно, отладчиком.

repr(*object*)
Ссылается на метод `repr()` экземпляра `aRepr`. Возвращает строковое представление, аналогичное возвращаемому встроенной функцией `repr()`, но с ограничениями по размеру.

Экземпляры класса `Repr` предоставляют несколько атрибутов данных, которые могут быть использованы для установки ограничений размеров представлений объектов различных типов, и методы, возвращающие представления объектов этих типов:

maxlevel
Максимальный уровень вложенности объектов, включаемых в представление. По умолчанию равен `6`.

maxdict
maxlist
maxtuple
Максимальное число элементов, включаемых в представление для словарей, списков и кортежей соответственно. По умолчанию включается представление четырех элементов словарей и шести элементов списков и кортежей.

maxlong
: Максимальное число символов в представлении длинных целых чисел. Цифры опускаются в середине. По умолчанию равно 40.

maxstring
: Максимальное число символов в представлении строки. Заметим, что в качестве исходного функция использует стандартное представление. Если исходное представление содержит управляющие последовательности, в итоговом представлении они могут быть искажены. По умолчанию максимальное число символов равно 30.

maxother
: Максимальное число символов в представлении других типов, по умолчанию равно 20.

repr(*object*)
: Возвращает строковое представление, аналогичное возвращаемому встроенной функцией `repr()`, но с указанными ограничениями по размеру.

repr1(*object*, *level*)
: Этот метод вызывается рекурсивно методом `repr()`. Использует тип объекта *object* для того, чтобы определить, какой метод вызвать для получения представления.

repr_*typename*(*object*, *level*)
: Методы, предназначенные для получения представления объектов определенного типа. Имя метода конструируется из имени типа, в котором пробелы заменены символами подчеркивания, путем добавления приставки `'repr_'`. Методы, предназначенные для получения представления объектов определенного типа, вызываются методом `repr1()` и сами должны вызывать `repr1()` для вложенных объектов с аргументом *level*, меньше на единицу.

Определяя классы, производные от класса `Repr`, Вы можете добавить поддержку для других типов объектов или изменить поведение для поддерживаемых типов. Следующий пример показывает, каким образом можно реализовать поддержку файловых объектов:

```
import repr
import sys

class MyRepr(repr.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>',
                        '<stdout>',
                        '<stderr>']:
            return obj.name
        else:
            return `obj`

aRepr = MyRepr()
print aRepr.repr(sys.stdin)       # выводит '<stdin>'
```

Глава 16

Работа со строками

Модули, описанные в этой главе, предоставляют широкий диапазон операций над строками.

string	Наиболее распространенные операции над строками.
re	Операции с регулярными выражениями.
StringIO	Работа со строками как с файловыми объектами.
cStringIO	Более быстрый вариант модуля `StringIO`.
codecs	Регистрация кодеров и работа с ними.

16.1 `string` — наиболее распространенные операции над строками

Модуль `string` определяет несколько строковых констант, содержащих символы, которые относятся к определенному классу. Их значения зависят от текущих национальных установок (см. описание модуля `locale`).

digits
: Десятичные цифры (`'0123456789'`).

hexdigits
: Символы, используемые для записи шестнадцатеричного представления (`'0123456789abcdefABCDEF'`).

letters
: Объединение строк `lowercase` и `uppercase`, описанных ниже.

lowercase
: Строка, содержащая все строчные буквы (буквы в нижнем регистре).

octdigits
: Символы, используемые для записи восьмеричного представления (`'01234567'`).

punctuation
: Строка символов, которые считаются знаками препинания.

printable
: Строка символов, которые считаются печатными. Является комбинацией констант `digits`, `letters`, `punctuation` и `whitespace`.

uppercase
: Строка, содержащая все прописные буквы (буквы в верхнем регистре).

whitespace
: Строка, содержащая все символы пропуска (whitespace). Обычно это `'\t\n\v\f\r '`

Модуль также определяет следующие функции:

maketrans(*from, to*)
: Возвращает строку, которая может быть использована в методе `translate()` строк для того, чтобы произвести замену символов в строке *from* на символы, находящиеся в той же позиции в строке *to*. Строки *from* и *to* должны иметь одинаковую длину. Не используйте в качестве аргументов строки, производные от констант `lowercase` и `uppercase` — они могут иметь разную длину.

zfill(*s, width*)
: Дополняет слева нулями строку *s*, представляющую число, до тех пор, пока длина строки не будет равной *width*. Строки, начинающиеся знаком, обрабатываются корректно.

Перечисленные ниже функции оставлены для совместимости, их следует использовать только в тех случаях, когда необходимо сохранить возможность выполнение программы со старыми версиями интерпретатора. Начиная с версии 1.6, возможности, предоставляемые этими функциями, доступны через методы обычных строк и строк Unicode или встроенные функции.

atof(*s*)
: Возвращает `float(`*s*`)`.

atoi(*s* [, *base*])
: Возвращает `int(`*s*, *base*`)`.

atol(*s* [, *base*])
: Возвращает `long(`*s*, *base*`)`.

capitalize(*word*)
: Возвращает *word*`.capitalize()`.

capwords(*s* [, *sep*])
: Разбивает строку на слова, используя в качестве разделителя *sep*, делает первую букву каждого слова прописной и возвращает объединение (через разделитель *sep*) полученных слов. Обратите внимание, что серии из символов пропуска при этом в начале и конце строки удаляются, а в середине строки заменяются одним

пробелом. Метод строк title() работает аналогично, но сохраняет исходные символы пропуска.

expandtabs(*s* [, *tabsize*])
Возвращает *s*.expandtabs(*tabsize*).

find(*s*, *sub* [, *start* [, *end*]])
Возвращает *s*.find(*sub*, *start*, *end*).

rfind(*s*, *sub* [, *start* [, *end*]])
Возвращает *s*.rfind(*sub*, *start*, *end*).

index(*s*, *sub* [, *start* [, *end*]])
Возвращает *s*.index(*sub*, *start*, *end*).

rindex(*s*, *sub* [, *start* [, *end*]])
Возвращает *s*.rindex(*sub*, *start*, *end*).

count(*s*, *sub* [, *start* [, *end*]])
Возвращает *s*.count(*sub*, *start*, *end*).

lower(*s*)
Возвращает *s*.lower().

split(*s* [, *sep* [, *maxsplit*]])
splitfields(*s* [, *sep* [, *maxsplit*]])
Возвращает *s*.split(*sep*, *maxsplit*).

join(*words* [, *sep*])
joinfields(*words* [, *sep*])
Возвращает *sep*.join(*words*).

lstrip(*s*)
Возвращает *s*.lstrip().

rstrip(*s*)
Возвращает *s*.rstrip().

strip(*s*)
Возвращает *s*.strip().

swapcase(*s*)
Возвращает *s*.swapcase().

translate(*s*, *table* [, *deletechars*])
Возвращает *s*.translate(*table*, *deletechars*), но работает только с обычными строками.

upper(*s*)
Возвращает *s*.upper().

ljust(*s*, *width*)
Возвращает `s.ljust(width)`.

rjust(*s*, *width*)
Возвращает `s.rjust(width)`.

center(*s*, *width*)
Возвращает `s.center(width)`.

replace(*s*, *old*, *new* [, *maxsplit*])
Возвращает `s.replace(old, new, maxsplit)`.

16.2 re — операции с регулярными выражениями

Этот модуль предоставляет операции с регулярными выражениями, аналогичные тем, которые используются в языке Perl. Обрабатываемые строки (обычные и строки Unicode) могут содержать любые символы, в том числе NUL.

В регулярных выражениях символ обратной косой черты ('\') используется в качестве специального символа, что конфликтует с его использованием с той же целью в строковых литеральных выражениях языка Python. Например, регулярное выражение, соответствующее символу обратной косой черты, должно быть записано как `'\\\\'`, так как само регулярное выражение должно быть '\\' и каждый символ обратной косой черты в литеральном выражении должен быть записан как '\\'. Выход из этой ситуации — использовать для шаблонов необрабатываемую запись строковых литералов (см. раздел 11.2.1).

16.2.1 Синтаксис регулярных выражений

Регулярное выражение (regular expression) — это шаблон, который определяет множество строк, ему удовлетворяющих. Функции, определенные в модуле `re` позволяют определить, удовлетворяет ли определенная строка данному регулярному выражению (или, что то же самое, регулярное выражение строке).

Объединение регулярных выражений также является регулярным выражением. Если строка *a* удовлетворяет регулярному выражению *A* и строка *b* удовлетворяет регулярному выражению *B*, то *a + b* удовлетворяет *A + B*. Таким образом, сложные регулярные выражения могут быть сконструированы из примитивных выражений, описанных ниже.

Регулярные выражения могут содержать как обычные символы, так и специальные управляющие последовательности. Большинство обычных символов, таких как 'A', 'a' или '0', являются простейшими регулярными выражениями, удовлетворяющими самим себе. Вы можете объединять обычные символы, так выражению `'list'` удовлетворяет строка `'list'`.

Ниже приведены символы и последовательности, которые имеют специальное значение (*atom* — “неделимое” выражение, то есть обычный символ или последовательность, обозначающая класс или группу; *expr* — произвольное выражение):

Последовательность	Назначение
. (точка)	В режиме по умолчанию удовлетворяет любому символу, кроме символа новой строки ('\n'). Если был задан флаг DOTALL, удовлетворяет любому символу, включая символ новой строки.
^ (символ вставки)	В режиме по умолчанию удовлетворяет началу строки. Если был задан флаг MULTILINE, также удовлетворяет пустой строке сразу после символа новой строки.
$	В режиме по умолчанию удовлетворяет концу строки. Если был задан флаг MULTILINE, также удовлетворяет пустой строке непосредственно перед символом новой строки.
*atom**	Удовлетворяет наибольшему возможному количеству (0 или более) фрагментов строки, удовлетворяющих выражению *atom*. Например, выражению 'ab*' удовлетворяют строки 'a', 'ab', 'abb' и т. д.
atom+	Удовлетворяет наибольшему возможному количеству (1 или более) фрагментов строки, удовлетворяющих выражению *atom*. Например, выражению 'ab+' удовлетворяют строки 'ab', 'abb' и т. д., но не удовлетворяет строка 'a'.
atom?	Удовлетворяет наибольшему возможному количеству (0 или 1) фрагментов строки, удовлетворяющих выражению *atom*. Например, выражению 'ab?' удовлетворяют строки 'a' и 'ab'.
{*m*,[*n*]}	Удовлетворяет наибольшему возможному количеству (от *m* до *n*) фрагментов строки, удовлетворяющих выражению *atom*. Например, выражению 'a{3,5}' удовлетворяют строки 'aaa', 'aaaa' и 'aaaaa'. Если число *n* опущено, оно считается равным бесконечности.
*atom**?	Удовлетворяет наименьшему возможному количеству (0 или более) фрагментов строки, удовлетворяющих выражению *atom*. Например, регулярное выражение '<.*>' удовлетворяет всей строке '<H1>title</H1>', в то время как '<.*?>' удовлетворяет только '<H1>'.
atom+?	Удовлетворяет наименьшему возможному количеству (1 или более) фрагментов строки, удовлетворяющих выражению *atom*.
atom??	Удовлетворяет наименьшему возможному количеству (0 или 1) фрагментов строки, удовлетворяющих выражению *atom*.

Последовательность	Назначение
`{m,[n]}?`	Удовлетворяет наименьшему возможному количеству (от *m* до *n*) фрагментов строки, удовлетворяющих выражению *atom*.
`[chars]`	Используется для описания множества символов. *chars* может включать в себя символы, диапазоны символов и классы (предопределенные множества символов). Специальные последовательности (кроме тех, которые начинаются с символа обратной косой черты) внутри квадратных скобок не являются активными. Например, выражению `'[ab?]'` удовлетворяют символы `'a'`, `'b'` и `'?'`, выражению `'[a-z]'` — все строчные буквы латинского алфавита, а выражению `'[a-zA-Z0-9]'` — все буквы латинского алфавита и цифры. Если Вы хотите включить в множество символ ‘`]`’ или ‘`-`’, поместите его первыми или используйте символ обратной косой черты: `'[]]'`, `r'[]\-]'`. Символ ‘`^`’ в множестве наоборот, не должен идти первым, должен быть единственным символом в множестве или записанным с использованием обратной косой черты.
`[^chars]`	Удовлетворяет любому символу, не входящему в множество `'[chars]'`.
`expr1|expr2`	Удовлетворяет строкам, удовлетворяющим выражению *expr1* или *expr2*, где *expr1* и *expr2* — произвольные регулярные выражения. Таким образом Вы можете соединить произвольное количество выражений. Будет использовано первое из них, при котором все регулярное удовлетворяет строке.
`(expr)`	Делает выражение *expr* “неделимым” и образует группу. Фрагмент строки, удовлетворяющий группе в данном контексте, может быть извлечен после выполнения операции сопоставления, а также может быть использован далее в этом же регулярном выражении с помощью специальной последовательности `r'\number'`, описанной ниже.
`(?options)`	Такая запись не образует группу и не используется для сопоставления, а лишь устанавливает опции для всего регулярного выражения. *options* может содержать буквы ‘`i`’, ‘`L`’, ‘`m`’, ‘`s`’, ‘`u`’, ‘`t`’ и ‘`x`’, которые соответствуют флагам `IGNORECASE`, `LOCALE`, `MULTILINE`, `DOTALL`, `UNICODE`, `TEMPLATE` и `VERBOSE`. Может быть полезна, если Вы хотите включить флаги в регулярное выражение вместо того, чтобы передавать их функции `compile()`.
`(?:expr)`	Как и запись `'(expr)'` делает выражение *expr* “неделимым”, но в отличие от последней не образует группы.

Последовательность	Назначение
(?P<*name*>*expr*)	Работает аналогично записи '(*expr*)' и, кроме того, делает доступным фрагмент строки, удовлетворяющий выражению *expr*, через имя *name*, то есть делает группу именованной. Имя группы должно быть корректным идентификатором языка Python. Именованные группы так же, как и обычные, нумеруются и доступны через запись r'*number*'.
(?P=*name*)	Удовлетворяет тексту, который ранее удовлетворил выражению, указанному в группе с именем *name*.
(?#*comment*)	Комментарий, игнорируется.
(?=*expr*)	Удовлетворяет *пустой* строке, но только если за ней следует текст, удовлетворяющий выражению *expr*. Например, выражению 'Александр (?=Пушкин)' удовлетворяет фрагмент 'Александр ' в строке 'Александр Пушкин'.
(?!*expr*)	Удовлетворяет *пустой* строке, но только если за ней следует текст, который не удовлетворяет выражению *expr*. Например, выражению 'Александр (?!Пушкин)' удовлетворяет фрагмент 'Александр ' в строке 'Александр Сергеевич Пушкин', но не в строке 'Александр Пушкин'.
(?<=*expr*)	Удовлетворяет *пустой* строке, но только если перед ней следует текст, который удовлетворяет выражению *expr*. Например, выражению '(?<=abc)def' удовлетворяет фрагмент 'def' в строке 'abcdef'. Выражение *expr* должно всегда удовлетворять строке одной длины. Например, Вы можете использовать 'abc' или 'a|b', но не 'a*'. Возможность использования такого синтаксиса присутствует, начиная с версии 2.0.
(?<!*expr*)	Удовлетворяет *пустой* строке, но только если перед ней следует текст, который не удовлетворяет выражению *expr*. Выражение *expr* должно всегда удовлетворять строке одной длины. Например, Вы можете использовать 'abc' или 'a|b', но не 'a*'. Возможность использования такого синтаксиса присутствует, начиная с версии 2.0.
number	Удовлетворяет содержимому группы с номером *number*. Группы нумеруются с 1. Например, выражению r'(.+) \1' удовлетворяют строки 'the the' и '55 55', но не удовлетворяет строка 'the end'. Эта специальная последовательность может быть использована для ссылок на группы с номерами до 99. Если в числе *number* первая цифра 0, *number* содержит три восьмеричные цифры или последовательность содержится в определении множества ('[*chars*]'), такая специальная последовательность будет интерпретироваться как символ с восьмеричным кодом *number*.

Последовательность	Назначение
\A	Удовлетворяет только началу строки.
\b	Удовлетворяет пустой строке в начале или конце слова[1]. Слово определяется как последовательность из букв, цифр и символов подчеркивания. В определении множества ('[*chars*]') последовательность r'\b' для совместимости с литеральными выражениями Python интерпретируется как символ возврата на одну позицию (BS).
\B	Удовлетворяет пустой строке в середине слова.
\d	Удовлетворяет любой десятичной цифре, эквивалентна множеству '[0-9]'. При использовании флага LOCALE или UNICODE удовлетворяет символу, который в данном языке или в базе данных Unicode считается десятичной цифрой.
\D	Удовлетворяет любому символу, не являющемуся десятичной цифрой; эквивалентна множеству '[^0-9]'. При использовании флага LOCALE или UNICODE удовлетворяет символу, который в данном языке или в базе данных Unicode не считается десятичной цифрой.
\s	Удовлетворяет любому символу пропуска (whitespace), эквивалентна множеству r'[\t\n\v\f\r]'. При использовании флага LOCALE или UNICODE удовлетворяет символу, который в данном языке или в базе данных Unicode считается символом пропуска.
\S	Удовлетворяет любому символу, не являющемуся символом пропуска (whitespace), эквивалентна множеству r'[^\t\n\v\f\r]'.При использовании флага LOCALE или UNICODE удовлетворяет символу, который в данном языке или в базе данных Unicode не считается символом пропуска.
\w	Удовлетворяет любому символу, являющемуся буквой, цифрой или символом подчеркивания. Если не был установлен флаг LOCALE, эквивалентна множеству '[a-zA-Z0-9_]'. С установленным флагом LOCALE или UNICODE удовлетворяет символу подчеркивания и символу, который в данном языке или в базе данных Unicode считается буквой или цифрой.
\W	Удовлетворяет любому символу, не входящему в класс r'\w', то есть эквивалентна множеству r'^\w'.
\Z	Удовлетворяет только концу строки.

[1]В текущих реализациях эта специальная последовательность не работает корректно при использовании национальных установок (флаг LOCALE), отличных от 'C'.

Последовательность	Назначение
char	Удовлетворяет символу *char*, где *char* — символ, который, будучи использованным в данном контексте без символа обратной косой черты, обозначает начало какой-либо из описанных выше управляющих последовательностей.

16.2.2 Сопоставление в сравнении с поиском

Модуль re предоставляет две примитивные операции с регулярными выражениями: сопоставление (match) и поиск (search). Если для Вас привычна семантика использования регулярных выражений в языке Perl, поиск — это то, что Вам нужно (см. описания функции search() и одноименного метода объектов, представляющих компилированные регулярные выражения).

Сопоставление дает положительный результат, только если шаблон удовлетворяет строке с самого ее начала или заданной позиции:

```
>>> import re
>>> from operator import truth
>>> truth(re.match('a', 'ba'))
0
>>> truth(re.search('a', 'ba'))
1
```

При поиске специальный символ '^' в многострочном режиме (флаг MULTILINE) помимо начала строки удовлетворяет позиции сразу после символа перехода на новую строку ('\n'), в то время как при сопоставлении '^' независимо от режима удовлетворяет только началу строки:

```
>>> import re
>>> from operator import truth
>>> re_obj = re.compile('.*^a', re.M)
>>> truth(re_obj.match('\na'))
0
>>> truth(re_obj.search('\na'))
1
```

16.2.3 Функции и константы, определенные в модуле

compile(*pattern* [, *flags*])

Компилирует регулярное выражение в строке *pattern* и возвращает представляющий его объект. Поведением регулярного выражения можно управлять с помощью аргумента *flags*. Значение аргумента *flags* может быть составлено из описанных ниже констант с помощью оператора |.

Последовательность инструкций

```
prog = re.compile(pat)
result = prog.match(str)
```

эквивалентна инструкции

```
result = re.match(pat, str)
```

Однако использование `compile()` более эффективно, если регулярное выражение предполагается использовать неоднократно.

IGNORECASE
I
Используется, если необходимо выполнить сопоставление или поиск без учета регистра букв. Использование этого флага совместно с флагом `LOCALE` в версиях интерпретатора 1.5.2 и более ранних бессмысленно, а иногда (если используемая кодировка не является надмножеством ASCII) и опасно.

LOCALE
L
Делает классы `r'\w'`, `r'\W'`, `r'\b'` и `r'\B'` зависящими от текущих национальных установок (используются национальные установки на момент выполнения операции сопоставления или поиска, а не компиляции регулярного выражения).

MULTILINE
M
Устанавливает многострочный режим, в котором специальные символы `'^'` (только при выполнении поиска) и `'$'` (при выполнении как сопоставления, так и поиска) удовлетворяют не только началу и концу строки, но и сразу после и перед символом `'\n'` соответственно.

DOTALL
S
Делает специальный символ `'.'` удовлетворяющим любому символу. Без этого флага `'.'` удовлетворяет любому символу, кроме `'\n'`.

VERBOSE
X
Использование этого флага позволяет писать регулярные выражения в более элегантном виде:

- Символы пропуска (whitespace) в строке-шаблоне игнорируются, за исключением определений множеств и записанных с использованием обратной косой черты.
- Все символы до конца строки после '#', за исключением случаев, когда '#' содержится в определении множества или записан с использованием обратной косой черты (`r'\#'`), считаются комментарием и игнорируются.

TEMPLATE
T
Устанавливает “режим шаблона”: поиск производится без отката назад в случае неудачи во время перебора. Эта опция является экспериментальной (ее поддержка может быть удалена в будущих версиях); доступна, начиная с версии 1.6.

UNICODE
U

Делает классы r'\w', r'\W', r'\b' и r'\B' интернациональными (в соответствии с классами символов в Unicode). Опция доступна, начиная с версии 1.6.

Приведенные ниже функции воспринимают в качестве аргумента *pattern* как строку, так и объект, представляющие компилированное регулярное выражение. Если необходимо установить флаги, используйте компилированное регулярное выражение или специальную последовательность вида '(?*flags*)' в строке.

search(*pattern*, *string* [, *flags*])
Ищет в строке *string* позицию, в которой она удовлетворяет регулярному выражению *pattern* и возвращает объект, представляющий результат сопоставления. Если такая позиция не найдена, возвращает None.

match(*pattern*, *string* [, *flags*])
Если 0 или более символов в начале строки *string* удовлетворяют регулярному выражению *pattern*, возвращает объект, представляющий результат сопоставления, в противном случае возвращает None.

split(*pattern*, *string* [, *maxsplit*])
Разбивает строку *string* в местах, удовлетворяющих регулярному выражению *pattern* и возвращает результат в виде списка. Если *pattern* содержит группирующие скобки, то текст групп также включается в возвращаемый список. Если задан и не равен 0 аргумент *maxsplit*, выполняется не более *maxsplit* расщеплений и остаток строки помещается в конец списка. Например:

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

findall(*pattern*, *string*)
Возвращает список непересекающихся фрагментов строки *string*, удовлетворяющих регулярному выражению *pattern*. Если *pattern* содержит группы, элементами списка будут строки, удовлетворяющие группе, или кортеж строк, если групп в шаблоне несколько. Результаты сопоставления с нулевой длиной также включаются в результат.

sub(*pattern*, *repl*, *string* [, *count*])
Возвращает строку, полученную заменой непересекающихся фрагментов строки *string*, удовлетворяющих регулярному выражению *pattern*, на *repl*. Если не найдено ни одного такого фрагмента, строка возвращается неизменной. *repl* может быть строкой или функцией. Функция должна воспринимать один аргумент — объект, представляющий результат сопоставления, и возвращать строку для замены. Например:

```
>>> def dashrepl(matchobj):
....    if matchobj.group(0) == '-': return ' '
....    else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
```

Если задан и не равен 0 аргумент `count`, он определяет максимальное число производимых замен. По умолчанию число замен не ограничено.

Замена удовлетворяющего регулярному выражению фрагмента нулевой длины производится только в том случае, если он не является смежным с предыдущим таким фрагментом. Таким образом, '`sub('x*', '-', 'abc')`' возвращает `'-a-b-c-'`.

Если *`repl`* является строкой, то обрабатываются содержащиеся в ней специальные последовательности, представляющие символы или ссылки на группы. Например, `r'\n'` заменяется символом новой строки, `r'\6'` (или `r'\g<6>'`) — фрагментом, удовлетворяющим шестой группе, и `r'\g<id>'` — фрагментом, удовлетворяющим группе с именем `id`.

subn(*pattern*, *repl*, *string* [, *count*])
Выполняет те же самые операции, что и функция `sub()`, но возвращает кортеж, состоящий из новой строки и числа произведенных замен.

escape(*string*)
Возвращает регулярное выражение, пригодное для поиска вхождений строки *string*, то есть записывает с использованием обратной косой черты символы строки, имеющие специальное значение в регулярных выражения.

error
Класс исключений, которые генерируются, если используемое регулярное выражение содержит ошибки (например, несоответствие группирующих скобок), а также при возникновении некоторых других ошибок. Несоответствие строки шаблону никогда не является ошибкой.

16.2.4 Объекты, представляющие регулярные выражения

Компилированные объекты регулярных выражений имеют следующие методы и атрибуты данных:

search(*string* [, *pos* [, *endpos*]])
Ищет в строке *string* позицию, в которой она удовлетворяет регулярному выражению и возвращает объект, представляющий результат сопоставления. Если такая позиция не найдена, возвращает `None`. Необязательные аргументы *pos* и *endpos* имеют такое же значение, как и для метода `match()`, описанного ниже.

match(*string* [, *pos* [, *endpos*]])
Если 0 или более символов в начале строки *string* удовлетворяют регулярному выражению, возвращает объект, представляющий результат сопоставления, в противном случае возвращает `None`.

Необязательный аргумент *pos* (по умолчанию равен 0) указывает позицию в строке, начиная с которой необходимо выполнять сопоставление. Использование этого аргумента *не* эквивалентно вызову метода для *string*[*pos*:], так как в первом случае специальные последовательности '^' и r'\A' удовлетворяют началу *реальной* строки. Аналогично аргумент *endpos* указывает позицию в строке, до которой будет выполняться сопоставление.

split(*string* [, *maxsplit*])
Вызов этого метода эквивалентен вызову функции split() с объектом, представляющим регулярное выражение, в качестве первого аргумента.

findall(*string*)
Вызов этого метода эквивалентен вызову функции findall() с объектом, представляющим регулярное выражение, в качестве первого аргумента.

sub(*repl*, *string* [, *count*])
Вызов этого метода эквивалентен вызову функции sub() с объектом, представляющим регулярное выражение, в качестве первого аргумента.

subn(*repl*, *string* [, *count*])
Вызов этого метода эквивалентен вызову функции subn() с объектом, представляющим регулярное выражение, в качестве первого аргумента.

flags
Значение аргумента *flags*, переданного функции compile() при компиляции объекта. Этот атрибут не отражает флаги, установленные с помощью специальных последовательностей в самом регулярном выражении.

groupindex
Словарь, отображающий имена групп к их номерам (пустой, если регулярное выражение не содержит именованных групп).

pattern
Строка, содержащая регулярное выражение, которое было использовано при компиляции объекта.

16.2.5 Объекты, представляющие результат сопоставления

Объекты, возвращаемые методами search() и match() компилированных регулярных выражений, а также одноименными функциями, имеют следующие методы и атрибуты данных:

group([*group1* [, *group2* ...]])
Возвращает один или более фрагментов, которые при выполнении операции сопоставления удовлетворили определениям одной или нескольких групп. Числа *groupN* указывают на группы с номерами от 1 до 99 или на фрагмент, удовлетворяющий всему регулярному выражению (группа с номером 0). *groupN* может

быть также именем группы, заданной в регулярном выражении с помощью синтаксиса `'?P<name>expr'`. Если шаблон не содержит группы с указанным номером (именем), генерируется исключение `IndexError`.

С одним аргументом метод `group()` возвращает одну строку, с несколькими аргументами — кортеж из строк (по строке для каждого указанного номера группы), без аргументов — строку, удовлетворяющую всему регулярному выражению (как если бы метод был вызван с одним аргументом, равным `0`). Если регулярное выражение какой-либо группы не было удовлетворено фрагментом строки, возвращаемое значение для этой группы будет `None` вместо строки. Если же оно было удовлетворено несколько раз, в возвращаемое значение включается только последний результат.

Приведем небольшой пример. После выполнения инструкции

```
m = re.match(r"(?P<int>\d+)\.(\d*)", '3.14')
```

`m.group(1)` и `m.group('int')` возвращают `'3'`, а `m.group(2)` возвращает `'14'`.

groups([*default*])
Возвращает кортеж из фрагментов, которые при выполнении операции сопоставления удовлетворили определениям всех групп, содержащихся в регулярном выражении. *default* используются в качестве значения по умолчанию для групп, которые не удовлетворяют никакому фрагменту. Если аргумент *default* не задан, в таких случаях всегда используется `None`.

groupdict([*default*])
Возвращает словарь, отображающий имена групп к фрагментам, которым эти группы удовлетворяют. *default* используется в качестве значения по умолчанию для групп, которые не удовлетворяют никакому фрагменту. Если аргумент *default* не задан, в таких случаях всегда используется `None`.

start([*group*])
end([*group*])
Возвращает позиции начала и конца фрагмента в исходной строке, который удовлетворяет группе с номером (или именем) *group* (по умолчанию используется `0`, что соответствует всему регулярному выражению). Если группа не удовлетворяют никакому фрагменту, возвращают `-1`. Фрагмент, удовлетворяющий группе *g*, то есть *m*`.group(`*g*`)`, может быть также получен как *m*`.string[`*m*`.start(`*g*`):`*m*`.end(`*g*`)]`.

span([*group*])
Возвращает кортеж '`(`*m*`.start(`*group*`), `*m*`.end(`*group*`))`', где *m* — объект, к которому применяется метод. Обратите внимание, что, если группа не удовлетворяют никакому фрагменту, метод возвращает '`(-1, -1)`'.

expand(*template*)
В шаблоне *template* заменяет специальные последовательности, ссылающиеся на символы (например, '`\n`') и удовлетворяющие фрагменты объекта ('`\1`', '`\g<1>`',

'\g<*name*>') аналогично методу sub() и возвращает результат. Этот метод доступен, начиная с версии 2.0.

pos

Значение аргумента *pos*, переданного функции (методу) search() или match().

endpos

Значение аргумента *endpos*, переданного функции (методу) search() или match().

lastgroup

Имя последней удовлетворенной группы (наиболее охватывающей) или None, если эта группа не имеет имени или не была удовлетворена ни одна группа. Например:

```
>>> import re
>>> re.match('(?P<g1>a)(?P<g2>b(?P<g3>c))',
...          'abc').lastgroup
'g2'
```

lastindex

Номер последней удовлетворенной группы (наиболее охватывающей) или None, если не была удовлетворена ни одна группа. Например:

```
>>> import re
>>> re.match('(?P<g1>a)(?P<g2>b(?P<g3>c))',
...          'abc').lastindex
2
```

re

Объект компилированного регулярного выражения, который использовался для выполнения операции поиска или сопоставления.

string

Значение аргумента *string*, переданного функции (методу) search() или match().

16.3 StringIO и cStringIO — работа со строками как с файловыми объектами

Модуль StringIO реализует класс StringIO, экземпляры которого ведут себя аналогично файловым объектам, но работает не с реальным файлом, а со строковым буфером. Модуль cStringIO, реализованный на C (и поэтому более быстрый), предоставляет аналогичный интерфейс, за исключением того, что вместо класса он определяет функцию-конструктор StringIO(). Объекты, возвращаемые этой функцией, ведут себя аналогично экземплярам класса StringIO.

Экземпляры класса `StringIO` поддерживают как обычные строки, так и строки Unicode. Однако следует быть осторожным при использовании одновременно обоих типов — если обычная строка содержит символы с установленным восьмым битом, методы `write()` и `close()` могут сгенерировать исключение `UnicodeError`. Объекты, созданные конструктором `cStringIO.StringIO()` поддерживают только обычные строки: строки Unicode, передаваемые методу `write()`, автоматически преобразуются и, если это невозможно, генерируется исключение `UnicodeError`.

StringIO([*string*])

Создает и возвращает объект, представляющий строковой буфер. Если задан аргумент *string*, он используется в качестве начального содержимого буфера, в противном случае создается пустой объект. Указатель у создаваемого объекта всегда устанавливается на начало буфера.

Конструктор `StringIO()`, определенный в модуле `cStringIO` возвращает объекты разного типа в зависимости от наличия аргумента. В случае, если конструктор вызван с аргументом (должен быть обычной строкой, иначе будет сгенерировано исключение `ValueError`), Вы можете только читать информацию из созданного строкового буфера. При вызове без аргументов возвращаемый объект допускает как чтение, так и запись.

InputType (в модуле `cStringIO`)

Тип объекта, созданного вызовом функции `cStringIO.StringIO()` со строкой в качестве аргумента.

OutputType (в модуле `cStringIO`)

Тип объекта, созданного вызовом функции `cStringIO.StringIO()` без аргументов.

Объекты, возвращаемые конструктором `StringIO()` поддерживают операции, характерные для файловых объектов, а также имеют дополнительный метод:

getvalue()

Возвращает содержимое буфера в виде строки. Этот метод может быть вызван в любое время, до того как буфер был закрыт с помощью метода `close()`.

16.4 codecs — регистрация кодеров и работа с ними

Доступен, начиная с версии 1.6.

Этот модуль определяет базовые классы для кодеров и предоставляет доступ к внутреннему реестру кодеров.

register(*search_function*)

Регистрирует *search_function* как функцию для поиска кодеров. Функция *search_function* вызывается с одним аргументом — именем кодировки, записанной строчными буквами, и должна возвращать кортеж объектов, поддерживающих вызов, '(*encoder*, *decoder*, *stream_reader*, *stream_writer*)'. *encoder* и *decoder* должны иметь интерфейс, аналогичный методам encode() и decode() экземпляров класса Codec, описанного ниже. *stream_reader* и *stream_writer* должны быть конструкторами, возвращающими объекты аналогичные экземплярам классов StreamReader и StreamWriter соответственно. Если *search_function* не может найти объекты для данной кодировки, она должна вернуть None.

lookup(*encoding*)

Ищет в реестре кодеров средства работы с кодировкой *encoding* и возвращает кортеж объектов '(*encoder*, *decoder*, *stream_reader*, *stream_writer*)', описанный выше. Если объекты для данной кодировки не найдены, генерируется исключение LookupError.

Для упрощения работы с файлами и другими потоками, содержащими данные в определенной кодировке, модуль определяет следующие функции:

open(*filename* [, *mode* [, *encoding* [, *errors* [, *buffering*]]]])

Открывает файл с именем *filename* в кодировке *encoding* и возвращает файловый объект (экземпляр класса StreamReaderWriter, описанного ниже), обеспечивающий прозрачное кодирование/декодирование данных при записи/чтении. Аргумент *mode* указывает режим, в котором открывается файл (по умолчанию используется 'rb'). Во избежание потери данных файл всегда открывается в двоичном режиме. Аргумент *errors* определяет поведение при ошибках и может иметь значения 'strict' (в случае возникновения ошибки генерируется исключение ValueError или производного от него класса; используется по умолчанию), 'ignore' (ошибки игнорируются) или 'replace' (используется подходящий символ замены, обычно '\uFFFD'). Аргумент *buffering* имеет такое же значение, как и для встроенной функции open().

EncodedFile(*file*, *input* [, *output*, [*errors*]])

Возвращает файловый объект (экземпляр класса StreamRecoder, описанного ниже), обеспечивающий прозрачное перекодирование при работе с потоком *file* (файловый объект). Данные в файле хранятся в кодировке *output*, в то время как строки передаваемые методам write() и writelines() и возвращаемые методами read(), readline() и readlines() содержат данные в кодировке *input*. В качестве промежуточной кодировки обычно используется Unicode. Если кодировка *output* не задана, она считается равной *input*. Аргумент *errors* имеет такое же значение, как и для функции open().

Модуль определяет следующие классы, которые могут быть использованы в качестве базовых:

Codec()
Базовый класс для кодеров/декодеров. Производные от него классы должны реализовать следующие методы:

encode(*input* [, *errors*])
Должен возвращать кортеж из строки, полученной при кодировании *input*, и количества обработанных символов.

decode(*input* [, *errors*])
Должен возвращать кортеж из строки, полученной при раскодировании *input*, и количества обработанных символов.

StreamWriter(*stream* [, *errors*])
Базовый класс для файловых объектов, обеспечивающих прозрачное кодирование данных при записи его в поток *stream*. Этот класс является производным от класса `Codec` и использует для кодирования свой метод `encode()`, который должен быть переопределен в производном классе.

StreamReader(*stream* [, *errors*])
Базовый класс для файловых объектов, обеспечивающих прозрачное декодирование данных при чтении их из потока *stream*. Этот класс является производным от класса `Codec` и использует для декодирования свой метод `decode()`, который должен быть переопределен в производном классе.

Для упрощения реализации методов `encode()` и `decode()` классов, производных от `Codec`, могут быть полезны следующие функции:

charmap_encode(*input*, *errors*, *encoding_map*)
Кодирует строку *input*, используя отображение кодов символов *encoding_map*. Возвращает результат в том виде, в котором должен возвращать метод `encode()`.

charmap_decode(*input*, *errors*, *encoding_map*)
Декодирует строку *input*, используя отображение кодов символов *encoding_map*. Возвращает результат в том виде, в котором должен возвращать метод `decode()`.

Хорошими примерами использования описанных выше классов могут служить исходные тексты модулей пакета `encodings`.

Экземпляры приведенных ниже классов возвращаются функциями `open()` и `EncodedFile()`:

StreamReaderWriter(*stream*, *Reader*, *Writer* [, *errors*])
Создает файловый объект, обеспечивающий доступ на чтение с кодированием (через объект *Reader*(*stream*, *errors*)) и запись с декодированием (через объект *Writer*(*stream*, *errors*)) к потоку *stream*.

StreamRecoder(*stream*, *encode*, *decode*, *Reader*, *Writer* [, *errors*])

Экземпляры этого класса обеспечивают прозрачное перекодирование при работе с потоком *file* (файловый объект). Данные в файле хранятся в кодировке *output*, в то время как строки передаваемые методам write() и writelines() и возвращаемые методами read(), readline() и readlines() содержат данные в кодировке *input*.

Следующие константы могут быть полезны при чтении и записи файлов, формат которых зависит от платформы:

BOM
BOM_BE
BOM_LE
BOM32_BE
BOM32_LE
BOM64_BE
BOM64_LE

Эти константы определяют метки, которые используются для обозначения порядка следования байтов (byte order mark, BOM). Константа BOM равна BOM_BE или BOM_LE и определяет "родной" порядок следования для данной платформы. Суффиксы '_BE' и '_LE' обозначают соответственно big-endian и little-endian.

Глава 17

Средства интернационализации

Средства, описанные в этой главе, позволяют писать программы, учитывающие национальные установки пользователя.

locale	Использование национальных особенностей.
gettext	Выдача сообщений на родном языке.

17.1 locale — использование национальных особенностей

Модуль locale позволяет работать с различными языками, не зная национальных и культурных особенностей каждой страны. Изначально система использует стандартные национальные установки (с именем 'C').

Error

Исключения этого класса генерируются функцией setlocale(), если установки не могут быть изменены.

setlocale(*category* [, *locale*])

Если задан аргумент *locale*, изменяет текущие национальные установки категории *category*. Аргумент *locale* является именем, характеризующим национальную принадлежность (locale). Если аргумент *locale* равен пустой строке, используются пользовательские установки по умолчанию. Начиная с версии 2.0, Вы можете использовать в качестве *locale* кортеж из двух строк: кода языка и кодировки. В случае успешного изменения установок, возвращает новое имя, в противном случае генерирует исключение Error.

Если аргумент *locale* опущен или равен None, возвращает текущие установки для категории *category*.

На большинстве платформ использование функции setlocale() небезопасно в многопоточных программах. Обычно приложения начинаются со следующих строк:

```
import locale
locale.setlocale(locale.LC_ALL,'')
```

Это позволяет использовать пользовательские установки по умолчанию для всех

категорий[1]. Если национальные установки после этого не меняются, у Вас не должно возникнуть проблем с использованием нескольких потоков.

localeconv()

Возвращает словарь, характеризующий некоторые национальные соглашения. В качестве ключей могут быть использованы следующие строки:

'decimal_point' (категория LC_NUMERIC)
: Символ, используемый в качестве десятичной точки.

'grouping' (категория LC_NUMERIC)
: Список чисел, указывающих, в каких относительных позициях ожидается использование группирующего разделителя localeconv()['thousands_sep']. Если последовательность заканчивается символом CHAR_MAX, разделитель далее не используется. Если последовательность заканчивается нулем, далее повторно используется последний размер группы .

'thousands_sep' (категория LC_NUMERIC)
: Символ, используемый в качестве разделителя между группами.

'int_curr_symbol' (категория LC_MONETARY)
: Международное обозначение национальной валюты (в соответствии со стандартом ISO 4217).

'currency_symbol' (категория LC_MONETARY)
: Местное обозначение национальной валюты.

'mon_decimal_point' (категория LC_MONETARY)
: Символ, используемый в качестве десятичной точки в денежных суммах.

'mon_grouping' (категория LC_MONETARY)
: Имеет такой же формат, как и localeconv()['grouping']. Используется для денежных сумм.

'mon_thousands_sep' (категория LC_MONETARY)
: Символ, используемый в качестве разделителя между группами в денежных суммах.

'positive_sign' (категория LC_MONETARY)
'negative_sign' (категория LC_MONETARY)
: Символы, используемые для обозначения положительных и отрицательных денежных сумм.

'int_frac_digits' (категория LC_MONETARY)
'rac_digits' (категория LC_MONETARY)
: Число цифр после десятичной точки для международных и местных сумм.

'p_cs_precedes' (категория LC_MONETARY)
'n_cs_precedes' (категория LC_MONETARY)
: 0, если символ валюты указывается после суммы, и 1, если символ валюты

---

[1] В ОС Windows используются разные кодировки для консольных и графических приложений, для русского языка это cp866 и Unicode. Однако установки по умолчанию подразумевают кодировку, которая использовалась в старых версиях Windows в графическом режиме — cp1251. Поэтому использование национальных установок по умолчанию в Windows, к сожалению, не имеет смысла.

указывается перед суммой, для положительных и отрицательных денежных сумм соответственно.

'p_sep_by_space' (категория LC_MONETARY)

'n_sep_by_space' (категория LC_MONETARY)

1, если сумма и символ валюты должны быть разделены пробелом, иначе 0, для положительных и отрицательных денежных сумм соответственно.

'p_sign_posn' (категория LC_MONETARY)

'n_sign_posn' (категория LC_MONETARY)

Указывает, в каком формате должна быть выведена денежная сумма со знаком (соответственно положительным и отрицательным): 0 — сумма и символ валюты заключаются в скобки, 1 — знак указывается перед суммой и символом валюты, 2 — после суммы и символа валюты, 3 — непосредственно перед символом валюты, 4 — непосредственно после символа валюты и CHAR_MAX — формат не установлен.

CHAR_MAX

Символическая константа, обычно используется в словаре, возвращаемом функцией localeconv(), если формат не установлен.

getdefaultlocale([*envvars*])

Пытается определить пользовательские национальные установки по умолчанию для данной категории и возвращает их в виде кортежа из строк с кодом языка и кодировкой (вместо строки может быть None, если значение не может быть определено)[2]. Если установки по умолчанию не могут быть определены с помощью системного вызова, используются переменные окружения с именами, приведенными в списке *envvars* (по умолчанию используется ‘['LANGUAGE', 'LC_ALL', code'LC_CTYPE', 'LANG']’). Функция доступна, начиная с версии 2.0.

getlocale([*category*])

Возвращает текущие установки для категории *category* в виде кортежа из строк с кодом языка и кодировкой (вместо строки может быть None, если значение не может быть определено). Аргумент *category* может быть любой константой вида LC_*, кроме LC_ALL; по умолчанию используется LC_CTYPE. Функция доступна, начиная с версии 2.0.

normalize(*locale*)

Возвращает нормализованное имя национальных установок. Если имя *locale* не может быть нормализовано, оно возвращается без изменений. Если в исходном имени не указана кодировка, используется кодировка по умолчанию для данного языка (ISO8859-5 для русского). Функция доступна, начиная с версии 2.0.

resetlocale([*category*])

Изменяет текущие национальные установки для категории *category* (по умолчанию LC_ALL) на пользовательские установки по умолчанию[3]. Функция доступна,

[2] В Windows NT getdefaultlocale() возвращает верное значение, которое, однако, не может быть использовано в качестве аргумента функции setlocale(), так как последняя требует имя национальных установок в своем (не стандартном) виде. Ситуация возможно изменится в будущих версиях языка.

[3] Не работает в Windows NT из-за несовместимости функций getdefaultlocale() и setlocale() (см. также сноску 2).

начиная с версии 2.0.

strcoll(*string1*, *string2*) (категория LC_COLLATE)
Производит лексикографическое сравнение строк *string1* и *string2* в соответствии с текущими национальными установками. Как и другие функции сравнения возвращает положительное, отрицательное число или 0 в зависимости от того, располагается строка *string1* до или после *string2* или строки равны. Строки не должны содержать символов '\000'.

strxfrm(*string*) (категория LC_COLLATE)
Преобразует строку таким образом, чтобы можно было проводить более быстрое сравнение с помощью встроенной функции cmp() (или операторов сравнения) и результат был таким же, какой возвращает функция strcoll(). Строка не должна содержать символов '\000'.

format(*format*, *val* [, *grouping*]) (категория LC_NUMERIC)
Возвращает строковое представление числа с учетом текущих национальных установок. Строка формата следует соглашениям, принятым для оператора %. Функция принимает во внимание местный символ десятичной точки и, если задан и является истиной аргумент *grouping*, группирует соответствующим образом цифры.

str(*float*) (категория LC_NUMERIC)
Возвращает строковое представление числа аналогично встроенной функции str(), но использует символ десятичной точки в соответствии с текущими национальными установками.

atof(*string*) (категория LC_NUMERIC)
Преобразует строку, содержащую представление вещественного числа в соответствии с текущими национальными установками, в вещественное число и возвращает его.

atoi(*string*) (категория LC_NUMERIC)
Преобразует строку, содержащую представление целого числа в соответствии с текущими национальными установками, в целое число и возвращает его.

Ниже перечислены константы, которые могут использоваться в качестве первого аргумента функции setlocale() (категории национальных установок):

LC_CTYPE
От установок этой категории зависит поведение методов обычных строк и функций модуля string, предназначенных для анализа и изменения регистра букв.

LC_COLLATE
От установок этой категории зависит поведение функций strcoll() и strxfrm().

LC_TIME
От установок этой категории зависит поведение функции time.strftime().

LC_MONETARY

Категория для формата денежных сумм (см. описание функции `localeconv()` выше).

LC_MESSAGES (UNIX)

Эта категория обычно определяет язык выводимых сообщений. Однако в настоящих реализациях интерпретатора установки этой категории не используются (см. описание модуля `gettext`).

LC_NUMERIC

Категория, определяющая формат чисел. Влияет на поведение функций `format()`, `atoi()`, `atof()` и `str()`, определенных в этом модуле.

LC_ALL

Комбинация всех категорий. При использовании этой константы для изменения текущих установок функция `setlocale()` пытается изменить установки для всех категорий. Если изменение хотя бы для одной категории невозможно, установки для всех категорий остаются прежними. Если Вы используете `LC_ALL` для определения текущих установок, возвращается строка, указывающая установки для каждой категории. Эта строка может быть использована в дальнейшем для восстановления установок.

17.2 `gettext` — выдача сообщений на родном языке

Модуль `gettext` предоставляет возможность модулям и приложениям выдавать сообщения на родном языке пользователя. Этот модуль поддерживает как интерфейс к каталогу сообщений GNU gettext, так и основанный на классах интерфейс более высокого уровня, который больше подходит для кода на языке Python. В обоих случаях Вы пишите модуль или приложение с сообщениями на одном естественном языке и предоставляете каталог переведенных сообщений на другие языки.

Замечание: в текущих реализациях язык, на который осуществляется перевод, не зависит от национальных установок, выбранных с помощью функции `locale.setlocale()`, а выбирается в зависимости от значений переменных окружения LANGUAGE, LC_ALL, LC_MESSAGES и LANG (первая из них, имеющая непустое значение) или указывается явно. Далее мы будем обозначать строку с кодом выбранного языка как *language*.

17.2.1 Интерфейс GNU gettext

Используя интерфейс GNU gettext, Вы затрагиваете все приложение. Обычно это то, что нужно для одноязычных приложений. Если же Вы пишите модуль или Вашему приложению необходимо менять язык “на лету”, Вам следует использовать интерфейс основанный на классах.

bindtextdomain(*domain* [, *localedir*])
Привязывает домен с именем *domain* к каталогу *localedir*. В дальнейшем модуль gettext будет искать двоичный файл с переведенными сообщениями с путем os.path.join(*localedir*, *language*, 'LC_MESSAGES', *domain* +'.mo'). Если аргумент *localedir* опущен или равен None, возвращает каталог, к которому на данный момент привязан домен *domain* (по умолчанию используется каталог 'os.path.join(sys.prefix, 'share', 'locale')').

textdomain([*domain*])
Если задан и не равен None аргумент *domain*, возвращает текущий домен, иначе устанавливает глобальный (для всего приложения) домен равным *domain* и возвращает его.

gettext(*message*)
Возвращает перевод сообщения *message* на язык, соответствующий текущим национальным установкам. Поиск перевода производится в базе данных сообщений для текущего домена. Обычно эта функция используется под именем _() в глобальном пространстве имен приложения (см. пример ниже).

dgettext(*domain*, *message*)
Работает аналогично функции gettext(), но ищет перевод в базе данных для домена *domain*.

GNU gettext также определяет функцию dcgettext(), однако она считается бесполезной и поэтому на данный момент не реализована.

Приведем пример типичного использования описанного интерфейса:

```
import gettext
gettext.textdomain('myapplication')
_ = gettext.gettext
...
# Вывод строки, подлежащей переводу:
print _('This is a translatable string.')
...
```

17.2.2 Интерфейс, основанный на классах

Этот интерфейс предоставляет большие гибкость и удобства, чем интерфейс GNU gettext. Поэтому мы рекомендуем его для использования в модулях и приложениях, написанных на языке Python. Модуль gettext определяет “класс-переводчик”, который реализует чтение '.mo'-файлов и предоставляет методы, возвращающие обычные строки или строки Unicode с переводом. Экземпляры этого класса могут установить функцию _() во встроенном пространстве имен.

find(*domain* [, *localedir* [, *languages*]])
Эта функция реализует стандартный алгоритм поиска '.mo'-файлов для домена *domain* в каталоге *localedir* для языков, перечисленных в строке *languages* (по умолчанию используется значение одной из переменных окружения: LANGUAGE, LC_ALL, LC_MESSAGES или LANG). Строка *languages* (или одна из перечисленных переменных окружения) должна содержать список через двоеточие кодов языков в порядке убывания их приоритета. Функция find() возвращает путь к '.mo'-файлу, из которого следует брать перевод сообщения. Если ни один из файлов не найден, возвращает None.

translation(*domain* [, *localedir* [, *languages* [, *class_*]]])
Catalog(*domain* [, *localedir* [, *languages* [, *class_*]]])
Возвращает объект, реализующий перевод сообщений. Аргументы *domain*, *localedir* и *languages* передаются функции find() для поиска нужного '.mo'-файла. Возвращаемый объект является экземпляром класса *class_* (по умолчанию используется GNUTranslations). Конструктор класса вызывается с одним аргументом — файловым объектом, реализующим доступ к '.mo'-файлу. Если '.mo'-файл не найден, генерируется исключение IOError. Функция translation() кэширует созданные объекты и использует их при повторных запросах объекта для того же '.mo'-файла.

Имя Catalog() определено для совместимости с версией модуля gettext, которую использует GNOME.

install(*domain* [, *localedir* [, *unicode*]])
Эквивалентно вызову 'translation(*domain*, *localedir*).install(*unicode*)'. Эта функция предоставляет самый простой способ установить функцию _(), реализующую перевод сообщений. Для удобства _() устанавливается в пространство встроенных имен и, таким образом, доступна во всех модулях, используемых приложением.

Ниже описаны классы, непосредственно реализующие перевод строк сообщений.

NullTranslations([*fp*])
Этот класс реализует основной интерфейс, и используется в качестве базового класса для остальных классов-переводчиков. При инициализации устанавливает "защищенные" атрибуты *_info* и *_charset* созданного экземпляра равными None и, если задан и не равен None аргумент *fp*, вызывает метод _parse(*fp*), который должен быть переопределен в производных классах.

Экземпляры NullTranslations имеют следующие методы:

_parse(*fp*)
Ничего не делает в базовом классе. В производных классах этот метод должен считывать данные из файла, представленного файловым объектом *fp*, и инициализировать каталог сообщений экземпляра.

gettext(*message*)
В базовом классе возвращает сообщение *message* без изменений. В производных классах должен возвращать перевод сообщения.

ugettext(*message*)
В базовом классе возвращает сообщение *message* в виде строки Unicode. В производных классах должен возвращать перевод сообщения в виде строки Unicode.

info()
Возвращает значение "защищенного" атрибута `_info` (метаданные каталога в виде словаря).

charset()
Возвращает значение "защищенного" атрибута `_charset` (кодировка перевода сообщений в каталоге).

install([*unicode*])
Если аргумент *unicode* опущен или является ложью, устанавливает собственный метод `gettext()` в пространство встроенных имен под именем `_()`, в противном случае устанавливает собственный метод `ugettext()`.

Заметим, что функция в пространстве встроенных имен становится доступной во всех модулях приложения. Сами модули никогда не должны использовать это метод. Чтобы обеспечить корректную работу модуля в тех случаях, когда импортирующее его приложение не устанавливает функцию `_()` в пространство встроенных имен, следует в начале модуля поместить примерно следующий код:

```
def gettext(message): return message

import __builtin__
if hasattr(__builtin__, 'ps1'):
    # Используется интерактивный режим
    _ = gettext
elif not hasattr(__builtin__, '_'):
    __builtin__._ = gettext
```

Это также позволит обеспечить корректную работу (без перевода сообщений) при импортировании модуля в интерактивном режиме.

Модуль `gettext` также определяет класс `GNUTranslations`, производный от `NullTranslations`. Этот класс переопределяет метод `_parse()`, обеспечивая чтение '.mo'-файлов в формате GNU gettext (как big-endian, так и little-endian). Он также анализирует метаданные каталога и сохраняет их в "защищенном" атрибуте экземпляра `_info`. Если каталог содержит запись с ключом `'Content-Type'`, то указанная в ней кодировка сохраняется в "защищенном" атрибуте `_charset`.

Если '.mo' оказывается поврежденным, при инициализации экземпляра `GNUTranslations` будет сгенерировано исключение `IOError`.

Класс `GNUTranslations` также переопределяет методы `gettext()` и `ugettext()`. Последний передает атрибут `_charset` встроенной функции `unicode()` для получения строки Unicode.

ОС Solaris определяет собственный формат '.mo'-файлов, который в настоящее время не поддерживается.

17.2.3 Изготовление каталога переведенных сообщений

Для того, чтобы подготовить код на языке Python к интернационализации, необходимо взглянуть на строки в Вашем файле. Все строки, подлежащие переводу, должны быть записаны литерально и использоваться в качестве аргумента функции `_()`. Например:

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

Здесь строка `'writing a log message'` помечена как подлежащая переводу, в то время как `'mylog.txt'` и `'w'` — нет.

В стандартную поставку Python включена утилита **pygettext**, которая извлекает строки, подлежащие переводу, из файла с исходным кодом и помещает их в '.pot'-файл[4]. Именно с этим файлом работает человек, выполняющий перевод. С помощью утилит пакета GNU gettext Вы можете изготовить '.mo'-файл, который читается экземплярами класса `GNUTranslations`.

[4]Аналогичную работу выполняет программа **xpot**, доступная в пакете **po-utils** по адресу http://www.iro.umontreal.ca/contrib/po-utils/HTML.

Глава 18

Математический аппарат

Модули, описанные в этой главе, позволяют выполнять основные математические операции. Если Вас интересуют многомерные массивы и функции линейной алгебры, взгляните на пакет модулей “Numerical Python” (NumPy), доступный по адресу http://numpy.sourceforge.net/.

math	Математические функции для работы с вещественными числами.
cmath	Математические функции для работы с комплексными числами.
random	Генерация псевдослучайных чисел с различными распределениями.
whrandom	Генератор псевдослучайных чисел.
bisect	Поддержание последовательностей в сортированном состоянии. Реализует алгоритм поиска путем деления пополам.
array	Эффективные массивы чисел одинакового типа.

18.1 math — математические функции для работы с вещественными числами

Модуль math всегда доступен и предоставляет доступ к стандартным математическим функциям. Эти функции не работают с комплексными числами — для этого следует использовать одноименные функции из модуля cmath.

acos(*x*)
: Возвращает арккосинус *x*.

asin(*x*)
: Возвращает арксинус *x*.

atan(*x*)
: Возвращает арктангенс *x*.

atan2(*x*, *y*)
: Эквивалентно atan(*x*/*y*). Аргумент *y* может быть равен 0 — в этом случае возвращает pi/2.

ceil(*x*)
Возвращает наименьшее вещественное число с нулевой дробной частью большее, чем *x*.

cos(*x*)
Возвращает косинус *x*.

cosh(*x*)
Возвращает гиперболический косинус *x*.

exp(*x*)
Возвращает `e**x`.

fabs(*x*)
Возвращает абсолютное значение *x*.

floor(*x*)
Возвращает наибольшее вещественное число с нулевой дробной частью меньшее, чем *x*.

fmod(*x*, *y*)
Результат этой функции зависит от реализации одноименной функции библиотеки языка C. Обычно дает такой же результат, как `x % y`.

frexp(*x*)
Возвращает пару '`(m, e)`', где *m* — мантисса (вещественное число) и *e* — экспоненциальная часть (целое число). Для чисел *m* и *e* всегда выполняется условие '`x == m * 2**e`'. Если аргумент *x* равен нулю, возвращает '`(0.0, 0)`'. В противном случае всегда выполняется '`0.5 <= abs(m) < 1`'.

hypot(*x*, *y*)
Возвращает евклидово кодовое расстояние, '`sqrt(x*x + y*y)`'.

ldexp(*m*, *e*)
Функция, обратная `frexp()`. Возвращает '`m * (2**e)`'.

log(*x*)
Возвращает натуральный логарифм *x*.

log10(*x*)
Возвращает десятичный логарифм *x*.

modf(*x*)
Возвращает кортеж из пары вещественных чисел — дробной и целой части *x*. Оба возвращаемых числа имеют такой же знак, как у числа *x*.

pow(*x*, *y*)
Возвращает `x**y`.

sin(*x*)
Возвращает синус *x*.

sinh(*x*)
: Возвращает гиперболический синус x.

sqrt(*x*)
: Возвращает квадратный корень из x.

tan(*x*)
: Возвращает тангенс x.

tanh(*x*)
: Возвращает гиперболический тангенс x.

Обратите внимание, что функции `frexp()` и `modf()` имеют другой интерфейс, нежели их эквиваленты в языке C: они возвращают пару значений вместо того, чтобы возвращать второе значение через аргумент.

Модуль также определяет две константы:

pi
: Число π.

e
: Число e.

18.2 cmath — математические функции для работы с комплексными числами

Модуль `cmath` всегда доступен и предоставляет доступ к математическим функциям, работающими с комплексными числами.

acos(*x*)
: Возвращает арккосинус x.

acosh(*x*)
: Возвращает гиперболический арккосинус x.

asin(*x*)
: Возвращает арксинус x.

asinh(*x*)
: Возвращает гиперболический арксинус x.

atan(*x*)
: Возвращает арктангенс x.

atanh(*x*)
Возвращает гиперболический арктангенс x.

cos(*x*)
Возвращает косинус x.

cosh(*x*)
Возвращает гиперболический косинус x.

exp(*x*)
Возвращает `e**`x.

log(*x*)
Возвращает натуральный логарифм x.

log10(*x*)
Возвращает десятичный логарифм x.

sin(*x*)
Возвращает синус x.

sinh(*x*)
Возвращает гиперболический синус x.

sqrt(*x*)
Возвращает квадратный корень из x.

tan(*x*)
Возвращает тангенс x.

tanh(*x*)
Возвращает гиперболический тангенс x.

Обратите внимание, что функции `frexp()` и `modf()` имеют другой интерфейс, нежели их эквиваленты в языке C: они возвращают пару значений вместо того, чтобы возвращать второе значение через аргумент.

Модуль также определяет две константы:

pi
Число π (вещественное).

e
Число e (вещественное).

Заметим, что эти функции похожи, но не идентичны одноименным функциям в модуле `math`. Различия проявляются даже при использовании вещественных аргументов. Например, `math.sqrt(-1)` генерирует исключение, в то время как `cmath.sqrt(-1)` возвращает `1j`. Кроме того, функции модуля `cmath` всегда возвращают комплексное число, даже если оно может быть выражено в рамках вещественного (в этом случае число имеет нулевую мнимую часть).

18.3 random — псевдослучайные числа с различными распределениями

Этот модуль определяет функции, генерирующие псевдослучайные числа с различными распространенными распределениями. Все они используют генератор псевдослучайных чисел с равномерным распределением, предоставляемый модулем whrandom. Аргументы описанных здесь функций имеют имена в соответствии с общепринятой в математике практикой обозначения параметров уравнений распределений.

betavariate(*alpha*, *beta*)
: Бета-распределение. Оба аргумента должны быть больше -1, возвращает значение между 0 и 1.

cunifvariate(*mean*, *arc*)
: Циклическое равномерное распределение. *mean* — средний угол, *arc* — ширина диапазона. Оба аргумента должны быть в диапазоне от 0 до π. Возвращаемое значение находятся в диапазоне от *mean* - *arc*/2 до *mean* + *arc*/2 (также приводится к диапазону от 0 до π).

expovariate(*lambda*)
: Экспоненциальное распределение. Аргумент *lambda* равен единице, деленной на желаемое математическое ожидание распределения. Возвращает значения от 0 до $+\infty$.

gamma(*alpha*, *beta*)
: Гамма-распределение. Аргументы должны удовлетворять условиям *alpha* > -1 и *beta* > 0.

gauss(*mu*, *sigma*)
normalvariate(*mu*, *sigma*)
: Нормальное распределение (Гаусса). *mu* — математическое ожидание и *sigma* — стандартное отклонение. Функция gauss() имеет немного более быструю реализацию.

lognormvariate(*mu*, *sigma*)
: Логарифмически нормальное распределение. Натуральный логарифм от значений, возвращаемых этой функцией, дает нормальное распределение с математическим ожидание *mu* и стандартным отклонением *sigma*. *mu* может иметь любое значения, *sigma* должен быть больше нуля.

vonmisesvariate(*mu*, *kappa*)
: Распределение фон Мизеса. *mu* — математическое ожидание, выраженное в радианах от 0 до 2π, *kappa* — коэффициент кучности, который должен быть больше или равен нулю. Если аргумент *kappa* равен нулю, это распределение вырождается в равномерное распределение в диапазоне от 0 до 2π.

paretovariate(*alpha*)
: Распределение Парето. *alpha* — параметр формы распределения.

weibullvariate(*alpha*, *beta*)
Распределение Вейбулла. *alpha* — масштабный коэффициент, *beta* — параметр формы распределения.

Для удобства модуль random также экспортирует имена random(), uniform(), randrange(), randint() и choice() из модуля whrandom и переопределяет seed():

seed([*obj*])
Инициализирует генератор случайных чисел хэш-значением объекта *obj*. Если аргумент *obj* опущен или равен None, использует для инициализации текущее время (аналогично whrandom.seed()).

18.4 whrandom — генератор псевдослучайных чисел

Этот модуль определяет класс whrandom, реализующий генератор псевдослучайных чисел Вичмана-Хилла[1]:

whrandom([*x*, *y*, *z*])
Создает, инициализирует, вызывая метод seed(), и возвращает объект-генератор.

Экземпляры класса whrandom имеют следующие методы:

seed([*x*, *y*, *z*])
Инициализирует генератор целыми числами *x*, *y* и *z*. Числа должны быть в диапазоне от 0 до 255 (включительно). Если аргументы опущены или равны нулю, используются значения, производные от текущего времени. Если один или два аргумента равны нулю, нулевые значения заменяются единицей.

random()
Возвращает случайное вещественное число *r* такое, что 0.0 <= *r* < 1.0.

uniform(*a*, *b*)
Возвращает случайное вещественное число *r* такое, что *a* <= *r* < *b*.

randrange([*start*,] *stop* [, *step*])
Возвращает случайное целое число из 'range(*start*, *stop*, *step*)'.

randint(*a*, *b*)
Возвращает случайное целое число из 'range(*a*, *b* +1)'. Этот метод оставлен лишь для совместимости со старыми версиями модуля — используйте randrange().

[1] Wichmann, B. A. and Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", *URL* 31 (1982) 188–190.

choice(*seq*)
Выбирает случайный элемент из непустой последовательности *seq* и возвращает его.

При импортировании модуль whrandom создает экземпляр класса whrandom и делает его методы seed(), random(), uniform(), randrange(), randint() и choice() доступными в глобальном пространстве имен модуля. Заметим, что использование отдельных экземпляров класса whrandom приведет к использованию независимых последовательностей псевдослучайных чисел.

Методы экземпляров whrandom дают случайные числа с равномерным распределением. Если Вам необходимо получить случайные числа с другими распределениями, воспользуйтесь функциями, определенными в модуле random.

18.5 bisect — поддержание последовательностей в сортированном состоянии

Этот модуль предоставляет возможность поддерживать последовательности в сортированном состоянии без необходимости в сортировке после каждого добавления элемента, что особенно важно для больших списков, сравнение элементов которого является дорогой операцией. Для выполнение этой задачи модуль реализует алгоритм поиска путем деления пополам.

Модуль bisect предоставляет следующие функции:

bisect(*list*, *item* [, *lo* [, *hi*]])
Находит место в списке (индекс), в которое необходимо вставить *item*, для того, чтобы сохранить список в сортированном состоянии. Аргументы *lo* и *hi* могут быть использованы для указания подмножества списка, которое следует принимать в рассмотрение. Возвращаемое значения подходит для использования в качестве первого аргумента метода *list*.insert().

insort(*list*, *item* [, *lo* [, *hi*]])
Вставляет *item* в список *list*, сохраняя сортированное состояние списка. Эквивалентно '*list*.insert(bisect.bisect(*list*, *item*, *lo*, *hi*), *item*)'.

Функция bisect() может быть полезна для распределения данных по категориям. В следующем примере эта функция используется для определения оценки (обозначаемой буквой, как это принято во многих западных странах) за экзамен по общему количеству набранных баллов. Пусть 85 и более баллов соответствует оценке 'A', 75–84 — 'B' и т. д.:

```
>>> grades = "FEDCBA"
>>> breakpoints = [30, 44, 66, 75, 85]
```

```
>>> from bisect import bisect
>>> def grade(total):
...           return grades[bisect(breakpoints, total)]
...
>>> grade(66)
'C'
>>> map(grade, [33, 99, 77, 44, 12, 88])
['E', 'A', 'B', 'D', 'F', 'A']
```

18.6 array — эффективные массивы чисел

Этот модуль определяет новый тип объектов `array`, эффективно реализующий массивы значений основных типов: символов, целых и вещественных чисел. Массивы ведут себя аналогично спискам с одним исключением: все сохраняемые в массиве объекты должны быть одного определенного типа. Тип элементов определяется при создании массива одним из следующих символов, идентифицирующих тип:

Символ	**Тип языка C**	**Минимальный размер в байтах**
'c'	`char` (символ)	1
'b'	`signed char` (знаковое целое)	1
'B'	`unsigned char` (беззнаковое целое)	1
'h'	`signed short` (знаковое целое)	2
'H'	`unsigned short` (беззнаковое целое)	2
'i'	`signed int` (знаковое целое)	2
'I'	`unsigned int` (беззнаковое целое)	2
'l'	`signed long` (знаковое целое)	4
'L'	`unsigned long` (беззнаковое целое)	4
'f'	`float` (вещественное число)	4
'd'	`double` (вещественное число)	8

Реальное представление значений зависит от архитектуры машины. Используемый размер доступен через атрибут `itemsize` массивов. При извлечении элементов из массивов, при конструировании которых использовался символ `'L'` или `'I'`, значение представляется типом `long int` языка Python, так как в рамках типа `int` не может быть представлен весь диапазон чисел типов `unsigned int` и `unsigned long`.

`array``(`*`typecode`* `[,` *`initializer`*`])`

Создает новый объект-массив, тип элементов которого определяется символом *`typecode`*, инициализирует элементами последовательности *`initializer`* (должен быть списком или обычной строкой; *`initializer`* передается методу `fromlist()` или `fromstring()`) и возвращает его.

ArrayType
Объект типа, соответствующий объектам, которые возвращает функция `array()`.

Массивы имеют следующие атрибуты данных и методы:

typecode
Символ, определяющий тип элементов массива.

itemsize
Размер в байтах внутреннего представления одного элемента массива.

append(*x*)
Добавляет *x* в конец массива.

buffer_info()
Возвращает кортеж с адресом и длиной в байтах буфера, используемого для хранения содержимого массива. Возвращаемые значения действительны до тех пор, пока массив существует и к нему не применяются операции, изменяющие длину.

byteswap()
Изменяет порядок следования байтов для всех элементов массива. Эта операция может быть выполнена только для значений, внутреннее представление которых имеет размер 1, 2, 4 или 8 байт. Метод может быть полезен при чтении данных из файла, который был записан на машине с другим порядком следования байтов.

count(*x*)
Возвращает число элементов массива равных *x*.

extend(*a*)
Добавляет элементы массива *a* в конец массива.

fromfile(*f*, *n*)
read(*f*, *n*)
Считывает и добавляет в конец массива *n* элементов (машинное представление) из файла, представленного файловым объектом (встроенного типа `file`) *f*. Если файл содержит меньше элементов, чем *n*, генерируется исключение `EOFError`, но все прочитанные к этому времени элементы добавляются в массив.

Имя `read()` присутствует для совместимости со старыми версиями.

fromlist(*list*)
Добавляет в конец массива элементы из списка *list*. Это эквивалентно инструкции '`for x in list: a.append(x)`' с одним исключением: если список содержит элементы неверного типа, массив остается без изменений.

fromstring(*s*)
Добавляет в конец массива элементы из строки *s*, интерпретируя строку как машинное представление значений (то есть аналогично тому, как это делает метод `fromfile()`). Число символов в строке должно быть кратным размеру элементов массива.

index(*x*)
Возвращает индекс первого элемента массива равного *x*. Если массив не содержит таких значений, генерируется исключение `ValueError`.

insert(*i*, *x*)
Вставляет значение *x* в массив перед элементом с индексом *i*.

pop([*i*])
Удаляет из массива элемент с индексом *i* и возвращает его. Использование отрицательных индексов позволяет вести отсчет с конца. По умолчанию аргумент *i* равен `-1`, то есть соответствует последнему элементу.

remove(*x*)
Удаляет из массива первый элемент равный *x*.

reverse()
Меняет порядок следования элементов массива на обратный.

tofile(*f*)
write()
Записывает все элементы массива (машинное представление) в файл, представленный файловым объектом *f*. Имя `write()` присутствует для совместимости со старыми версиями.

tolist()
Возвращает список элементов массива.

tostring()
Возвращает строку с машинным представлением элементов массива (та же самая последовательность байтов, которая записывается в файл методом `tofile()`).

Глава 19

Интерфейсные классы к встроенным типам

Модули, описанные в этой главе, позволяют определять новые классы, наследующие возможности встроенных типов.

UserString	Интерфейсный класс для создания строковых объектов.
UserList	Интерфейсный класс для создания последовательностей.
UserDict	Интерфейсный класс для создания отображений.

19.1 `UserString` — интерфейсный класс для создания строковых объектов

Этот модуль определяет интерфейсный класс, предназначенный для использования в качестве базового класса при определении классов, ведущих себя аналогично строкам. Вы можете переопределять существующие методы и добавлять новые, таким образом изменяя поведение или добавляя новые возможности.

UserString([*initialdata*])
Возвращает экземпляр класса, который ведет себя аналогично объектам встроенного типа `string` (обычная строка) или `unicode` (строка Unicode). Экземпляр инициализируется из *initialdata* (объект произвольного типа). Если *initialdata* не является строкой (`string` или `unicode`), экземпляр инициализируется строковым представлением объекта (`str(`*initialdata*`)`). Если аргумент *initialdata* опущен, экземпляр инициализируется пустой строкой. Данные из *initialdata* сохраняются в виде строки в атрибуте `data` созданного экземпляра.

В дополнение к методам и операциям, характерным для строк (см. раздел 11.2.1), экземпляры класса `UserString` имеют следующий атрибут:

data
Строка (`string` или `unicode`), в которой хранятся данные.

В качестве примера класса, производного от `UserString` модуль определяет еще один класс:

`MutableString``([`*`initialdata`*`])`
Возвращает экземпляр класса, представляющий изменяемую строку. Изменяемые строки не могут быть использованы в качестве ключей в словарях: *вычисление хэш-значения изменяемых объектов, для которых определена операция сравнения иначе, чем простое сравнение идентификаторов, бессмысленно и может привести к ошибкам, которые трудно обнаружить.* Класс `MutableString` определен здесь в основном для того, чтобы пользователь не определил аналогичный собственный класс, забыв при этом переопределить специальный метод `__hash__()`.

Помимо унаследованных от `UserString`, экземпляры класса `MutableString` поддерживают операции, позволяющие изменить объект: изменение и удаление символа (элемента последовательности) и подстроки (среза последовательности) — часть операций, характерных для изменяемых последовательностей (см. раздел 11.2.6).

19.2 `UserList` — интерфейсный класс для создания последовательностей

Этот модуль определяет интерфейсный класс, предназначенный для использования в качестве базового класса при определении последовательностей. Вы можете переопределять существующие методы и добавлять новые, таким образом изменяя поведение или добавляя новые возможности.

Модуль определяет единственное имя — класс `UserList`:

`UserList``([`*`initialdata`*`])`
Возвращает экземпляр класса, который ведет себя аналогично объектам встроенного типа `list` (список). Экземпляр инициализируется данными, взятыми из *`initialdata`* (последовательность произвольного типа). Если аргумент *`initialdata`* опущен, экземпляр изначально не содержит элементов. Данные из *`initialdata`* копируются и сохраняются в виде списка в атрибуте `data` созданного экземпляра.

В дополнение к методам и операциям, характерным для всех изменяемых последовательностей (см. раздел 11.2.6), экземпляры класса `UserList` имеют следующий атрибут:

`data`
Список, в котором хранятся данные.

19.3 UserDict — интерфейсный класс для создания отображений

Этот модуль определяет интерфейсный класс, предназначенный для использования в качестве базового класса при определении отображений. Вы можете переопределять существующие методы и добавлять новые, таким образом изменяя поведение или добавляя новые возможности.

Модуль определяет единственное имя — класс UserDict:

UserDict([*initialdata*])
Возвращает экземпляр класса, который ведет себя аналогично объектам встроенного типа dictionary (словарь). Экземпляр инициализируется данными, взятыми из *initialdata* (отображение произвольного типа). Если аргумент *initialdata* опущен, экземпляр изначально не содержит записей. Данные из *initialdata* копируются и сохраняются в виде словаря в атрибуте data созданного экземпляра.

В дополнение к методам и операциям, характерным для всех отображений (см. раздел 11.3), экземпляры класса UserDict имеют следующий атрибут:

data
Словарь, в котором хранятся данные.

Глава 20

Сохранение и копирование объектов

Модули, описанные в этой главе, предоставляют возможность сохранения и копирования объектов языка Python.

pickle Преобразует объекты языка Python в последовательность байтов и обратно.

cPickle Более быстрый вариант модуля pickle.

shelve Сохранение объектов в базе данных в стиле DBM.

marshal Позволяет получить байт-компилированное представление объектов кода (и сопутствующих им объектов) и восстановить объекты из их байт-компилированного представления.

struct Преобразование объектов в структуры языка C.

20.1 pickle и cPickle — представление объектов в виде последовательности байтов

Модуль pickle реализует простой, но мощный алгоритм “консервирования” (pickling, преобразования объектов в последовательность байтов) почти любых объектов языка Python с возможностью дальнейшего восстановления (unpickling). Однако этот модуль не заботится о сохранении имен объектов и не обеспечивает к ним совместный доступ. Полученную последовательность байтов Вы можете записать в файле, сохранить в базе данных или переслать по сети на другую машину. Модуль shelve предоставляет простой интерфейс для сохранения “законсервированных” объектов в базе данных в стиле DBM.

В то время как реализованный на языке Python модуль pickle работает достаточно медленно, реализованный на C модуль cPickle работает иногда почти в 1000 раз быстрее. Модуль cPickle использует точно такой же алгоритм и имеет аналогичный интерфейс, за исключением того, что вместо классов Pickler и Unpickler используются функции, возвращающие объекты с аналогичным интерфейсом.

Хотя модули pickle и cPickle и используют внутри себя встроенный модуль marshal, они отличаются от этого модуля способом обработки данных:

- Модули pickle и cPickle отслеживают объекты, которые уже были обработаны, и никогда не обрабатывают их повторно. Это позволяет "консервировать" рекурсивные объекты (объекты, содержащие ссылки на себя), которые не могут быть обработаны модулем marshal. Аналогично обрабатываются совместно используемые объекты (различные объекты ссылаются на один и тот же объект): они остаются совместно используемыми, что очень важно для изменяемых объектов.

- Модуль marshal совсем не поддерживает экземпляры классов, в то время как модули pickle и cPickle позволяют с ними работать. Определение класса должно находиться в том же модуле, в котором оно было при "консервации" объекта.

Используемый формат данных характерен для языка Python и не имеет ограничений, характерных, например, для стандартного формата XDR (External Data Representation), который не позволяет представить совместное использование объектов. Однако это означает, что программы на других языках не смогут восстановить "законсервированный" объект.

По умолчанию для представления используются только печатные ASCII символы, что позволяет просматривать файл с данными с помощью обычного текстового редактора с целью отладки или восстановления данных. С другой стороны, такое представление занимает больше места, чем двоичный формат. Вы можете выбрать двоичный формат, используя ненулевое значение аргумента *bin* конструктора Pickler и функций dump() и dumps().

Модули pickle и cPickle не поддерживают объекты кода. В этом нет большой необходимости, так как для "консервирования" объектов кода традиционно используется модуль marshal. Кроме того, это позволяет избежать возможного включения в файлы данных троянских коней.

Для удобства работы с долгоживущими модулями предоставляется поддержка ссылок на объекты, которые не подлежат "консервированию". Для таких объектов запоминается идентификатор, возвращаемый атрибутом-функцией или методом persistent_id(). Идентификатор может быть произвольной строкой из печатных ASCII символов. Для восстановления объекта по идентификатору необходимо определить атрибут-функцию persistent_load().

На "консервирование" экземпляров классов есть несколько ограничений. Во-первых, класс должен быть определен в глобальном пространстве имен одного из модулей. Во-вторых, все переменные экземпляра (то есть атрибут __dict__) должны поддерживать "консервацию" или же экземпляр класса должен иметь метод __getstate__() протокола копирования, возвращающий объект, который поддерживает "консервацию".

При восстановлении экземпляра класса его метод __init__() обычно *не* вызывается. Для того, чтобы он вызывался, класс должен иметь метод __getinitargs__() протокола копирования. Другие методы протокола копирования (см. раздел **??**) __getstate__() и __setstate__() используются для сохранения и восстановления состояния объекта.

Обратите внимание, что при “консервации” экземпляра класса сам объект-класс и, соответственно, его атрибуты не сохраняются — сохраняются только атрибуты экземпляра и запоминается имя класса и модуля, в котором класс определен. Поэтому класс должен быть определен в глобальном пространстве имен модуля, который, однако, в момент восстановления может быть и не импортирован. Это сделано с целью иметь возможность исправить ошибки в определении класса или добавить новые методы и загрузить объект, созданный со старой версией класса. Если Вы собираетесь работать с долгоживущими объектами, которые переживут множество версий класса, разумно сохранить номер версии класса, чтобы затем необходимые преобразования могли быть выполнены методом `__setstate__()`.

Модули `pickle` и `cPickle` определяют следующие конструкторы:

`Pickler``(`*`file`* `[, `*`bin`*`])`
: Класс (в модуле `pickle`) или функция (в модуле `cPickle`), возвращает объект, реализующий “консервирование”. Аргумент *`file`* должен быть файловым объектом, имеющим метод `write()`. Если задан отличный от нуля аргумент *`bin`* (целое число), используется более компактный двоичный формат.

`Unpickler``(`*`file`*`)`
: Класс (в модуле `pickle`) или функция (в модуле `cPickle`), возвращает объект, реализующий восстановление “законсервированного” объекта. Аргумент *`file`* должен быть файловым объектом, имеющим методы `read()` и `readline()`.

Объекты, возвращаемые конструктором `Pickler()`, имеют следующие (основные) методы:

`dump``(`*`object`*`)`
: “Консервирует” объект *`object`*.

`persistent_id``(`*`object`*`)`
: Этот метод (функция или любой другой объект, поддерживающий вызов) вызывается для каждого из вложенных объектов. Должен возвращать строку-идентификатор постоянного объекта или `None`, если объект подлежит консервации. Изначально метод `persistent_id()` не определен (`cPickle`) или всегда возвращает `None` (`pickle`).

Объекты, возвращаемые конструктором `Unpickler()`, имеют методы, предназначенные для выполнения обратных действий:

`load``()`
: Восстанавливает и возвращает ранее “законсервированный” объект.

`noload``()` (только в модуле cPickle)
: Проходит весь цикл восстановления объекта, но сам объект не создает. Может быть полезен для тестирования `persistent_load`

persistent_load(*id_string*)
: Этот метод (функция или любой другой объект, поддерживающий вызов) вызывается для восстановления ссылки на постоянный объект по строке-идентификатору *id_string*, которая при "консервации" была возвращена методом persistent_id(). Изначально метод persistent_load() не определен.

Вы можете несколько раз вызывать метод dump() объекта Pickler. Для восстановления всех "законсервированных" объектов необходимо будет столько же раз вызвать метод load() соответствующего объекта Unpickler(). Если один и тот же объект "консервируется" несколько раз методом dump() одного и того же объекта Pickler, метод load() восстановит ссылки на один и тот же объект. Предполагается, что "консервация" нескольких объектов производится без внесения в них изменений в промежутке между вызовами метода load(). Если Вы измените объект и затем повторно "законсервируете" его с помощью того же объекта Pickler, объект не будет "законсервирован" — сохранится лишь ссылка на старое значение.

Кроме конструкторов Pickler() и Unpickler(), модули определяют следующие функции и объекты данных:

dump(*object, file* [, *bin*])
: "Консервирует" объект *object* в файл *file*. Эквивалентно вызову 'Pickler(*file*, *bin*).dump(*object*)'.

load(*file*)
: Восстанавливает "законсервированный" объект из файла *file* и возвращает его.

dumps(*object* [, *bin*])
: Возвращает "законсервированное" представление объекта в виде строки вместо того, чтобы записывать его в файл.

loads(*string*)
: Восстанавливает "законсервированный" объект из строки *string* и возвращает его.

format_version
: Строка-версия формата, используемого при "консервации".

compatible_formats
: Список старых версий (строк) формата данных, которые, помимо текущей версии (format_version), могут быть восстановлены.

Ниже приведена иерархия исключений, которые используются модулями pickle и cPickle при возникновении ошибок (до версии 2.0 исключения определены в виде строк, что сильно затрудняет их обработку):

PickleError
: Базовый класс для всех исключений, является производным от стандартного класса исключений Exception.

PicklingError
Класс исключений, которые генерируются при попытке "законсервировать" объект, который не может быть "законсервирован".

UnpicklingError
Класс исключений, которые генерируются при восстановлении объекта, если его конструктор не зарегистрирован (см. описание модуля `copy_reg`).

Ниже приведен список объектов, которые могут быть "законсервированы":

- `None`;
- Целые, длинные целые и вещественные числа.
- Простые строки и строки Unicode.
- Кортежи, списки и словари, содержащие только объекты, которые могут быть "законсервированы".
- Классы и функции, определенные в глобальном пространств имен модуля (на самом деле запоминаются только имена модуля и функции).
- Экземпляры классов, атрибут `__dict__` которых может быть "законсервирован".
- Экземпляры классов, имеющих соответствующие методы протокола копирования (см. раздел **??**).
- Объекты, поддержка которых зарегистрирована с помощью модуля `copy_reg`. Именно таким образом осуществляется поддержка комплексных чисел.

20.2 `shelve` — сохранение объектов в базе данных в стиле DBM

С помощью этого модуля Вы можете создать "стеллаж" (shelf) — объект с интерфейсом словаря, который позволяет сохранять в базе данных в стиле DBM объекты в "законсервированном" виде. Для консервации используется модуль `cPickle` или `pickle`, если первый недоступен.

open(*filename* [, *flag*])
Открывает файл *filename* с базой данных в стиле DBM и на его основе создает и возвращает объект-стеллаж. Флаг *flag* используется в качестве второго аргумента в функции `anydbm.open()` (см. описание модуля `anydbm`).

Объекты-стеллажи поддерживают большинство операций, характерных для отображений (см. раздел 11.3): доступ к объектам по ключу, добавление и удаление записей,

методы has_key(), keys(), get(). В качестве ключа могут использоваться только обычные строки. Методы sync() и close() вызывают соответствующие методы объекта, представляющего базу данных.

Модуль shelve не поддерживает одновременного доступа, если при этом производится запись в базу данных. Одновременный доступ на чтение в принципе безопасен.

20.3 marshal — байт-компилированное представление объектов

Этот модуль позволяет получить байт-компилированное представление объектов кода (code), а также сопутствующих им объектов: None; объектов, которые могут быть представлены литеральными выражениями (любые числа и строки); кортежей, списков и словарей, содержащих только объекты, для которых может быть получено байт-компилированное представление. Попытка получить байт-компилированное представление для рекурсивных объектов (содержащих ссылки на самих себя) приведет к зацикливанию. Формат байт-компилированного кода специфичен для языка Python, но не зависит от платформы. marshal не является модулем общего назначения. Для сохранения и передачи объектов следует использовать модули pickle, cPickle и shelve.

Модуль определяет следующие функции:

dump(*object*, *file*)
Записывает байт-компилированное представление объекта *object* в файл. Аргумент *file* должен быть объектом типа file, открытым для записи в двоичном режиме ('wb' или 'w+b'). Если для объекта *object* не может быть получено байт-компилированное представление, генерируется исключение ValueError, но при этом в файл будет записан "мусор", который не будет корректно считан функцией load().

load(*file*)
Считывает байт-компилированное представление для одного объекта из файла, восстанавливает его и возвращает. Если данные в файле не могут быть корректно обработаны, в зависимости от ситуации генерируется одно из исключений EOFError, ValueError или TypeError. Аргумент *file* должен быть объектом типа file, открытым для чтения в двоичном режиме ('rb' или 'r+b').

dumps(*object*)
Возвращает строку с байт-компилированным представлением объекта *object*.

loads(*string*)
Восстанавливает объект из байт-компилированного представления *string*. Лишние символы в строке игнорируются.

20.4 `struct` — преобразование объектов в структуры языка C

Этот модуль позволяет преобразовывать значения некоторых объектов в структуры языка C в виде строк и обратно. Данные в строке располагаются в соответствии со строкой формата. Эти возможности могут быть использованы для чтения и сохранения двоичных данных, пересылки данных через сетевое соединение.

`error`
: Класс исключений, которые генерируются при различных ошибках. В качестве аргумента используется строка, описывающая подробности.

`pack`(*format*, *value1* ...)
: Возвращает строку, содержащую значения *value1* ..., упакованные в соответствии с форматом. Количество и тип аргументов должны соответствовать значениям, которые требует строка формата *format*.

`unpack`(*format*, *string*)
: Распаковывает строку *string* в соответствии с форматом *format* и возвращает кортеж объектов. Строка должна содержать ровно такое количество данных, которое требует строка формата, то есть длина строки должна быть равной `calcsize(`*format*`)`.

`calcsize`(*format*)
: Возвращает размер структуры (то есть длину строки), соответствующей формату *format*.

Символы строки формата имеют следующее значение:

Символ	Тип языка C	Тип объекта в языке Python
x	— (пустой байт)	— (пустой байт)
c	`char`	символ (строка длиной 1)
b	`signed char`	`int`
B	`unsigned char`	`int`
h	`short`	`int`
H	`unsigned short`	`int`
i	`int`	`int`
I	`unsigned int`	`long int` (`int`, если в языке C тип `int` меньше, чем `long`)
l	`long`	`int`
L	`unsigned long`	`long int`
f	`float`	`float`
d	`double`	`float`
s	`char[]`	`string`
p	`char[]`	`string`
P	`void *`	`int`

Перед символом формата может идти число, обозначающее количество повторений. Например, строка формата '4h' полностью эквивалентна строке 'hhhh'. Символы пропуска между символами формата игнорируются, однако символы пропуска между числом и символом формата не допускаются.

Число перед символом формата ‘s’ интерпретируется как длина строки, а не число повторений. То есть '10s' обозначает строку из 10 символов, в то время как '10c' — 10 раз по одному символу. При упаковке строки урезаются или дополняются до нужной длины символами с кодом 0. При распаковке строки имеют указанную в строке формата длину.

Символ формата ‘p’ может быть использован для упаковки строк в стиле языка Pascal. В этом случае первый байт является длиной строки, затем следует сама строка. Число перед символом ‘p’ указывает на длину строки вместе с байтом, в котором сохранится длина строки.

По умолчанию для представления используется “родной” для данной платформы формат и порядок следования байтов. Поместив в начало строки формата один из описанных ниже символов, Вы можете определить порядок следования байтов, размер и выравнивание структур.

Символ	Порядок следования байтов	Размер и выравнивание
@	родной	родные
=	родной	стандартные
<	little-endian	стандартные
>	big-endian	стандартные
!	общепринятый в сети (= big-endian)	стандартные

Символ формата ‘P’ может быть использован только с “родным” порядком следования байтов.

Для того, чтобы выравнять конец структуры в соответствии с определенным типом, можно поместить в конец строки формата символ, определяющий этот тип с числом повторения равным 0. Например, формат 'llh0l' определяет дополнительные два байта в конце (подразумевая, что тип long выравнивается в рамках четырех байтов). Этот способ работает только с “родными” размером и выравниванием.

Приведем небольшой пример использования модуля struct (платформа Intel, little-endian):

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
'\001\000\002\000\003\000\000\000'
>>> unpack('hhl', '\001\000\002\000\003\000\000\000')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Глава 21

Доступ к средствам, предоставляемым операционной системой

Модули, описанные в этой главе, предоставляют общий интерфейс к службам, доступным во всех (или почти всех) операционных системах.

os	Основные службы операционной системы.
os.path	Работа с именами путей.
stat	Средства для интерпретации значений, возвращаемых функциями `os.stat()`, `os.lstat()` и `os.fstat()`.
statvfs	Константы, предназначенные для интерпретации значений, возвращаемых функцией `os.statvfs()`.
filecmp	Сравнение файлов и каталогов.
popen2	Доступ к потокам ввода/вывода дочерних процессов.
time	Определение и обработка времени.
sched	Планировщик задач общего назначения.
getpass	Переносимый способ запросить пароль и определить имя пользователя.
getopt	Обработка опций в командной строке.
tempfile	Создание временных файлов.
errno	Символические имена стандартных системных ошибок.
glob	Раскрытие шаблона имен путей в стиле UNIX shell.
fnmatch	Сопоставление имен файлов с шаблоном в стиле UNIX shell.
shutil	Операции над файлами высокого уровня.
signal	Обработка асинхронных событий.
socket	Сетевой интерфейс низкого уровня.
select	Асинхронные операции ввода/вывода с участием нескольких файловых дескрипторов.
mmap	Отображение файлов в память.

21.1 os — основные службы операционной системы

Модуль os позволяет переносимо использовать основные службы операционной системы (ОС). Этот модуль производит поиск встроенного модуля, характерного для данной ОС (posix, nt, mac и др.) и экспортирует определенные в нем функции и объекты данных. Модуль спроектирован таким образом, чтобы обеспечить одинаковый интерфейс в тех случаях, когда для разных ОС доступны одинаковые возможности. Функции, определенные только для некоторых ОС также доступны через модуль os, однако их использование не переносимо.

error
Является ссылкой на встроенный класс исключений OSError (см. раздел 13).

name
Имя зависящего от ОС модуля. В настоящее время могут быть использованы следующие имена: 'posix', 'nt', 'dos', 'mac', 'os2', 'ce', 'java'.

path
Ссылка на зависящий от операционной системы модуль для работы с именами файлов и путями, например, posixpath или macpath. Этот модуль может быть импортирован напрямую как os.path.

strerror(*code*) (UNIX, Windows)
Возвращает строку с сообщением о системной ошибке, соответствующим коду *code*.

21.1.1 Параметры процесса

Функции и объекты данных, описанные в этом разделе, предоставляют информацию и средства управления для текущего процесса.

environ
Отображение имен переменных окружения к их значениям. Например, environ['HOME'] дает значение переменной окружения HOME. Если на используемой платформе доступна функция putenv(), отображение environ может быть также использовано для изменения значений переменных окружения. Функция putenv() автоматически вызывается при внесении изменений в environ. Если функция putenv() не доступна, Вы можете использовать измененное отображение в качестве аргумента функции, создающей дочерний процесс.

chdir(*path*)
getcwd()
Эти функции описаны в разделе 21.1.4.

ctermid() (UNIX)
Возвращает имя файла, соответствующего контролируемому текущим процессом терминалу.

getegid() (UNIX)
Возвращает идентификатор эффективной группы пользователей текущего процесса.

geteuid() (UNIX)
Возвращает идентификатор эффективного пользователя текущего процесса.

getgid() (UNIX)
Возвращает идентификатор группы пользователей текущего процесса.

getgroups() (UNIX)
Возвращает список идентификаторов дополнительных групп, ассоциированных с текущим процессом.

getlogin() (UNIX)
Возвращает настоящее регистрационное имя пользователя для текущего процесса, даже если существует несколько таких имен для одного и того же идентификатора пользователя.

getpgrp() (UNIX)
Возвращает идентификатор группы процессов текущего процесса.

getpid() (UNIX, Windows)
Возвращает идентификатор текущего процесса.

getppid() (UNIX)
Возвращает идентификатор родительского процесса.

getuid() (UNIX)
Возвращает идентификатор пользователя текущего процесса.

putenv(*varname*, *value*) (большинство вариантов UNIX, Windows)
Устанавливает значение переменной окружения с именем *varname* равным строке *value*. Эти изменения отражаются на дочерних процессах.

setegid(*egid*) (UNIX)
Устанавливает идентификатор эффективной группы пользователей текущего процесса равным *egid*.

seteuid(*euid*) (UNIX)
Устанавливает идентификатор эффективного пользователя текущего процесса равным *euid*.

setgid(*gid*) (UNIX)
Устанавливает идентификатор группы пользователей текущего процесса равным *egid*.

setpgrp() (UNIX)
Устанавливает идентификатор группы процессов текущего процесса равным идентификатору процесса (эквивалентно вызову 'setpgid(0, 0)').

setpgid(*pid*, *pgrp*) (UNIX)
Устанавливает идентификатор группы процессов процесса с идентификатором *pid* равным *pgrp*. Аргумент *pid* равный 0 обозначает текущий процесс. Если аргумент *pgrp* равен 0, в качестве идентификатора группы используется идентификатор процесса.

setreuid(*ruid*, *euid*) (UNIX)
Устанавливает идентификаторы реального и эффективного пользователей текущего процесса равными *ruid* и *euid* соответственно.

setregid(*rgid*, *egid*) (UNIX)
Устанавливает идентификаторы реальной и эффективной групп пользователей текущего процесса равными *rgid* и *egid* соответственно.

setsid() (UNIX)
Создает новые сессию и группу процессов с идентификаторами, равными идентификатору текущего процесса.

setuid(*uid*) (UNIX)
Устанавливает идентификатор пользователя текущего процесса равным *uid*.

umask(*mask*) (UNIX, Windows)
Устанавливает маску прав на доступ к файлам, которые будут созданы текущим процессом.

uname() (некоторые современные варианты UNIX)
Возвращает кортеж из пяти строк: имя системы, имя узла, выпуск, версия, тип машины. Некоторые системы обрезают имя узла, поэтому лучше воспользоваться вызовом socket.gethostname() или даже socket.gethostbyaddr(socket.gethostname()) (см. описание модуля socket).

21.1.2 Создание файловых объектов

Приведенные ниже функции создают новые файловые объекты.

fdopen(*fd* [, *mode* [, *bufsize*]]) (UNIX, Windows, Macintosh)
Возвращает файловый объект, связанный с файловым дескриптором *fd*. Аргументы *mode* и *bufsize* имеют такое же значение, как и соответствующие аргументы встроенной функции open().

popen(*command* [, *mode* [, *bufsize*]]) (UNIX, Windows)

Создает канал (pipe) с командой *command* и возвращает связанный с ним файловый объект. Вы можете читать вывод команды из файлового объекта или записывать в него входные данные в зависимости от режима *mode*: 'r' (чтение, используется по умолчанию) или 'w' (запись). Аргумент *bufsize* имеет такое же значение, как и соответствующий аргумент встроенной функции open(). Код завершения выполненной команды (в формате, аналогичном формату результата функции wait()) возвращается методом close() файлового объекта. Однако, если выполнение команды завершилось успешно (код завершения равен 0), метод close() возвращает None.

В предыдущих версиях под Windows эта функция вела себя ненадежно из-за некорректной работы функции _popen() библиотеки Windows. Начиная с версии 2.0, используется новая реализация.

tmpfile() (UNIX)

Создает и возвращает файловый объект для временного файла, открытого в режиме обновления ('w+'). Этот файл будет автоматически удален после уничтожения всех ассоциированных с ним файловых дескрипторов.

Для всех функций вида popen*() аргумент *bufsize* определяет размер буфера каналов и *mode* — режим, в котором будут открыты каналы ('t' — текстовый, используется по умолчанию, или 'b' — двоичный). Аналогичные функции доступны в модуле popen2, но элементы возвращаемого кортежа для них располагаются в ином порядке.

popen2(*cmd* [, *bufsize* [, *mode*]]) (UNIX, Windows)

Выполняет команду *cmd* в качестве дочернего процесса и возвращает кортеж из файловых объектов, соответствующих его стандартным потокам ввода и вывода. Функция доступна, начиная с версии 2.0.

popen3(*cmd* [, *bufsize* [, *mode*]]) (UNIX, Windows)

Выполняет команду *cmd* в качестве дочернего процесса и возвращает кортеж из файловых объектов, соответствующих его стандартным потокам ввода, вывода и ошибок. Функция доступна, начиная с версии 2.0.

popen4(*cmd* [, *bufsize* [, *mode*]]) (UNIX, Windows)

Выполняет команду *cmd* в качестве дочернего процесса и возвращает кортеж из двух файловых объектов: соответствующий его стандартному потоку ввода и объединенному потоку вывода и ошибок. Функция доступна, начиная с версии 2.0.

21.1.3 Операции с файловыми дескрипторами

В этом разделе описаны операции низкого уровня над потоками с использованием файловых дескрипторов.

close(*fd*) (UNIX, Windows, Macintosh)

Закрывает файловый дескриптор *fd*.

dup(*fd*) (UNIX, Windows, Macintosh)
Возвращает дубликат файлового дескриптора *fd*.

dup2(*fd1*, *fd2*) (UNIX, Windows)
Копирует файловый дескриптор *fd1* в *fd2*, предварительно закрыв последний, если это необходимо.

fpathconf(*fd*, *name*) (UNIX)
Возвращает системную информацию, относящуюся к открытому файлу. Аргумент *name* должен быть строкой с именем системного значения или целым числом. Известные для данной ОС имена системных значений даны в словаре pathconf_names. Если имя *name* не известно, генерируется исключение ValueError. Если имя не поддерживается ОС (даже если оно включено в pathconf_names), генерируется исключение OSError с errno.EINVAL в качестве номера ошибки.

fstat(*fd*) (UNIX, Windows)
Возвращает информацию о файловом дескрипторе *fd* в виде, аналогичном тому, в котором возвращает результат функция stat().

fstatvfs(*fd*) (UNIX)
Возвращает информацию о файловой системе, на которой находится файл, ассоциированный с файловым дескриптором *fd*, в виде, аналогичном тому, в котором возвращает результат функция statvfs().

ftruncate(*fd*, *length*) (UNIX)
Укорачивает файл, соответствующий дескриптору *fd*, так, чтобы его длина была не более *length* байт.

isatty(*fd*) (UNIX)
Возвращает 1, если дескриптор файла *fd* открыт и связан с tty-подобным устройством, иначе возвращает 0.

lseek(*fd*, *offset*, *whence*) (UNIX, Windows, Macintosh)
Устанавливает текущую позицию в файле, соответствующего дескриптору *fd*. Аргумент *whence* указывает точку отсчета: 0 — начало файла, 1 — текущая позиция и 2 — конец файла.

open(*filename*, *flags* [, *mode*]) (UNIX, Windows, Macintosh)
Открывает файл с именем *filename*, устанавливает для него различные флаги в соответствии с аргументом *flags* и права на доступ в соответствии с аргументом *mode* (по умолчанию 0777) и текущим значением маски (umask). Возвращает дескриптор открытого файла. В качестве флагов могут быть использованы константы, описанные в конце раздела, объединенные оператором '|'.

openpty() (некоторые варианты UNIX)
Открывает новую пару псевдотерминалов и возвращает пару дескрипторов '(*master*, *slave*)' для pty и tty соответственно.

pipe() (UNIX, Windows)
Создает канал и возвращает пару дескрипторов '(*r*, *w*)', которые могут использоваться для чтения и записи соответственно.

read(*fd*, *n*) (UNIX, Windows, Macintosh)
Считывает не более *n* байт из файла, ассоциированного с дескриптором *fd*. Возвращает строку из считанных байтов.

tcgetpgrp(*fd*) (UNIX)
Возвращает идентификатор группы процессов, ассоциированной с терминалом, дескриптор которого использован в качестве аргумента *fd*.

tcsetpgrp(*fd*, *pg*) (UNIX)
Устанавливает идентификатор группы процессов, ассоциированной с терминалом, дескриптор которого использован в качестве аргумента *fd*, равным *pg*.

ttyname(*fd*) (UNIX)
Возвращает имя терминала, ассоциированного с дескриптором *fd*. Если *fd* не является дескриптором устройства tty, генерируется исключение.

write(*fd*, *str*) (UNIX, Windows, Macintosh)
Записывает строку *str* в файл, ассоциированный с дескриптором *fd*. Возвращает количество реально записанных байтов.

В качестве аргумента *flags* функции open() могут использоваться следующие константы, объединенные оператором '|':

O_RDONLY (UNIX, Windows, Macintosh)
O_WRONLY (UNIX, Windows, Macintosh)
O_RDWR (UNIX, Windows, Macintosh)
Определяют доступ к файлу: только для чтения, только для записи или для чтения и записи.

O_APPEND (UNIX, Windows, Macintosh)
Дописывать данные в конец файла.

O_CREAT (UNIX, Windows, Macintosh)
Создать файл, если он не существует.

O_EXCL (UNIX, Windows, Macintosh)
Открывать файл только, если он не существует.

O_TRUNC (UNIX, Windows, Macintosh)
Если файл существует, его содержимое будет удалено.

O_NDELAY (UNIX)
O_NONBLOCK (UNIX)
Открыть файл без блокировки.

O_DSYNC (UNIX)
O_RSYNC (UNIX)
O_SYNC (UNIX)
Открыть файл для синхронного ввода/вывода.

O_NOCTTY (UNIX)
: Если файл является устройством tty, он не станет терминалом, контролирующим текущий процесс.

O_BINARY (Windows, Macintosh)
O_TEXT (Windows, Macintosh)
: Открыть файл в двоичном или текстовом режиме.

21.1.4 Файлы и каталоги

access(*path*, *mode*) (UNIX, Windows)
: Проверяет доступ текущего процесса к файлу (каталогу) *path* на чтение, запись и/или выполнение. Аргумент *mode* должен быть равен `F_OK` для проверки существования файла или одна или несколько (объединенных оператором '`|`') констант из `R_OK`, `W_OK` и `X_OK` для проверки доступа. Возвращает `1`, если доступ разрешен, иначе возвращает `0`.

F_OK
: Используется в качестве аргумента *mode* функции `access()` для проверки наличия файла.

R_OK
: Используется в качестве аргумента *mode* функции `access()` для проверки доступа к файлу на чтение.

W_OK
: Используется в качестве аргумента *mode* функции `access()` для проверки доступа к файлу на запись.

X_OK
: Используется в качестве аргумента *mode* функции `access()` для проверки доступа к файлу на выполнение.

chdir(*path*) (UNIX, Windows, Macintosh)
: Изменяет текущий рабочий каталог на *path*.

getcwd() (UNIX, Windows, Macintosh)
: Возвращает строку, представляющую текущий рабочий каталог.

chmod(*path*, *mode*) (UNIX, Windows)
: Изменяет режим доступа к файлу (каталогу) на *mode* (целое число).

chown(*path*, *uid*, *gid*) (UNIX)
: Изменяет идентификаторы пользователя и группы пользователей файла (каталога) *path* на *uid* и *gid* соответственно.

link(*src*, *dst*) (UNIX)
: Создает жесткую ссылку с именем *dst*, указывающую на *src*.

listdir(*path*) (UNIX, Windows, Macintosh)

Возвращает список имен файлов и каталогов в каталоге *path* (используются пути относительно каталога *path*). Список не включает специальные имена (`curdir` и `pardir`), даже если они присутствуют.

lstat(*path*) (UNIX)

Работает аналогично функции `stat()`, но не следует символическим ссылкам.

mkfifo(*path* [, *mode*]) (UNIX)

Создает канал (FIFO) с именем *path* и режимом доступа *mode* (целое число, по умолчанию равен `0666`) с учетом текущего значения маски доступа (umask).

С именованными каналами работают как с обычными файлами. Созданный канал существует до тех пор, пока он не будет удален, например, с помощью функции `unlink()`. Обычно сервер открывает именованный канал для чтения и клиент — для записи.

mkdir(*path* [, *mode*]) (UNIX, Windows, Macintosh)

Создает каталог с именем *path* и режимом доступа *mode* (целое число, по умолчанию используется `0777`) с учетом маски доступа (umask).

makedirs(*path* [, *mode*])

Функция для рекурсивного создания каталогов. Работает аналогично функции `mkdir()`, но автоматически создает все необходимые промежуточные каталоги. Генерирует исключение `IOError`, если каталог *path* уже присутствует или не может быть создан.

pathconf(*path*, *name*) (UNIX)

Возвращает системную информацию, относящуюся к файлу с именем *path*. Аргумент *name* должен быть строкой с именем системного значения или целым числом. Известные для данной ОС имена системных значений даны в словаре `pathconf_names`. Если имя *name* неизвестно, генерируется исключение `ValueError`. Если имя не поддерживается ОС (даже если оно включено в `pathconf_names`), генерируется исключение `OSError` с `errno.EINVAL` в качестве номера ошибки.

pathconf_names (UNIX)

Словарь, отображающий имена, воспринимаемые функциями `pathconf()` и `fpathconf()`, к целым значениям для этих имен.

readlink(*path*) (UNIX)

Возвращает строку с именем файла (каталога), на который указывает символическая ссылка *path*.

remove(*path*) (UNIX, Windows, Macintosh)
unlink(*path*) (UNIX, Windows, Macintosh)

Удаляют файл с именем *path*. Для удаления каталогов воспользуйтесь функцией `rmdir()`.

removedirs(*path*)

Функция для рекурсивного удаления каталогов. Работает аналогично функции

rmdir(), но также рекурсивно удаляет и пустые родительские каталоги (если это возможно). Генерирует исключение IOError, если каталог *path* не может быть удален.

rename(*src, dst*) (UNIX, Windows, Macintosh)
Переименовывает файл (каталог) *src* в *dst*.

renames(*old, new*)
Функция для рекурсивного переименования. Работает аналогично функции rename(), но сначала пытается создать родительские каталоги, необходимые для создания *new*. После переименования *old* в *new* удаляет правую часть пути *old* с помощью функции removedirs().

rmdir(*path*) (UNIX, Windows, Macintosh)
Удаляет каталог с именем *path*.

stat(*path*) (UNIX, Windows, Macintosh)
Возвращает информацию о файле (каталоге) *path* в виде кортежа, содержащего как минимум 10 наиболее важных (и переносимых) целочисленных элементов в следующем порядке: *st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*. В стандартном модуле stat определены функции и константы, которые будут полезны для обработки результата.

Под ОС Windows некоторые элементы будут иметь произвольные значения.

statvfs(*path*) (UNIX)
Возвращает информацию о файловой системе, на которую ссылается путь *path*, в виде кортежа из 10 наиболее общих целочисленных элементов в следующем порядке: f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax. В стандартном модуле statvfs определены функции и константы, которые будут полезны для обработки результата.

symlink(*src, dst*) (UNIX)
Создает символическую ссылку с именем *dst*, указывающую на *src*.

tempnam([*dir* [, *prefix*]]) (UNIX)
Возвращает уникальное имя, подходящее для создания временного файла в каталоге *dir*. Если аргумент *dir* опущен или равен None, используется общепринятое место для временных файлов. Если задан и не равен None аргумент *prefix* (строка), он используется в качестве приставки к имени временного файла. Функция доступна, начиная с версии 1.6.

tmpnam() (UNIX)
Возвращает уникальное имя, подходящее для создания временного файла. Используется общепринятое место для временных файлов. Функция доступна, начиная с версии 1.6.

TMP_MAX
Максимальное количество уникальных имен, которые может сгенерировать функция tmpnam() перед тем, как использовать их снова. Эта константа доступна, начиная с версии 1.6.

utime(*path*, *times*) (UNIX, Windows, Macintosh)
Устанавливает время последнего доступа и последнего внесения изменений файла *path*. Начиная с версии 2.0, в качестве аргумента *times* Вы можете использовать `None` для того, чтобы установить текущее время. Иначе *times* должен быть кортежем вида '`(`*atime*`,` *mtime*`)`', где *atime* — время последнего доступа и *mtime* — время последнего внесения изменений.

21.1.5 Управление процессами

Функции, описанные в этом разделе, предназначены для создания и управления процессами.

Функциям с именами вида `exec*()` передается список аргументов запускаемой программы. Во всех случаях первый из них передается как собственное имя программы, а не как аргумент командной строки, то есть `sys.argv[0]`, если Вы запускаете программу на языке Python. Например, '`os.execv('/bin/echo', ['foo', 'bar'])`' выведет слово "bar" на стандартный поток вывода, в то время как строка `'foo'` просто игнорируется.

abort`()` (UNIX, Windows)
Посылает сигнал `SIGABRT` текущему процессу. По умолчанию в UNIX сигнал `SIGABRT` приводит к прерыванию процесса и сохранению дампа памяти ('core'), под Windows процесс прерывается с кодом завершения, равным 3. Программа может зарегистрировать другой обработчик сигнала с помощью функции `signal.signal()`.

execl(*path* [, *arg1* ...]) (UNIX, Windows)
Эквивалентно вызову '`execv(`*path*`, (`*arg1*`, ...))`'.

execle(*path* [, *arg1* ...], *env*) (UNIX, Windows)
Эквивалентно вызову '`execve(`*path*`, (`*arg1*`, ...), `*env*`)`'.

execlp(*path* [, *arg1* ...]) (UNIX, Windows)
Эквивалентно вызову '`execvp(`*path*`, (`*arg1*`, ...))`'.

execv(*path*, *args*) (UNIX, Windows)
Выполняет файл *path* со списком аргументов *args* (кортеж или список), заменяя текущий процесс (то есть интерпретатор).

execve(*path*, *args*, *env*) (UNIX, Windows)
Выполняет файл *path* со списком аргументов *args* (кортеж или список) и окружением *env* (словарь, отображающий имена переменных окружения к их значениям), заменяя текущий процесс (то есть интерпретатор).

execvp(*path*, *args*) (UNIX, Windows)
Ищет в путях (`environ['PATH']`) файл *path* и выполняет его аналогично функции `execv()`.

execvpe(*path*, *args*, *env*) (UNIX, Windows)
Ищет в путях (*env*['PATH']) файл *path* и выполняет его аналогично функции execve().

_exit(*code*) (UNIX, Windows)
Прерывает выполнение программы с кодом завершения *code* без выполнения каких-либо дополнительных действий. Нормальным способом прерывания выполнения программы является вызов sys.exit(*n*). Функцию _exit() следует использовать только в дочернем процессе после вызова fork().

fork() (UNIX)
Создает ответвление от текущего процесса. Возвращает 0 в дочернем процессе и идентификатор дочернего процесса в родительском.

forkpty() (некоторые варианты UNIX)
Создает ответвление от текущего процесса, используя новый псевдотерминал в качестве терминала, контролирующего дочерний процесс. Возвращает пару '(*pid*, *fd*)', где *pid* равен 0 в дочернем процессе и идентификатору дочернего процесса в родительском и fd является файловым дескриптором псевдотерминала. Для большей переносимости используйте модуль pty.

kill(*pid*, *sig*) (UNIX)
Посылает сигнал *sig* процессу с идентификатором *pid*. Смотрите также описание модуля signal.

nice(*increment*) (UNIX)
Добавляет значение *increment* к текущему приоритету процесса, возвращает новый приоритет.

plock(*op*) (UNIX)
Блокирует в памяти сегменты программы. Эта функция доступна, начиная с версии 2.0.

spawnv(*mode*, *path*, *args*) (UNIX, Windows)
Выполняет отдельным процессом файл *path* с аргументами *args* (кортеж или список). Аргумент *mode* может иметь значения, описанные ниже. На платформах UNIX эта функция доступна, начиная с версии 1.6.

spawnve(*mode*, *path*, *args*, *env*) (UNIX, Windows)
Выполняет отдельным процессом файл *path* с аргументами *args* (кортеж или список) и используя *env* в качестве окружения. Аргумент *mode* может иметь значения, описанные ниже. На платформах UNIX эта функция доступна, начиная с версии 1.6.

P_WAIT (UNIX, Windows)
P_NOWAIT (UNIX, Windows)
P_NOWAITO (UNIX, Windows)
P_OVERLAY (Windows)
P_DETACH (Windows)
Возможные значения аргумента *mode* функций spawnv() и spawnve().

startfile(*path*) (Windows)

Открывает файл, используя ассоциированную с расширением (или типом) *path* программу, аналогично команде `start` стандартного командного интерпретатора Windows (или двойному щелчку мыши в Explorer). Функция заканчивает свое выполнение сразу же после запуска. Не существует способа дождаться завершения выполнения программы или определения ее кода завершения. Используйте функцию `os.path.normpath()` для приведения пути к правильному виду. Функция `startfile()` доступна, начиная с версии 2.0.

system(*command*) (UNIX, Windows)

Выполняет в стандартном для данной платформы командном интерпретаторе команду *command* (строка). Возвращает код завершения в формате, аналогичном формату результата функции `wait()`, за исключением Windows 95 и 98 — в этих ОС всегда возвращает `0`.

times() (UNIX, Windows)

Возвращает кортеж из вещественных чисел, указывающих суммарное затраченное время пользователем, системой, пользователем в дочерних процессах, системой в дочерних процессах и реальное время работы с определенного момента в прошлом.

wait() (UNIX)

Ждет завершения дочернего процесса и возвращает кортеж вида '(*pid*, *code*)', где *pid* — идентификатор дочернего процесса и *code* — код, младший байт которого содержит номер сигнала, завершившего выполнение процесса (старший бит установлен, если был записан дамп памяти), а старший байт — код завершения (если номер сигнала равен `0`).

waitpid(*pid*, *options*) (UNIX)

Ждет завершения дочернего процесса, заданного аргументом *pid*, и возвращает кортеж вида '(*pid*, *code*)' (см. описание функции `wait()`). Точная семантика функции зависит от значения аргумента *options*, который должен быть равен `0` для обычных операций.

Если аргумент *pid* больше `0`, `waitpid()` запрашивает информацию для процесса с идентификатором *pid*. Если аргумент *pid* равен `0`, запрашивается информация для любого дочернего процесса из той же группы, что и текущий процесс. Если аргумент *pid* равен `-1`, запрашивается информация для любого дочернего процесса. Если аргумент *pid* меньше `-1`, запрашивается информация для любого процесса в группе с идентификатором -*pid* (абсолютное значение аргумента *pid*).

WNOHANG (UNIX)

При использовании этой опция функция `waitpid()` не ожидает завершения процесса, а возвращает информацию только, если она сразу доступна.

Следующие функции позволяют извлечь информацию из закодированного кода завершения, возвращаемого функциями `system()`, `wait()` и `waitpid()`:

WIFSTOPPED(*code*) (UNIX)

Возвращает истину, если процесс был остановлен.

WIFSIGNALED(*code*) (UNIX)
Возвращает истину, если процесс был завершен сигналом.

WIFEXITED(*code*) (UNIX)
Возвращает истину, если процесс был завершен системным вызовом функции exit().

WEXITSTATUS(*code*) (UNIX)
Если выражение WIFEXITED(*code*) верно, возвращает целочисленный параметр, использованный при системном вызове функции exit(), в противном случае возвращаемое значение не несет какого-либо смысла.

WSTOPSIG(*code*) (UNIX)
Возвращает номер сигнала, который привел к остановке процесса.

WTERMSIG(*code*) (UNIX)
Возвращает номер сигнала, который привел к завершению процесса.

21.1.6 Различная системная информация

confstr(*name*) (UNIX)
Возвращает строку, содержащую системную конфигурационную информацию. Аргумент *name* должен быть строкой с именем системного значения или целым числом. Известные для данной ОС имена системных значений даны в словаре confstr_names. Если системное значение *name* не задано, возвращает пустую строку. Если имя *name* неизвестно, генерируется исключение ValueError. Если имя не поддерживается ОС (даже если оно включено в confstr_names), генерируется исключение OSError с errno.EINVAL в качестве номера ошибки.

confstr_names (UNIX)
Словарь, отображающий имена, воспринимаемые функцией confstr(), к целым значениям для этих имен.

sysconf(*name*) (UNIX)
Возвращает целое системное конфигурационное значение. Аргумент *name* должен быть строкой с именем системного значения или целым числом. Известные для данной ОС имена системных значений даны в словаре sysconf_names. Если системное значение *name* не задано, возвращает -1. Если имя *name* неизвестно, генерируется исключение ValueError. Если имя не поддерживается ОС (даже если оно включено в sysconf_names), генерируется исключение OSError с errno.EINVAL в качестве номера ошибки.

sysconf_names (UNIX)
Словарь, отображающий имена, воспринимаемые функцией sysconf(), к целым значениям для этих имен.

Следующие константы используются при работе с именами путей. Операции высокого уровня над именами путей определены в модуле os.path.

curdir
Строка, используемая для ссылки на текущий каталог, например '.' или ':'.

pardir
Строка, используемая для ссылки на родительский каталог, например '..' или '::'.

sep
Символ, который используется ОС для разделения компонент пути, например, '/', '\' или ':'.

altsep
Альтернативный символ, используемый ОС для разделения компонент пути, или None, если имеется только один символ. В DOS и Windows эта константа имеет значение '/'.

pathsep
Символ, который обычно используется ОС для разделения имен путей в списке (как в переменной окружения PATH), например, ':' или ';'.

defpath
Путь поиска исполняемых файлов по умолчанию, если не установлена переменная окружения PATH.

linesep
Последовательность символов, которая используется для завершения строк на данной платформе, например, '\n', '\r' или '\r\n'.

21.2 os.path — работа с именами путей

Этот модуль определяет полезные операции над именами путей.

abspath(*path*)
Возвращает нормализованную абсолютную версию пути *path*. На большинстве платформ вызов этой функции эквивалентен вызову 'normpath(join(os.getcwd(), *path*))'.

basename(*path*)
Возвращает основное имя (basename) пути *path*. Это второй элемент пары split(*path*). Обратите внимание, что поведение этой функции отличается от поведения программы **basename**, которая для '/foo/bar/' выводит 'bar', в то время как функция basename() возвращает пустую строку.

commonprefix(*list*)
Возвращает наибольшую общую часть путей в списке *list* (произвольная последовательность), полученную посимвольным сравнением (то есть результат может и не быть корректным путем). Если список *list* пуст, возвращает пустую строку ('').

dirname(*path*)
Возвращает имя каталога в пути *path*. Это первый элемент пары split(*path*).

exists(*path*)
Возвращает 1, если путь с именем *path* (файл или каталог) существует, иначе возвращает 0.

expanduser(*path*)
Возвращает аргумент с компонентом вида '~' или '~*user*', замененным домашним каталогом пользователя. '~' заменяется значением переменной окружения HOME, поиск имени домашнего каталога для пользователя *user* производится с помощью модуля pwd. Если замена по каким-либо причинам не может быть произведена или путь не начинается с символа '~', возвращает *path* без изменений.

expandvars(*path*)
Возвращает аргумент с компонентами вида '$*name*' и '${*name*}', замененными значением переменной окружения *name*. Некорректные имена и имена, ссылающиеся на неопределенные переменные, оставляются в строке *path* без изменений. При использовании ОС Macintosh всегда возвращает *path* без изменений.

getatime(*path*)
Возвращает время (число секунд с начала эпохи, см. описание модуля time) последнего доступа к файлу (каталогу) *path*. Генерирует исключение IOError, если файл не существует или не доступен.

getmtime(*path*)
Возвращает время (число секунд с начала эпохи, см. описание модуля time) последнего внесения изменений в файл (каталог) *path*. Генерирует исключение IOError, если файл не существует или не доступен.

getsize(*path*)
Возвращает размер в байтах файла (каталога) *path*. Генерирует исключение IOError, если файл не существует или не доступен.

isabs(*path*)
Возвращает 1, если путь *path* является абсолютным.

isfile(*path*)
Возвращает 1, если путь *path* указывает на существующий обычный файл, иначе возвращает 0. Эта функция следует символическим ссылкам, то есть функции islink() и isfile() могут обе возвращать истину для одного и того же пути.

isdir(*path*)
Возвращает 1, если путь *path* указывает на существующий каталог, иначе возвращает 0. Эта функция следует символическим ссылкам, то есть функции islink() и isdir() могут обе возвращать истину для одного и того же пути.

islink(*path*)
Возвращает 1, если путь *path* является символической ссылкой, иначе возвращает 0. Всегда возвращает 0, если символические ссылки не поддерживаются.

ismount(*path*)
Возвращает 1, если путь *path* является точкой монтирования, то есть каталог *path* и родительский для него каталог находятся на разных устройствах (разделах) или ссылаются на один и тот же узел (i-node) одного устройства.

join(*path1* [, *path2* ...])
Объединяет компоненты пути. Если какой-либо компонент является абсолютным путем, все предыдущие компоненты отбрасываются и объединение продолжается. Возвращаемое значение является объединением всех непустых строк *path1*, *path2* и т. д., в качестве разделителя используется `os.sep`. Если последний аргумент является пустой строкой, результат заканчивается символом `os.sep`.

normcase(*path*)
Возвращает *path* с нормализованным регистром букв. На платформах UNIX возвращает аргумент без изменений. Для файловых систем, нечувствительных к регистру букв, преобразует все буквы к нижнему регистру. Под Windows также заменяет символы косой черты (`os.altsep`) на символы обратной косой черты (`os.sep`).

normpath(*path*)
Возвращает нормализованное имя пути *path*. Эта функция сворачивает излишние разделители и ссылки на текущий и родительский каталоги, то есть имена путей `'A//B'`, `'A/./B'` и `'A/foo/../B'` преобразуются к `'A/B'`. Под Windows также заменяет символы косой черты (`os.altsep`) на символы обратной косой черты (`os.sep`). Функция `normpath()` не нормализует регистр букв — для этих целей используйте функцию `normcase()`.

samefile(*path1*, *path2*) (UNIX, Macintosh)
Возвращает 1, если оба пути ссылаются на один и тот же файл (каталог), иначе возвращает 0. Генерирует исключение `IOError`, если вызов `os.stat()` для какого-либо пути заканчивается ошибкой.

sameopenfile(*file1*, *file2*) (UNIX, Macintosh)
Возвращает 1, если файловые объекты *file1* и *file2* ссылаются на один и тот же файл (файловые объекты могут представлять разные файловые дескрипторы), иначе возвращает 0.

samestat(*stat1*, *stat2*) (UNIX, Macintosh)
Возвращает 1, если кортежи *stat1* и *stat2* (результат, возвращаемый функциями `fstat()`, `lstat()` и `stat()`) ссылаются на один и тот же файл.

split(*path*)
Разбивает имя пути *path* на пару строк '(*head*, *tail*)', где *tail* — последний компонент пути и *head* — остальная часть пути. *tail* никогда не содержит символа `os.sep`: если путь *path* заканчивается этим символом, *tail* будет пустой строкой. Завершающие символы `os.sep` в *head* отбрасываются, за исключением случаев, когда *path* ссылается на корневой каталог. В любом случае 'join(*head*, *tail*)' дает путь, эквивалентный *path*.

splitdrive(*path*)
Разбивает имя пути *path* на пару строк '(*drive*, *tail*)', где *drive* — спецификатор логического диска или пустая строка и *head* — остальная часть пути. В системах, которые не используют спецификаторы логических дисков *drive* всегда является пустой строкой. В любом случае выражение '*drive* + *tail* == *path*' всегда верно.

splitext(*path*)
Разбивает имя пути *path* на пару строк '(*root*, *ext*)' так, чтобы выполнялось условие '*root* + *ext* == *path*' и строка *ext* была пустой или начиналась с точки и содержала не более одной точки.

walk(*path*, *visit*, *arg*)
Вызывает '*visit*(*arg*, *dirname*, *names*)' для каждого каталога в дереве, начиная с *path* (включая каталог *path*). Аргумент *dirname* указывает имя каталога, для которого вызывается функция *visit*, а *names* является списком файлов и каталогов в каталоге *dirname* (os.listdir(*dirname*), то есть используется путь относительно каталога *dirname*). Функция *visit* может вносить изменения в *names* и, тем самым, оказывать влияние на то, какие каталоги будут посещаться далее по дереву, то есть чтобы избежать посещения каких-либо частей дерева каталогов.

21.3 stat — интерпретация os.stat()

Этот модуль определяет константы и функции, необходимые для интерпретации элементов кортежа '(*st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime* ...)', возвращаемого функциями os.stat(), os.lstat() и os.fstat().

S_ISDIR(*st_mode*)
Истина, если аргумент *st_mode* получен для каталога.

S_ISCHR(*st_mode*)
Истина, если аргумент *st_mode* получен для специального файла, представляющего устройство посимвольного ввода/вывода.

S_ISBLK(*st_mode*)
Истина, если аргумент *st_mode* получен для специального файла, представляющего блочное устройство.

S_ISREG(*st_mode*)
Истина, если аргумент *st_mode* получен для обычного файла.

S_ISFIFO(*st_mode*)
Истина, если аргумент *st_mode* получен для именованного канала (FIFO).

S_ISLNK(*st_mode*)
Истина, если аргумент *st_mode* получен для символической ссылки.

S_ISSOCK(*st_mode*)
Истина, если аргумент *st_mode* получен для канала сетевого соединения (socket).

S_IMODE(*st_mode*)
Возвращает часть состояния, которую можно установить с помощью функции `os.chmod()` — доступ, sticky bit, set-group-id и set-user-id.

S_IFMT(*st_mode*)
Возвращает часть состояния, описывающую тип файла (используется функциями `S_IS*()`, описанными выше).

Обычно гораздо удобнее воспользоваться функциями `os.path.is*()` (см. описание модуля `os.path`) для определения типа файла. Описанные в этом разделе функции полезны в тех случаях, когда необходимо выполнить несколько проверок для одного и того же файла, и Вы хотите избежать накладных расходов на вызов системной функции `stat()` для каждой проверки.

Следующие переменные являются символическими индексами для кортежа, возвращаемого функциями `os.stat()`, `os.fstat()` и `os.lstat()`.

ST_MODE
Режим защиты файла.

ST_INO
Внутренний номер файла (i-node).

ST_DEV
Номер устройства, на котором файл расположен.

ST_NLINK
Количество (жестких) ссылок на файл.

ST_UID
Идентификатор пользователя-владельца.

ST_GID
Идентификатор группы-владельца.

ST_SIZE
Размер файла в байтах.

ST_ATIME
Время последнего доступа.

ST_MTIME
Время последнего внесения изменений.

ST_CTIME
Время последнего изменения статуса.

Приведем простой пример:

```
import os, sys
from stat import *

def visit(arg, dir, names):
    for name in names:
        fullname = os.path.join(dir, name)
        mode = os.stat(fullname)[ST_MODE]
        if S_ISDIR(mode):
            print 'Каталог', fullname
        elif S_ISREG(mode):
            print 'Обычный файл', fullname
        else:
            print 'Файл неизвестного типа', fullname

if __name__ == '__main__':
    os.path.walk(sys.argv[1], visit, None)
```

21.4 statvfs — интерпретация os.statvfs()

Каждая из констант, определенных в этом модуле является индексом элемента кортежа, возвращаемого функцией os.statvfs(), содержащего определенную информацию. Использование этих констант позволяет избежать необходимости помнить "магические индексы".

F_BSIZE
Предпочтительный размер блоков для данной файловой системы.

F_FRSIZE
Основной размер блоков для данной файловой системы.

F_BLOCKS
Общее число блоков.

F_BFREE
Число свободных блоков.

F_BAVAIL
Число свободных блоков, доступных рядовому пользователю.

F_FILES
Общее число файловых узлов (то есть максимальное возможное число файлов).

F_FFREE
Число свободных файловых узлов.

F_FAVAIL
Число файловых узлов, доступных рядовому пользователю.

F_FLAG
Флаги, их значение зависит от используемой системы.

F_NAMEMAX
Максимальная длина имен файлов.

21.5 `filecmp` — сравнение файлов и каталогов

Модуль `filecmp` предоставляет функции для быстрого и полного сравнения файлов и каталогов.

cmp(*f1, f2* [, *shallow*])
Сравнивает файлы с именами *f1* и *f2*[1]. Возвращает `1`, если они одинаковые, иначе возвращает `0`. По умолчанию производится поверхностное сравнение, используя информацию, возвращаемую функцией `os.stat()`. Однако, если задан и является ложью аргумент *shallow*, при необходимости будет производиться полное сравнение содержимого файлов.

Результаты сравнений содержимого файлов сохраняются во внутреннем кэше и используются при повторном сравнении, если информация для этих файлов, возвращаемая функцией `os.stat()` не изменилась.

cmpfiles(*dir1, dir2, common* [, *shallow*])
Сравнивает файлы, имена которых перечислены в *common* (произвольная последовательность), в каталогах *dir1* и *dir2*[1]. Возвращает кортеж из трех списков имен '(*match, mismatch, errors*)'. Список *match* содержит имена файлов, одинаковых в обоих каталогах, список *mismatch* — разных и *errros* содержит имена файлов, которые по каким-либо причинам не удается сравнить (например, к ним нет доступа). Аргумент *shallow* имеет такое же значение, как и в функции `cmp()`.

Кроме того, модуль определяет класс, позволяющий сравнивать содержимое каталогов (включая содержимое вложенных каталогов):

dircmp(*dir1, dir2* [, *ignore*])
Конструирует новый объект, предназначенный для сравнения каталогов *dir1* и *dir2*. Имена, перечисленные в списке *ignore* (по умолчанию равен '`['RCS', 'CVS', 'tags']`'), игнорируются.

[1] На самом деле функции `cmp()` и `cmpfiles()` имеет еще один необязательный аргумент. Мы намеренно его не документируем, так как его использование не переносимо, а иногда и опасно.

Экземпляры класса dircmp имеют следующие методы и атрибуты данных:

report()
Выводит на стандартный поток вывода (sys.stdout) подробный отчет о сравнении каталогов.

report_partial_closure()
Выводит на стандартный поток вывода (sys.stdout) подробный отчет о сравнении каталогов и одного уровня вложенных каталогов.

report_full_closure()
Выводит на стандартный поток вывода (sys.stdout) подробный отчет о сравнении каталогов и рекурсивно всех вложенных каталогов.

left_list
Список имен файлов и каталогов в *dir1*, не указанных в аргументе *ignore*.

right_list
Список имен файлов и каталогов в *dir2*, не указанных в аргументе *ignore*.

common
Список имен файлов и каталогов, которые присутствуют как в *dir1*, так и в *dir2*.

left_only
Список имен файлов и каталогов, которые присутствуют только в *dir1*.

right_only
Список имен файлов и каталогов, которые присутствуют только в *dir2*.

common_dirs
Список имен каталогов, которые присутствуют как в *dir1*, так и в *dir2*.

common_files
Список имен файлов, которые присутствуют как в *dir1*, так и в *dir2*.

common_funny
Список имен, найденных в обоих каталогах, но имеющих разный тип или дающих ошибку при вызове функции os.stat().

same_files
Список файлов, одинаковых в обоих каталогах (сравнивается только информация, возвращаемая функцией os.stat()).

diff_files
Список файлов, присутствующих в обоих каталогах, содержимое которых различается.

funny_files
Список файлов, присутствующих в обоих каталогах, которые по каким-либо причинам не удается сравнить (например, к ним нет доступа).

subdirs
Словарь, отображающий имена каталогов из списка `common_dirs` в экземпляры класса `dircmp` для этих каталогов.

Заметим, что значения атрибутов вычисляются по мере необходимости, а не сразу при конструировании объекта. Это позволяет избежать излишних расходов при запросе значений только тех атрибутов, которые можно быстро вычислить. Однажды вычисленное значение сохраняется и используется при дальнейших запросах.

21.6 `popen2` — доступ к потокам ввода/вывода дочерних процессов

Модуль доступен в операционных системах UNIX и Windows (начиная с версии 2.0).

Модуль `popen2` позволяет порождать подпроцессы и иметь доступ к их стандартным потокам ввода, вывода и ошибок. Во всех случаях команда выполняется в стандартном для данной платформы интерпретаторе команд (устанавливается переменной окружения SHELL в UNIX и COMSPEC в Windows). Заметим, что, начиная с версии 2.0, функциональность этого модуля доступна через одноименные функции модуля `os`, имеющие аналогичный интерфейс, но возвращающие кортеж с другим порядком следования элементов.

Основной интерфейс модуля представлен тремя функциями. Для всех функций аргумент *bufsize* определяет размер буфера каналов и *mode* (начиная с версии 2.0) — режим, в котором будут открыты каналы (`'t'` — текстовый, используется по умолчанию, или `'b'` — двоичный).

popen2(*cmd* [, *bufsize* [, *mode*]])
Выполняет команду *cmd* в качестве дочернего процесса и возвращает для него кортеж из файловых объектов, представляющих стандартные потоки вывода и ввода.

popen3(*cmd* [, *bufsize* [, *mode*]])
Выполняет команду *cmd* в качестве дочернего процесса и возвращает для него кортеж из файловых объектов, представляющих стандартные потоки вывода, ввода и ошибок.

popen4(*cmd* [, *bufsize* [, *mode*]])
Выполняет команду *cmd* в качестве дочернего процесса и возвращает кортеж из двух файловых объектов: соответствующий его объединенному потоку вывода и ошибок и стандартному потоку ввода. Функция доступна, начиная с версии 2.0.

Кроме того, модуль определяет следующие классы (на платформе UNIX они используются для реализации описанных выше функций):

Popen3(*cmd* [, *capturestderr* [, *bufsize*]]) (UNIX)
Конструирует и возвращает объект, представляющий дочерний процесс. Если задан и является истиной аргумент *capturestderr*, возвращаемый объект будет перехватывать стандартный поток ошибок дочернего процесса. Аргумент *bufsize* указывает размер буфера потоков.

Popen4(*cmd* [, *bufsize*]) (UNIX)
Этот класс аналогичен классу `Popen3`, но всегда направляет стандартные потоки вывода и ошибок дочернего процесса в один файловый объект.

Экземпляры классов `Popen3` и `Popen4` имеют следующие методы и атрибуты данных:

poll()
Возвращает `-1`, если выполнение дочернего процесса еще не закончилось, в противном случае возвращает код завершения.

wait()
Ждет окончания выполнения дочернего процесса и возвращает код завершения.

fromchild
Файловый объект, представляющий стандартный поток вывода дочернего процесса. Для экземпляров класса `Popen4` этот объект представляет объединение стандартных потоков вывода и ошибок дочернего процесса.

tochild
Файловый объект, представляющий стандартный поток ввода дочернего процесса.

childerr
Файловый объект, представляющий стандартный поток ошибок дочернего процесса, если аргумент *capturestderr* конструктора являлся истиной, иначе `None`. Для экземпляров класса `Popen4` всегда равен `None`.

pid
Идентификатор дочернего процесса.

21.7 `time` — определение и обработка времени

Модуль `time` предоставляет различные функции для работы со временем. Этот модуль всегда доступен, но не все функции определены для всех платформ.

Немного о терминологии и принятых соглашениях.

- Отсчет времени начинается с *начала эпохи*, 0 часов 1 января определенного года. Для платформ UNIX это 1970 год. Чтобы определить начало эпохи для Вашей платформы, взгляните на `gmtime(0)`.

- Функции в этом модуле не могут работать с временем до начала эпохи и после некоторого момента в будущем. Последняя точка определяется библиотекой языка C, лежащей в основе реализации. Для платформы UNIX это обычно 2038 год.

- Реализация библиотек Python зависит от библиотеки языка C для данной платформы, которая обычно не имеет проблем с 2000 годом, так как время представляется числом секунд, прошедших с начала эпохи. Функции, работающие с представлением времени в виде кортежа, как правило, требуют все четыре цифры года. Для совместимости с предыдущими версиями допускается использование двух цифр, если переменная `accept2dyear` не равна нулю. По умолчанию эта переменная равна `1`, однако, если переменная окружения PYTHONY2K установлена и равна непустой строке, `accept2dyear` инициализируется нулем. Таким образом, Вы можете запретить использование только двух цифр, установив переменную окружения PYTHONY2K равной непустой строке.

  Если представление года двумя цифрами воспринимается, оно преобразуется в соответствии со стандартами POSIX и X/Open: значения 69–99 преобразуются к 1969–1999 и 0–68 к 2000–2068. Значения 100–1899 в любом случае запрещены.

- UTC — Universal Coordinated Time, универсальное синхронизированное время, известное также как GMT (Greenwich Mean Time, среднее время по Гринвичу).

- DST — Daylight Saving Time, летнее время — корректировка часового пояса (обычно) на один час в течение части года. Правила перехода на летнее время определяются местными законами и могут меняться от года к году. Библиотека языка C имеет таблицу с локальными правилами и является в этом отношении единственным верным источником.

- Точность различных функций, работающих в реальном времени, может быть меньше, чем Вы могли бы предположить исходя из единиц, с которыми они работают.

  С другой стороны, точность функций `time()` и `sleep()` выше, чем точность одноименных функций библиотеки C на платформе UNIX: время в них выражается вещественным числом, функция `time()` наиболее точное время (используя функцию `gettimeofday()` языка C, если она доступна), функция `sleep()` воспринимает время с ненулевой дробной частью (используя функцию `select()` языка C, если она доступна).

- Представление времени в виде кортежа, возвращаемое функциями `gmtime()`, `localtime()` и `strptime()` и воспринимаемое функциями `asctime()`, `mktime()` и `strftime()`, содержит 9 целочисленных элементов:

Индекс	**Назначение поля**	**Диапазон значений**
0	год	например 1993, смотрите правила обработки выше
1	месяц	1–12
2	день (число)	1–31
3	час	0–23
4	минута	0–59

Индекс	Назначение поля	Диапазон значений
5	секунда	0–61; значения 60 и 61 используются изредка для согласования с солнечным календарем
6	день недели	0–6; 0 соответствует понедельнику
7	юлианское представление даты (числом дней от начала года)	1–366
8	флаг коррекции летнего времени	0, 1 или -1; смотрите ниже

Обратите внимание, что в отличие от аналогичной структуры в языке C, значения месяцев находится в диапазоне 1–12, а не 0–11. Значение года обрабатывается так, как описано выше. Флаг коррекции для летнего времени равный `-1`, как правило, приводит к автоматическому выбору правильного состояния.

Модуль определяет следующие функции и объекты данных:

accept2dyear

Если эта переменная не равна нулю, допускается использование представления года двумя цифрами. По умолчанию эта переменная равна `1`, однако, если переменная окружения PYTHONY2K установлена и равна непустой строке, `accept2dyear` инициализируется нулем. Таким образом, Вы можете запретить использование только двух цифр, установив переменную окружения PYTHONY2K равной непустой строке. Вы можете также менять значение `accept2dyear` во время выполнения программы.

altzone()

Сдвиг локального часового пояса с учетом перехода на летнее время в секундах на запад от UTC. Это значение всегда отрицательно, если часовой пояс находится восточнее UTC (в том числе и Россия). Используйте эту функцию, только если значение переменной `daylight` не равно нулю.

asctime(*time_tuple*)

Преобразует кортеж *time_tuple*, представляющий время (возвращается функциями `gmtime()`, `localtime()` и `strptime()`), в строку из 24 символов вида `'Sun Jun 20 23:21:05 1993'`. Обратите внимание, что в отличие от одноименной функции языка C, строка не завершается символом перехода на новую строку.

clock()

Возвращает текущее процессорное время в секундах в виде вещественного числа. Точность возвращаемого времени зависит от точности одноименной функции библиотеки языка C. Функция `clock()` может быть использована, например, для замера производительности.

ctime(*secs*)

Преобразует время *secs*, выраженное в секундах с начала эпохи в строку, представляющую локальное время. Эквивалентно `asctime(localtime(`*secs*`))`.

daylight

Эта переменная имеет ненулевое значение, если определен часовой пояс с переходом на летнее время (DST).

gmtime(*secs*)

Преобразует время *secs*, выраженное в секундах с начала эпохи в кортеж, представляющий универсальное время (UTC).

localtime(*secs*)

Преобразует время *secs*, выраженное в секундах с начала эпохи в кортеж, представляющий локальное время. Флаг коррекции времени равен `1` (последний элемент), если применяется переход на летнее время (DST).

mktime(*time_tuple*)

Эта функция выполняет преобразование, обратное функции `localtime()`. Аргумент *time_tuple* должен быть кортежем из 9 элементов, представляющим *локальное* время. Для совместимости с функцией `time()` возвращает вещественное число. Если *time_tuple* не представляет корректного времени, генерирует исключение `OverflowError`.

sleep(*secs*)

Приостанавливает выполнение на *secs* секунд. Аргумент *secs* может быть вещественным числом для указания более точного времени. Реальное время, на которое приостанавливается выполнение программы, может быть меньше *secs*, так как любой перехваченный сигнал прерывает выполнение функции `sleep()`, после чего выполняется обработчик сигнала. Кроме того, время может быть и больше *secs* из-за планирования активности различных задач системы.

strftime(*format*, *time_tuple*)

Возвращает время, выраженное кортежем *time_tuple*, в виде строки в соответствии с форматом *format*. Строка формата помимо обычных символов может содержать следующие управляющие последовательности (названия месяцев, дней недели и т. п. зависят от текущих национальных установок — см. описание модуля `locale`):

Последовательность	Назначение
`%a`	Сокращенное название дня недели.
`%A`	Полное название дня недели.
`%b`	Сокращенное название месяца.
`%B`	Полное название месяца.
`%c`	Общепринятое (в соответствии с текущими национальными установками) представление даты и времени.
`%d`	Десятичное представление числа (дня месяца), `'01'`–`'31'`.
`%H`	Десятичное представление часа, `'00'`–`'23'`.
`%I`	Десятичное представление часа, `'01'`–`'12'`.

Последовательность	Назначение
%j	Десятичное представление дня года, '001'–'366'.
%m	Десятичное представление месяца, '01'–'12'.
%M	Десятичное представление минут, '01'–'59'.
%p	Национальный эквивалент обозначения 'AM' (до полудня) или 'PM' (после полудня).
%S	Десятичное представление секунд, '00'–'61'. Значения 60 и 61 используются изредка для согласования с солнечным календарем.
%U	Десятичное представление порядкового номера недели года, '00'–'53'. Считается, что неделя начинается с воскресенья, все дни в новом году до первого воскресенья относятся к неделе с номером 0.
%w	Десятичное представление дня недели, '0'–'6' ('0' соответствует воскресенью).
%W	Десятичное представление порядкового номера недели года, '00'–'53'. Считается, что неделя начинается с понедельника, все дни в новом году до первого понедельника относятся к неделе с номером 0.
%x	Общепринятое (в соответствии с текущими национальными установками) представление даты.
%X	Общепринятое (в соответствии с текущими национальными установками) представление времени.
%y	Представление года без указания века (двумя десятичными цифрами), '00'–'99'.
%Y	Полное десятичное представление года (четырьмя цифрами).
%Z	Название часового пояса (или пустая строка, если часовой пояс не задан).
%%	Буква '%'.

Некоторые платформы могут поддерживать дополнительные управляющие последовательности и позволяют указать ширину поля.

strptime(*string* [, *format*])

(большинство современных вариантов UNIX)

Разбирает строковое представление времени *string* в соответствии с форматом *format* и возвращает кортеж с числами, описанный выше. В аргументе *format* должны использоваться такие же управляющие последовательности, как и в строке формата функции strftime(). По умолчанию *format* равен строке '%a %b %d %H:%M:%S %Y', соответствующей формату, используемому функцией ctime(). Если строка *string* не соответствует формату *format*, генерируется исключение ValueError.

time()

Возвращает универсальное время (UTC) в виде вещественного числа в секундах с начала эпохи. Заметим, что не все платформы предоставляют время с точностью большей, чем 1 секунда.

timezone

Сдвиг локального часового пояса (без учета перехода на летнее время) в секундах на запад от UTC (то есть положительное в США, равное нулю в Великобритании и отрицательное в России).

tzname

Возвращает кортеж из двух строк: первая является названием локального часового пояса без учета перехода на летнее время и вторая — с учетом перехода на летнее время (если таковой не определен, эта строка не должна быть использована).

21.8 sched — планирование задач

Модуль sched определяет класс, позволяющий планировать запуск задач:

scheduler(*timefunc*, *delayfunc*)

Возвращает объект, являющийся планировщиком задач общего назначения. Аргументы — функции, взаимодействующие с внешним миром. Функция *timefunc* вызывается без аргументов и должна возвращать число (время, в любых единицах). Функция *delayfunc* вызывается с одним аргументом — временем (в тех же единицах, которые использует функция *timefunc*), должна осуществлять приостановку выполнения на указанное время. Кроме того, чтобы дать возможность выполняться другим потокам в многопоточных приложениях, после выполнения каждой задачи функция *timefunc* вызывается с аргументом 0.

Экземпляры класса scheduler имеют следующие методы:

enterabs(*time*, *priority*, *action*, *arguments*)

Планирует выполнение новой задачи в указанное время. Аргумент *time* должен быть числом, выражающим время в тех же единицах, которые использует функция *timefunc*. В указанное время будет вызвана функция *action* с аргументами *arguments* (кортеж). Задачи, запланированные на одно и то же время, будут выполняться в порядке их приоритетов (аргумент *priority*).

Метод возвращает объект, представляющий запланированную задачу, который может быть использован для ее снятия с помощью метода cancel(), описанного ниже.

enter(*delay*, *priority*, *action*, *arguments*)

Планирует выполнение новой задачи через указанное время. Аргумент *delay*

должен быть числом, выражающим время в тех же единицах, которые использует функция *timefunc*, по прошествии которого должна быть запущена задача. Остальные аргументы и возвращаемое значение имеют такое же значение, как и для метода enterabs().

cancel(*task*)
Удаляет задачу из очереди. Если в очереди отсутствует задача *task*, генерирует исключение RuntimeError.

empty()
Возвращает 1, если очередь пуста, иначе возвращает 0.

run()
Выполняет все запланированные задачи. Этот метод будет ожидать (используя функцию delayfunc, переданную конструктору) наступления времени, в которое должна быть выполнена каждая задача, выполнит ее и т. д. до тех пор, пока не будут выполнены все задачи в очереди.

Исключения, сгенерированные функциями *timefunc*, *delayfunc* и *action* не обрабатываются. Если исключение сгенерировала функция *action*, планировщик не будет пытаться выполнить эту задачу при последующих вызовах метода run().

Приведем простой пример:

```
>>> import sched, time
>>> s=sched.scheduler(time.time, time.sleep)
>>> def print_time():
...     print "В функции print_time:", time.time()
...
>>> def print_some_times():
...     print time.time()
...     s.enter(5, 1, print_time, ())
...     s.enter(10, 1, print_time, ())
...     s.run()
...     print time.time()
...
>>> print_some_times()
930343690.257
В функции print_time: 930343695.274
В функции print_time: 930343700.273
930343700.276
```

21.9 getpass — запрос пароля и определение имени пользователя

Модуль getpass определяет две функции:

getpass([*prompt*]) (UNIX, Windows, Macintosh)
Запрашивает пароль у пользователя без отображения вводимых символов. В качестве приглашения используется строка *prompt* (по умолчанию равна 'Password: ').

getuser() (UNIX, Windows)
Возвращает регистрационное имя пользователя. Эта функция последовательно проверяет переменные окружения LOGNAME, USER, LNAME и USERNAME и возвращает значение первой переменной, имеющей непустое значение. Если ни одна из этих переменных не установлена, используется модуль pwd там, где он доступен (в противном случае генерируется исключение ImportError).

21.10 getopt — обработка опций в командной строке

Модуль getopt помогает обрабатывать аргументы, переданные программе в командной строке (sys.argv). Функция getopt() модуля использует такие же правила, как и одноименная функция языка C в UNIX, включая специальное значение аргументов '-' и '--'. Длинная запись опций, аналогичных тем, которые используются программами GNU, также может быть использована, если задан третий аргумент функции getopt().

getopt(*args*, *options* [, *long_options*])
Обрабатывает список аргументов *args* (обычно sys.argv[1:]).

Аргумент *options* должен быть строкой из букв, которые будут распознаваться как опции. Если опция требует наличия аргумента, после соответствующей буквы в строке *options* должно следовать двоеточие (':').

Если задан аргумент *long_options*, он должен быть списком строк с именами опций, которые будут распознаваться в виде длинной записи, без приставки '--'. После имени опций, требующих аргумента, в строке должен следовать знак равенства ('=').

Функция возвращает кортеж из двух элементов: первый является списком пар '(*option*, *value*)' (имя опции и соответствующее ей значение) и список аргументов, оставшихся необработанными (концевой срез последовательности *args*). Строки *option* в списке пар начинаются с одного дефиса для опций с короткой записью и с двух дефисов для опций с длинной записью. *value* является аргументом соответствующей опции или пустой строкой, если опция не имеет аргумента. Опции в списке следуют в том же порядке, в котором они следовали в *args*. Допускается смешение короткой и длинной записи опций.

GetoptError
error
Исключения этого класса генерируются, если в списке аргументов найдена нераспознанная опция или опция, требующая аргумента, записана без него. Исключение

также будет сгенерировано, если при использовании длинной записи задан аргумент для опции, которая его не требует. Исключения класса GetoptError имеют следующие атрибуты:

msg
Сообщение об ошибке.

opt
Имя опции, с которой связана ошибка, или пустая строка, если ошибка не связана с определенной опцией.

Имя GetoptError появилось в версии 1.6. Для совместимости с более старыми версиями присутствует имя error, ссылающееся на GetoptError.

Приведем небольшой пример, использующий только короткую запись опций:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Использовать длинную запись опций так же просто:

```
>>> s = '--condition=foo --testing --output-file
... abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file',
'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''),
('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

В реальной программе модуль getopt используется обычно примерно следующим образом:

```
import getopt, sys
```

```
USAGE = """\
...
"""

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:],
                                   'ho:',
                                   ['help', 'output='])
    except getopt.GetoptError:
        # выводим подсказку и выходим:
        print << sys.stderr, USAGE
        sys.exit(2)
    output = None
    for o, a in opts:
        if o in ('-h', '--help'):
            print USAGE
            sys.exit()
        if o in ('-o', '--output'):
            output = a
    # ...

if __name__ == '__main__':
    main()
```

21.11 `tempfile` — создание временных файлов

Этот модуль предоставляет переносимый способ генерации уникальных имен для временных файлов и их создания:

mktemp([*suffix*])

Возвращает уникальное имя временного файла. Это абсолютный путь файла, который в момент вызова функции не существует. При последующих вызовах функция `mktemp()` никогда не вернет такое же имя. Если задан аргумент *suffix*, он будет использоваться в качестве последней части (суффикса) имени файла.

TemporaryFile([*mode* [, *bufsize* [, *suffix*]]])

Возвращает файловый (или аналогичный) объект, который может быть использован для временного хранения данных. Файл создается максимально безопасным способом в соответствующем временном каталоге. В UNIX каталог, в котором находится файл, удаляется, что позволяет избежать атак, касающихся создания символической ссылки на файл или замены файла ссылкой на другой файл. В любом случае файл автоматически удаляется, как только он будет закрыт (включая неявное закрытие при достижении нуля количества ссылок на файловый объект).

Аргумент *mode* указывает режим, в котором файл будет открыт. По умолчанию он равен `'w+b'`, что позволяет как записывать, так и читать из файла не закрывая

его. Аргумент *bufsize* имеет такое же значение, как и во встроенной функции open(). Если задан аргумент *suffix*, он будет использоваться в качестве последней части (суффикса) имени файла.

gettempprefix()
Возвращает приставку к имени, которая будет использоваться для создания временных файлов (не включает в себя имя каталога).

Модуль определяет следующие глобальные переменные, определяющие, каким образом будет конструироваться имя временного файла. Заметим, что присваивание им нового значения невозможно, если Вы использовали инструкцию вида 'from tempfile import ...' для импортирования модуля.

tempdir
Если эта переменная имеет значение отличное от None, определяет имя каталога, в котором будут создаваться временные файлы. Инициализируется при первом вызове функции mktemp(). Значение по умолчанию берется из переменной окружения TMPDIR, TEMP или TMP. Если ни одна из этих переменных не установлена, используется общепринятая для данной платформы ('/var/tmp', '/usr/tmp' или '/tmp' в UNIX) или текущий каталог.

template
Если эта переменная имеет значение, отличное от None, определяет приставку к именам временных файлов. Инициализируется при первом вызове функции mktemp(). По умолчанию используется '@*pid*.' в UNIX, '~*pid*-' в Windows NT, 'Python-Tmp-' в Macintosh и 'tmp' на других платформах.

21.12 errno — символические имена стандартных системных ошибок

Этот модуль предоставляет символические имена для стандартных системных ошибок. Значением каждой символической константы является целое число, которое используется в качестве кода ошибки в исключениях класса IOError (атрибут errno).

errorcode
Словарь, отображающий код ошибки к его имени в данной системе. Например, выражение errorcode[EPERM] дает 'EPERM'.

Для получения сообщения об ошибке, соответствующего ее коду, используйте функцию os.strerror().

Ниже перечислены основные символические константы (в соответствии со стандартами POSIX.1, ISO C). Из этого списка модуль определяет только те константы, которые используются на данной платформе (Вы можете получить полный список имен доступных констант как результат выражения errno.errorcode.values()):

EPERM
Недопустимая операция.

ENOENT
Нет такого файла или каталога.

ESRCH
Нет такого процесса.

EINTR
Прерван системный вызов.

EIO
Ошибка ввода/вывода.

ENXIO
Нет такого устройства или адреса.

E2BIG
Слишком длинный список аргументов.

ENOEXEC
Неверный формат аргументов системного вызова `exec()`.

EBADF
Неверный дескриптор файла.

ECHILD
Нет дочернего процесса.

EAGAIN
Ресурс временно недоступен.

ENOMEM
Не хватает памяти.

EACCES
Доступ запрещен.

EFAULT
Неверный адрес.

EBUSY
Устройство или ресурс занят.

EEXIST
Файл уже существует.

EXDEV
Неверная ссылка (на файл, находящийся на другом устройстве).

ENODEV
Нет такого устройства.

ENOTDIR
Не является каталогом.

EISDIR
Является каталогом.

EINVAL
Неверный аргумент.

ENFILE
Слишком много открытых файлов в системе (переполнение таблицы файлов).

EMFILE
Слишком много открытых файлов.

ENOTTY
Неверная операция управления вводом/выводом (устройство должно быть терминалом).

ETXTBSY
Текстовый файл занят (например, при попытке выполнить файл, открытый для записи).

EFBIG
Файл слишком большой.

ENOSPC
На устройстве не осталось свободного места.

ESPIPE
Неверное перемещение указателя.

EROFS
Файловая система доступна только для чтения.

EMLINK
Слишком много ссылок.

EPIPE
Нарушен канал (pipe).

EDOM
Аргумент математической операции выходит за пределы области допустимых значений.

ERANGE
Результат математической операции не представим.

EDEADLK
Выполнение операции привело бы к взаимной блокировке.

ENAMETOOLONG
Слишком длинное имя файла.

ENOLCK
Блокировка записей недоступна.

ENOSYS
Функция не реализована.

ENOTEMPTY
Каталог не является пустым.

EBADMSG
Неверное сообщение.

EILSEQ
Неверная последовательность байтов.

EMSGSIZE
Слишком длинное сообщение.

ETIMEDOUT
Истекло время ожидания завершения операции.

EINPROGRESS
Операция в процессе выполнения.

21.13 `glob` — раскрытие шаблона имен путей

Модуль `glob` находит все имена путей, удовлетворяющих определенному шаблону, в соответствии с правилами, используемыми в UNIX shell. Этот модуль не раскрывает имя домашнего каталога, записанного с помощью символа '~', но корректно сопоставляет имена, возвращаемые функцией `os.listdir()` с заданным шаблоном с помощью функции `fnmatch.fnmatch()` (см. описание модуля `fnmatch` для ознакомления с синтаксисом шаблонов). Если Вы хотите раскрыть имя домашнего каталога, воспользуйтесь функцией `os.path.expanduser()`.

glob(*pathname*)
Возвращает (возможно, пустой) список имен путей, удовлетворяющих шаблону *pathname*. Аргумент *pathname* должен быть строкой, содержащей имя пути (абсолютное или относительное) и, возможно, групповые символы (см. описание модуля `fnmatch`). Групповые символы *не* удовлетворяют разделителю компонент имени пути[2].

Допустим, к примеру, что текущий каталог содержит только файлы '1.gif', '2.txt' и 'card.gif'. Тогда функция `glob()` будет работать следующим образом (обратите внимание, что запись начала пути сохраняется):

[2]Специальные последовательности вида '[*seq*]' и '![*seq*]' не должны содержать символы разделителей имени пути, так как это приведет к некорректной работе функции.

```
>>> from glob import glob
>>> glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob('*.gif')
['1.gif', 'card.gif']
>>> glob('?.gif')
['1.gif']
```

21.14 fnmatch — сопоставление имен файлов с шаблоном

Этот модуль предоставляет возможность сопоставления имен файлов (каталогов) с шаблоном, содержащим групповые символы в стиле UNIX shell. Возможность использования более сложных шаблонов (регулярных выражений) предоставляет модуль re.

Шаблон может содержать следующие специальные последовательности (групповые символы):

Последовательность	Назначение
*	Удовлетворяет любому количеству любых символов.
?	Удовлетворяет (одному) любому символу.
[*seq*]	Удовлетворяет (одному) любому символу, содержащемуся в последовательности *seq*.
[!*seq*]	Удовлетворяет (одному) любому символу, не содержащемуся в последовательности *seq*.

Обратите внимание, что разделители компонент имени пути ('/', '\', ':') не имеют специального значения. Точка в начале имени файла не рассматривается отдельно и удовлетворяет групповым символам '*' и '?'.

fnmatch(*filename, pattern*)

Возвращает 1, если имя файла (каталога) удовлетворяет шаблону *pattern*, иначе возвращает 0. Если ОС не чувствительна к регистру имен файлов, оба аргумента перед сопоставлением будут приведены к одному регистру функцией os.path.normcase(). Функция os.path.normcase() также заменяет все вхождения os.altsep на os.sep, то есть, например в Windows, символы '\' и '/' могут использоваться взаимозаменяемо (шаблоны '/' и '\\' удовлетворяют как '/', так и '\\').

fnmatchcase(*filename, pattern*)

Возвращает 1, если имя файла (каталога) удовлетворяет шаблону *pattern*, иначе возвращает 0. Аргументы не подвергаются никакой обработке, сравнение всегда производится с учетом регистра букв.

Модуль glob позволяет раскрыть шаблон, то есть найти все файлы, удовлетворяющие шаблону.

21.15 shutil — операции над файлами высокого уровня

Модуль shutil предоставляет набор операций высокого уровня над файлами и коллекциями файлов.

Предостережение: в операционных системах Macintosh не используются ответвления ресурсов и другие метаданные. Это означает, что при копировании файлов ресурсы будут утрачены, тип и коды создателя файлов не будут правильными.

copyfile(*src, dst*)
Копирует содержимое файла именем *src* в файл с именем *dst*. Если файл *dst* существует, он будет заменен.

copyfileobj(*fsrc, fdst* [, *bufsize*])
Копирует содержимое файлового (или подобного) объекта *fsrc* в файловый (или подобный) объект *fdst*. Если задан аргумент *bufsize*, он используется в качестве размера буфера (отрицательные значения означают, что файл не будет копироваться по частям).

copymode(*src, dst*)
Устанавливает у файла с именем *dst* биты разрешений доступа такие же, как у файла *src*.

copystat(*src, dst*)
Устанавливает у файла с именем *dst* биты разрешений доступа, время последнего доступа и время последнего внесения изменений такие же, как у файла *src*. Содержимое файла, владелец и группа остаются неизменными.

copy(*src, dst*)
Копирует файл с именем *src* в файл с именем *dst*. Если *dst* является каталогом, создает (или заменяет) в нем файл с таким же (основным) именем, как и *src*. Биты разрешения доступа также копируются.

copy2(*src, dst*)
Работает аналогично функции copy(), но также устанавливает у файла с именем *dst* время последнего доступа и время последнего внесения изменений такие же, как у файла *src* (то есть работает аналогично команде 'cp -p' в UNIX).

copytree(*src, dst* [, *symlinks*])
Рекурсивно копирует дерево каталогов *src* в *dst*. Каталог с именем *dst* не должен существовать — он будет создан. Копирование всех файлов производится с помощью функции copy2(). Если задан и является истиной аргумент *symlinks*, символические ссылки в исходном дереве будут представлены символическими

ссылками в новом дереве, в противном случае, в новое дерево будет скопировано содержимое файлов, на которые указывают символические ссылки. Ошибки, возникающие при копировании, выводятся на стандартный поток ошибок.

Эту функцию следует воспринимать скорее как пример (см. исходный текст модуля), нежели как реальное средство для работы с файлами.

rmtree(*path* [, *ignore_errors* [, *onerror*]])

Удаляет полностью дерево каталогов *path*. Если установлен и является истиной аргумент *ignore_errors*, ошибки, возникающие при удалении, будут проигнорированы, в противном случае для обработки ошибок вызывается функция *onerror* (или генерируется исключение, если аргумент *onerror* не задан).

Обработчик ошибок *onerror* должен поддерживать вызов с тремя аргументами: *function*, *path* и *excinfo*. *function* — функция, которая сгенерировала исключение (`os.remove()` или `os.rmdir()`), *path* — имя пути, которое было передано функции *function*, и *excinfo* — информация об исключении, возвращаемая функцией `sys.exc_info()`. Исключения, сгенерированные обработчиком ошибок, не обрабатываются.

21.16 signal — обработка асинхронных событий

Этот модуль предоставляет возможность устанавливать обработчики сигналов. Несколько общих правил:

- Однажды установленный обработчик сигнала остается до тех пор, пока Вы явно не восстановите исходный обработчик. Единственное исключение из этого правила — обработчик сигнала SIGCHLD, работа которого зависит от реализации на данной платформе.
- Не существует способа временно заблокировать сигналы в критических местах.
- Хотя обработчики сигналов и вызываются асинхронно, это может произойти только между атомными инструкциями интерпретатора Python. Это значит, что если сигнал поступает в момент длительной работы процедуры, полностью реализованной на языке C (например, использование регулярного выражения для поиска в длинном тексте), его обработка может быть отложена на произвольное время.
- При поступлении сигнала во время выполнения операции ввода/вывода после завершения обработки сигнала может быть сгенерировано исключение.
- Не имеет особого смысла обработка синхронных ошибок, таких как SIGFPE или SIGSEGV.
- Интерпретатор устанавливает некоторое количество обработчиков по умолчанию: сигнал SIGPIPE игнорируется (таким образом, уведомление об ошибках при работе с каналами и сетевыми соединениями может быть выполнено генерацией обычных исключений) и SIGINT генерирует исключение KeyboardInterrupt. Вы можете заменить любой из этих обработчиков.

- Следует проявлять некоторую осторожность при использовании модуля в многопоточных программах. Основное правило: всегда вызывайте функцию `signal()` только в основном потоке. Любой поток может использовать функции `alarm()`, `getsignal()` и `pause()`, но только основной — устанавливать обработчик и получать сигнал. Таким образом, сигналы не могут быть использованы для связи между потоками — для этих целей следует использовать блокировки.

Модуль `signal` определяет следующие константы и функции:

SIG_DFL
Используется в качестве аргумента функции `signal()` и значения, возвращаемого функцией `getsignal()` для обозначения обработчика, который используется системой (не интерпретатором) по умолчанию.

SIG_IGN
Используется в качестве аргумента функции `signal()` для обозначения обработчика, игнорирующего сигнал.

SIG*
Символические константы для номеров сигналов. Модуль определяет константы только для тех сигналов, которые поддерживаются в данной системе.

NSIG
Имеет значение на единицу большее, чем максимальный номер сигнала в данной системе.

alarm(*time*) (UNIX)
Если аргумент *time* не равен нулю, запрашивает посылку сигнала `SIGALRM` текущему процессу через *time* секунд. Ранее запланированная посылка сигнала при этом отменяется. Возвращает время в секундах, которое оставалось до посылки ранее запланированного сигнала. Если аргумент *time* равен нулю, отменяет ранее запланированную посылку сигнала и возвращает время, которое оставалось до его посылки. Если функция `alarm()` возвращает `0`, посылка сигнала до этого не была запланирована.

getsignal(*signalnum*)
Возвращает текущий обработчик сигнала с номером *signalnum*. Возвращаемое значение может быть объектом, поддерживающим вызов, или одним из специальных значений: `signal.SIG_IGN`, `signal.SIG_DFL` или `None`. Значение `None` означает, что обработчик не был установлен интерпретатором.

pause() (UNIX)
Приостанавливает выполнение процесса до получения сигнала, после чего будет вызван соответствующий обработчик.

signal(*signalnum*, *handler*)
Устанавливает *handler* в качестве обработчика сигнала с номером *signalnum*. Аргумент *handler* может объектом, поддерживающим вызов, или одним из специальных значений: `SIG_IGN` или `SIG_DFL`. Возвращает предыдущий обработчик сигнала.

Функция *handler* будет вызываться для обработки сигнала *signalnum* с двумя аргументами: номер сигнала и текущий кадр стека (или None).

В многопоточной программе при попытке вызвать эту функцию из неосновного потока генерируется исключение ValueError.

Приведем простой пример, использующий функцию alarm() для того, чтобы ограничить время ожидания открытия файла. Это может быть полезно, если файл является последовательным устройством, который может быть не включенным. Использование функции os.open() для таких файлов приведет к зависанию. Для того, чтобы решить эту проблему, можно перед открытием файла запросить посылку сигнала SIGALRM через 5 секунд: если операция займет слишком много времени, будет послан сигнал и обработчик сгенерирует исключение.

```
import signal, os

def handler(signum, frame):
    raise IOError("Couldn't open device!")

# Устанавливаем обработчик
signal.signal(signal.SIGALRM, handler)
# Запрашиваем посылку сигнала через 5 секунд
signal.alarm(5)

# Эта операция может привести к зависанию
fd = os.open('/dev/ttyS0', os.O_RDWR)

# Отключаем посылку сигнала
signal.alarm(0)
```

21.17 socket — сетевой интерфейс низкого уровня

Модуль socket предоставляет низкоуровневый доступ к сетевым соединениям. Этот модуль доступен для всех современных вариантов UNIX, Windows, Macintosh, BeOS, OS/2 и, возможно, для некоторых других платформ.

Адреса для сетевых соединений представляются одной строкой для семейства AF_UNIX и как пара '(*host*, *port*)' для семейства AF_INET, где *host* — строка, содержащая имя узла (например, 'daring.cwi.nl') или IP адрес (например, '100.50.200.5'), и *port* — номер порта. Воспринимаются также две специальные формы IP адресов: пустая строка соответствует INADDR_ANY и строка '<broadcast>' соответствует INADDR_BROADCAST.

Поддержка режима без блокировки осуществляется через метод setblocking().

Модуль socket определяет следующие константы и функции:

error

Исключения этого класса генерируются в случае возникновения ошибок, связанных с сетевым соединением или адресом. Аргументами исключения являются строка, описывающая причину возникновения ошибки, и, возможно, системный код ошибки (см. описание модуля `errno`).

AF_UNIX (UNIX)
AF_INET

Эти константы используются в качестве первого аргумента конструктора `socket()` и указывают семейство, к которому относится адрес соединения (UNIX-соединение или соединение с Internet).

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_RDM
SOCK_SEQPACKET

Эти константы используются в качестве второго аргумента конструктора `socket()` и указывают тип соединения (обычно используются только первые два типа).

INADDR_*
IP_*
IPPORT_*
IPPROTO_*
MSG_*
SO_*
SOL_*
SOMAXCONN

Множество констант такого вида определено в этом модуле и обычно используются в качестве аргументов методов `setsockopt()` и `getsockopt()` объектов, представляющих сетевое соединение. Информацию о них Вы можете получить в документации UNIX по сетевым соединениям и протоколу IP.

getfqdn([*name*])

Возвращает полное уточненное доменное имя для *name*. Если аргумент *name* опущен или является пустой строкой, он считается соответствующим локальному узлу. Для того, чтобы получить полное уточненное имя, проверяется сначала имя узла, возвращаемое функцией `gethostbyaddr()`, затем псевдонимы, если они имеются. Выбирается первое имя, содержащее точку. В случае, если уточненное имя не доступно, возвращает имя узла. Функция доступна, начиная с версии 2.0.

gethostbyname(*hostname*)

Возвращает строку, содержащую IP адрес узла с именем *hostname*. Если строка *hostname* содержит IP адрес, она возвращается без изменений.

gethostbyname_ex(*hostname*)

Возвращает кортеж '(*mainhostname*, *aliaslist*, *ipaddrlist*)', где *mainhostname* — основное имя узла, соответствующего *hostname*, *aliaslist* — список (возможно пустой) альтернативных имен для того же

адреса и *ipaddrlist* — список IP адресов для того же интерфейса к тому же узлу (обычно содержит одну строку).

gethostname()

Возвращает строку, содержащую имя узла машины, на которой в данный момент выполняется интерпретатор языка Python. Если Вы хотите получить IP адрес текущей машины, используйте gethostbyname(gethostname()). Заметим, что функция gethostname() не всегда возвращает полное имя — для этих целей следует использовать getfqdn(gethostname()).

gethostbyaddr(*ip_address*)

Возвращает кортеж '(*hostname*, *aliaslist*, *ipaddrlist*)', где *hostname* — основное имя узла, соответствующего IP адресу *ip_address*, *aliaslist* — список (возможно пустой) альтернативных имен для того же адреса и *ipaddrlist* — список IP адресов для того же интерфейса к тому же узлу (обычно содержит одну строку).

getprotobyname(*protocolname*)

Возвращает число, подходящее для использования в качестве третьего аргумента функции socket() и соответствующее протоколу с именем *protocolname*. Это необходимо только, если Вы открываете соединение в "прямом" режиме (SOCK_RAW), для остальных типов соединений правильный протокол выбирается автоматически, если соответствующий аргумент опущен или равен нулю.

getservbyname(*servicename*, *protocolname*)

Возвращает номер порта для службы с именем *servicename* и протоколом с именем *protocolname*. Имя протокола должно быть равным 'tcp' или 'udp'.

socket(*family*, *type* [, *proto*])

Создает новое сетевое соединение и возвращает соответствующий ему объект. *family* — семейство адресов (AF_INET или AF_UNIX), *type* — тип соединения (одна из констант SOCK_*) и *proto* — номер протокола. Если аргумент *proto* опущен, протокол выбирается автоматически.

fromfd(*fd*, *family*, *type* [, *proto*])

Создает объект сетевого соединения из файлового дескриптора *fd* и возвращает его. Файловый дескриптор должен соответствовать сетевому соединению (однако это не проверяется — последующие операции могут вызвать ошибку, если дескриптор непригоден). Остальные аргументы имеют такое же значение, как и для функции socket().

ntohl(*x*)

Преобразует представление 32-битового целого числа с общепринятым в сети порядком следования байтов к представлению с порядком следования байтов на данной машине.

ntohs(*x*)

Преобразует представление 16-битового целого числа с общепринятым в сети порядком следования байтов к представлению с порядком следования байтов на данной машине.

htonl(*x*)
Преобразует представление 32-битового целого числа с порядком следования байтов на данной машине к представлению с общепринятым в сети порядком следования байтов.

htons(*x*)
Преобразует представление 16-битового целого числа с порядком следования байтов на данной машине к представлению с общепринятым в сети порядком следования байтов.

inet_aton(*ip_string*)
Возвращает строку из четырех символов, содержащую IP адрес *ip_string* (строка вида '123.45.67.89') в упакованном двоичном виде. Эта функция может быть полезна для связи с программой, использующей стандартную библиотеку языка C.

inet_ntoa(*packed_ip*)
Возвращает строку с IP адресом, полученным распаковкой строки из четырех символов *packed_ip*, содержащей упакованное двоичное представление адреса.

SocketType
Тип объекта сетевого соединения, возвращаемого функцией socket().

Объекты сетевого соединения, возвращаемые функцией socket(), имеют следующие методы:

accept()
Принимает соединение и возвращает пару '(*conn*, *address*)', где *conn* — новый объект сетевого соединения, подходящий для посылки и получения данных, и *address* — адрес другого конца сетевого соединения. Объект сетевого соединения, к которому применяется метод accept(), должен быть привязан к адресу (метод bind()) и ожидать соединения (метод listen()).

bind(*address*)
Привязывает сетевое соединение к адресу *address*. Перед вызовом этого метода объект не должен быть привязанным к адресу. Формат аргумента зависит от семейства, к которому относится адрес (см. начало раздела).

close()
Закрывает сетевое соединение, после чего все операции над объектом будут вызывать ошибку. Сетевое соединение автоматически закрывается при достижении нуля количества ссылок на объект сетевого соединения.

connect(*address*)
Подсоединяется к удаленному узлу с адресом *address*. Формат аргумента зависит от семейства, к которому относится адрес (см. начало раздела).

connect_ex(*address*)
Работает аналогично методу connect(), но возвращает код ошибки, возвращаемый системной функцией connect() вместо того, чтобы генерировать исключение. Возвращаемое значение равно 0 в случае успешного выполнения операции и

значению системной переменной errno при возникновении ошибки. Метод может быть полезен для асинхронных соединений.

fileno()
Возвращает файловый дескриптор сетевого соединения (целое число). Может быть полезен совместно с функцией select.select().

getpeername()
Возвращает адрес удаленного узла, с которым установлено соединение. Формат возвращаемого значения зависит от семейства, к которому относится адрес (см. начало раздела). Метод может быть полезен, например, для определения номера порта на другом конце IP соединения.

getsockname()
Возвращает собственный адрес. Формат возвращаемого значения зависит от семейства, к которому относится адрес (см. начало раздела). Метод может быть полезен, например, для определения номера порта.

getsockopt(*level*, *optname* [, *buflen*])
Возвращает значение опции *optname*. В качестве аргумента *optname* может быть использована одна из символических констант вида SO_*, определенных в модуле. Если аргумент *buflen* опущен, возвращает целочисленное значение опции. Если аргумент *buflen* задан, он указывает максимальную длину возвращаемой строки.

listen(*backlog*)
Ожидает установления сетевого соединения. Аргумент *backlog* определяет максимальное число соединений в очереди и должен быть не меньше единицы (максимальное значение зависит от системы, обычно равно 5).

makefile([*mode* [, *bufsize*]])
Возвращает файловый объект (типа file, см. раздел 11.7), ассоциированный с сетевым соединением. Файловый объект ссылается на дубликат файлового дескриптора для сетевого соединения, таким образом, файловый объект и объект сетевого соединения могут быть закрыты (в том числе неявно — при достижении нуля количества ссылок на объект) независимо друг от друга. Аргументы *mode* и *bufsize* имеют такое же значение, как и для встроенной функции open().

recv(*bufsize* [, *flags*])
Получает данные через сетевое соединение и возвращает их в виде строки. Максимальное количество данных, получаемое за один раз, определяется аргументом *bufsize*. Описание возможных значений аргумента *flags* смотрите в документации для системной функции recv().

recvfrom(*bufsize* [, *flags*])
Работает аналогично методу recv(), но возвращает пару '(*string*, *address*)', где *string* — строка с полученными данными и *address* — адрес, с которого данные были посланы. Формат *address* зависит от семейства, к которому относится адрес (см. начало раздела).

send(*string* [, *flags*])
Посылает данные через сетевое соединение. Соединение должно быть установленным. Аргумент *flags* имеет такое же значение, как и для метода `recv()`. Возвращает размер посланных данных в байтах.

sendto(*string* [, *flags*], *address*)
Посылает данные на узел с адресом *address*. Соединение не должно быть установленным. Аргумент *flags* имеет такое же значение, как и для метода `recv()`. Возвращает размер посланных данных в байтах. Формат аргумента зависит от семейства, к которому относится адрес (см. начало раздела).

setblocking(*flag*)
Если *flag* равен `0`, устанавливает режим без блокировки, иначе устанавливает режим с блокировкой. Изначально все сетевые соединения находятся в режиме с блокировкой. В режиме без блокировки, если операция `recv()` или `send()` не может быть выполнена немедленно, генерируется исключение `error`. В режиме с блокировкой вызов методов `recv()` или `send()` приведет к блокированию до тех пор, пока операция не будет выполнена.

setsockopt(varlevel, *optname*, *value*)
Устанавливает значение опции *optname* равным *value*. В качестве аргумента *optname* может быть использована одна из символических констант вида `SO_*`, определенных в модуле. Аргумент *value* может быть целым числом или строкой с упакованными данными.

shutdown(*how*)
Отключает одну или обе половины соединения. Если аргумент *how* равен `0`, в дальнейшем получение данных будет запрещено. Если аргумент *how* равен `1`, запрещена будет передача данных. Если аргумент *how* равен `2`, в дальнейшем будет запрещено как получение, так и передача данных.

Приведем два простых примера, использующих TCP/IP протокол: сервера, посылающего назад полученные данные (обслуживая только одного клиента), и клиента, его использующего. Обратите внимание, что сервер должен вызвать последовательно методы `socket()`, `bind()`, `listen()`, `accept()` (возможно повторяя вызов `accept()`, если необходимо обслуживать несколько клиентов), в то время как клиенту необходимо выполнить только `socket()`, `connect()`. Кроме того, сервер применяет методы `send()` и `recv()` не к объекту, ожидающему соединения, а к новому объекту, возвращаемому методом `accept()`.

```
# Сервер
import socket

# Локальный узел
HOST = ''
# Произвольный непривилегированный порт
PORT = 50007
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Подсоединились с адреса', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

```
# Клиент
import socket

# Удаленный узел
HOST = 'daring.cwi.nl'
# Тот же порт, что используется сервером
PORT = 50007

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Полученные данные:', `data`
```

21.18 `select` — ожидание завершения ввода/вывода

Этот модуль предоставляет доступ к системным функциям `select()` и `poll()`. Заметим, что в Windows функция `select()` работает только для сетевых соединений. В UNIX средства, предоставляемые модулем, работают также с другими типами файлов, в частности, каналами. Однако эти средства не могут быть использованы для определения увеличения в размерах обычного файла.

`error`

Исключения этого класса генерируются при возникновении различных ошибок. В качестве аргументов при инициализации используются номер системной ошибки и соответствующая ему поясняющая строка.

`select`(*iwtd*, *owtd*, *ewtd* [, *timeout*])

Первые три аргумента — списки, которые могут содержать файловые дескрипторы или объекты с методом `fileno()`, возвращающим файловый дескриптор. *iwtd* — список объектов для ввода, *owtd* — для вывода и *ewtd* — для исключительных ситуаций (вывода ошибок). Аргумент *timeout* указывает максимальное

время ожидания, если он опущен, функция блокирует выполнение до тех пор, пока один из дескрипторов не будет готов.

Функция `select()` возвращает кортеж из трех списков, являющихся подсписками аргументов *iwtd*, *owtd* и *ewtd* и содержащих объекты, готовые к вводу/выводу. При достижении максимального времени ожидания, возвращает кортеж из трех пустых списков.

poll() (большинство вариантов UNIX)
: Возвращает объект, поддерживающий регистрацию файловых дескрипторов и реализующий их опрос на предмет наступления событий ввода/вывода.

Объекты, возвращаемые функцией `poll()`, имеют следующие методы:

register(*fd* [, *eventmask*])
: Регистрирует файловый дескриптор (или объект с методом `fileno()`, возвращающим файловый дескриптор) *fd*. При последующих вызовах метода `poll()` файловый дескриптор будет проверяться на предмет наступления событий ввода/вывода. Аргумент *eventmask* может быть комбинацией (с помощью оператора '`|`') констант POLLIN, POLLPRI и POLLOUT и указывает интересующие типы событий. По умолчанию проверяется наступление событий всех трех типов. Повторная регистрация одного и того же файлового дескриптора не является ошибкой.

unregister(*fd*)
: Снимает с регистрации файловый дескриптор (или объект с методом `fileno()`, возвращающим файловый дескриптор) *fd*. При попытке снять с регистрации объект, который не был зарегистрирован, будет сгенерировано исключение `KeyError`.

poll([*timeout*])
: Опрашивает зарегистрированные файловые дескрипторы на предмет наступления событий ввода/вывода и возвращает список пар вида '`(`*fd*`, `*event*`)`', где *fd* — файловый дескриптор, для которого наступило событие и возникла ошибка, и *event* — комбинация констант (см. ниже), описывающих наступившее событие или возникшую ошибку. Аргумент *timeout* указывает максимальное время ожидания, если он опущен, метод блокирует выполнение до тех пор, пока один из дескрипторов не будет готов. При достижении максимального времени ожидания, возвращает пустой список.

Ниже приведены константы, комбинации которых могут быть использованы в качестве аргумента *eventmask* метода `register()` и в результате, возвращаемом методом `poll()`:

POLLIN
: Имеются данные для чтения.

POLLPRI
: Имеются экстренные данные для чтения.

POLLOUT
Файловый дескриптор готов к выводу (то есть запись не приведет к блокированию).

POLLERR
Исключительная ситуация.

POLLHUP
Зависание.

POLLNVAL
Неверный запрос: дескриптор не открыт.

21.19 mmap — отображение файлов в память

Модуль mmap доступен в операционных системах UNIX и Windows и позволяет отображать файлы в память. Объекты, представляющие отображенные в память файлы, ведут себя с одной стороны как изменяемые строки, с другой — как файловые объекты. В большинстве случаев вы можете использовать эти объекты вместо строк. Например, с помощью модуля re Вы можете осуществлять поиск в отображенном в память файле. Вы можете вносить изменения в файл как в изменяемую последовательность (например, '*mapped_file*[*i*] = *char*' или '*mapped_file*[*i*:*j*] = *string*') различными способами, если эти изменения не требуют изменения размера файла.

Модуль определяет следующий конструктор:

mmap(*fd*, *size* [, **platform_args*])
Отображает в память *size* байт файла, соответствующего файловому дескриптору *fd*. Если Вы хотите отобразить файл, соответствующий файловому объекту, — используйте его метод fileno().

Назначение дополнительных аргументов (*platform_args*) зависит от используемой платформы. В Windows может быть указано имя создаваемого отображения (повторное использование имени приведет к использованию ранее созданного отображения). В UNIX могут быть указаны два дополнительных аргумента: *flags* и *prot*. Аргумент *flags* может иметь значение MAP_SHARED (используется по умолчанию) или MAP_PRIVATE и определяет характер отображения. В качестве аргумента *prot* может использоваться комбинация из констант PROT_* (по умолчанию используется PROT_READ | PROT_WRITE), определяющих желаемую степень защиты памяти.

В качестве значений аргументов, характерных для UNIX, Вы можете использовать следующие константы (наиболее важные):

MAP_SHARED
Отображение является общим для всех процессов.

MAP_PRIVATE
Отображение является частным. При внесении изменений создается копия. Таким образом, все изменения видны только текущему процессу.

PROT_READ
Обеспечивает доступ к отображению на чтение.

PROT_WRITE
Обеспечивает доступ к отображению на запись.

Объекты, представляющие отображенный в память файл, имеют следующие методы:

close()
Закрывает отображение. Последующие операции над объектом приведут к ошибке.

find(*string* [, *start*])
Возвращает наименьший индекс, указывающий, где найдена подстрока *string*. Аргумент *start* является индексом, указывающим с какого места необходимо производить поиск (по умолчанию он равен 0).

flush([*offset*, *size*])
Сбрасывает на диск изменения, внесенные в отображение файла. Без вызова этого метода не гарантируется, что внесенные изменения будут записаны до того, как объект отображения будет уничтожен. Если указаны аргументы *offset* и *size*, то сбрасываются только изменения в диапазоне (в байтах) от *offset* до *offset + size*, в противном случае сбрасывается все содержимое.

move(*dest*, *src*, *count*)
Копирует *count* байт из диапазона, начинающегося с индекса *src*, в диапазон, начинающийся с индекса *dest*.

read_byte()
Считывает и возвращает один символ из отображенного файла, то есть возвращает строку с одним символом в текущей позиции отображенного файла и перемещает указатель на 1.

readline()
Считывает и возвращает строку из отображенного файла, начиная с текущей позиции, до символа перехода на новую строку (включая его) или конца отображения.

resize(*newsize*)
Изменяет размер отображенного в память файла. Размер отображения становится равным *newsize* байт.

seek(*pos* [, *whence*])
Устанавливает текущую позицию в файле. Необязательный аргумент *whence* указывает на точку отсчета: 0 (по умолчанию) — начало файла, 1 — текущая позиция и 2 — конец файла.

size()
Возвращает размер файла в байтах, который может быть больше размера отображенной в память области.

tell()
Возвращает текущую позицию в файле.

write(*string*)
Записывает строку *string* в отображенный файл, начиная с текущей позиции. После этого указатель устанавливается на следующий после записанной строки байт.

write_byte(*byte*)
Записывает один символ *byte* в отображенный файл (перемещает указатель на 1).

Глава 22

Средства организации многопоточных программ

thread	Создание нескольких потоков и управление ими.
threading	Средства высокого уровня организации потоков.
Queue	Синхронизированные очереди.

22.1 thread — создание нескольких потоков и управление ими

Модуль thread предоставляет средства низкого уровня для работы с несколькими потоками, совместно использующими глобальные данные. Для синхронизации модуль предоставляет простые средства блокировки.

error
: Исключения этого класса генерируются в случае возникновения ошибок, специфичных для этого модуля.

LockType
: Объект типа для объектов блокировки.

start_new_thread(*function*, *args* [, *kwargs*])
: Создает новый поток и выполняет в нем функцию *function*, используя в качестве позиционных и именованных аргументов *args* (кортеж) и *kwargs* (словарь).

exit()
exit_thread()
: Генерирует исключение SystemExit. Если исключение (в данном потоке) не обрабатывается, выполнение потока будет молча завершено. Имя exit_thread() определено для совместимости со старыми версиями.

allocate_lock()
: Возвращает новый объект, реализующий блокировку. Изначально объект не является заблокированным.

get_ident()
Возвращает идентификатор текущего потока (отличное от нуля целое число). После завершения работы потока его идентификатор может быть использован повторно для нового потока.

Объекты, возвращаемые функцией allocate_lock, имеют следующие методы:

acquire([*waitflag*])
При вызове без аргументов захватывает (блокирует) объект, дождавшись, если это необходимо, его освобождения другим потокам (возвращает None). Если задан аргумент *waitflag* (целое число), поведение функции зависит от его значения: 1 — функция ожидает освобождения объекта другим потокам (так же, как и с опущенным аргументом), 0 (и другие значения) — объект захватывается только, если это может быть сделано немедленно. При наличии аргумента функция возвращает 1, если захват объекта прошел успешно, в противном случае возвращает 0.

release()
Освобождает объект. Объект должен быть захвачен ранее, но не обязательно в этом же потоке.

locked()
Возвращает текущий статус объекта: 1, если объект захвачен (заблокирован), иначе — 0.

Замечания:

- Если доступен модуль signal, прерывание (исключение KeyboardInterrupt) получает основной поток. Однако, если модуль signal недоступен, прерывание может получить любой поток.
- Вызов функций sys.exit() и thread.exit() и генерация исключения SystemExit эквивалентны.
- Не все функции, реализованные на языке C, при ожидании ввода/вывода позволяют выполняться другим потокам, но наиболее популярные (функции time.sleep() и select.select(), метод read() файловых объектов) работают правильно.
- Невозможно прервать выполнение метода acquire() объектов, предназначенных для блокировки, — исключение KeyboardInterrupt будет сгенерировано только после захвата объекта.
- Поведение при завершении работы основного потока раньше других зависит от платформы. Обычно остальные потоки немедленно завершают свою работу без выполнения ветвей finally инструкций try и вызова деструкторов объектов.

22.2 threading — средства высокого уровня организации потоков

Модуль threading предоставляет средства высокого уровня для работы с несколькими потоками, реализованные поверх модуля thread. Этот модуль безопасен для импортирования с помощью инструкции 'from threading import *'.

activeCount()
  Возвращает число объектов, представляющих активные потоки. Это значение равно длине списка, возвращаемого функцией enumerate().

Condition([*lock*])
  Создает и возвращает объект, реализующий условие. Этот объект позволяет одному или нескольким потокам ожидать уведомления от другого потока. Если задан аргумент *lock*, он должен быть объектом, возвращаемым функцией Lock() или RLock(). Этот объект будет использован для блокировки. По умолчанию создается новый объект с помощью функции RLock().

currentThread()
  Возвращает объект, соответствующий текущему потоку. Если текущий поток не был создан с помощью модуля threading, возвращаемый объект будет иметь ограниченную функциональность.

enumerate()
  Возвращает список объектов, соответствующих активным потокам, созданным с помощью модуля threading (в том числе и фоновые). Список всегда включает объекты для текущего и основного потока и не включает объекты, соответствующие потокам, которые уже завершили или еще не начали свое выполнение.

Event()
  Создает и возвращает объект, реализующий событие. Этот объект позволяет потоку ожидать наступления (установки в другом потоке) события.

Lock()
  Создает и возвращает объект, реализующий примитивную блокировку. В текущих реализациях является псевдонимом для функции thread.allocate_lock().

RLock()
  Создает и возвращает объект, реализующий блокировку. В отличие от объекта, возвращаемого функцией Lock(), допускает повторный захват в том же потоке. Объект должен быть освобожден тем же потоком ровно столько раз, сколько он был захвачен.

Semaphore([*count*])
  Создает и возвращает объект, реализующий семафор. Семафор может быть захвачен не более *count* раз (по умолчанию 1). При последующих попытках захватить семафор ожидает, пока другой поток не освободит его.

Thread([*keyword_list*])

Этот класс используется для представления потока. Конструктор воспринимает несколько именованных аргументов (*keyword_list*). Действия, выполняемые при запуске потока, могут быть установлены с помощью аргумента с именем target — должен быть объектом, поддерживающим вызов. Этот объект вызывается из метода run() с позиционными и именованными аргументами, заданными соответственно аргументами с именами args и kwargs (по умолчанию используются пустые кортеж и словарь). Аргумент с именем name указывает имя потока; по умолчанию используется 'Thread-*N*', где *N* — десятичное представление небольшого целого числа.

Интерфейсы объектов описаны в следующих разделах.

22.2.1 Объекты, реализующие блокировку

Функции Lock() и RLock() возвращают объекты, имеющие одинаковый набор методов:

acquire([*waitflag*])

При вызове без аргументов захватывает (блокирует) объект, дождавшись, если это необходимо, его освобождения другим потокам (возвращает None). Если задан аргумент *waitflag* (целое число), поведение функции зависит от его значения: 1 — функция ожидает освобождения объекта другим потокам (так же, как и с опущенным аргументом), 0 (и другие значения) — объект захватывается только, если это может быть сделано немедленно. При наличии аргумента функция возвращает 1, если захват объекта прошел успешно, в противном случае возвращает 0.

release()

Освобождает объект. Объект должен быть захвачен ранее. Если метод применяется к объекту, возвращаемому функцией RLock(), объект должен быть захвачен этим же потоком.

В то время как объекты, возвращаемые функцией Lock(), реализуют примитивную блокировку (объект может быть захвачен один раз), для объектов, возвращаемых функцией RLock(), метод acquire() может быть вызван многократно одним потоком — для того, чтобы объект мог быть захвачен другим потоком, необходимо ровно столько же раз вызвать метод release().

22.2.2 Условия

Объекты, возвращаемые функцией Condition() всегда ассоциированы с объектом, реализующим блокировку. Такой объект может быть указан в качестве аргумента при инициализации или, по умолчанию, будет создан конструктором.

Объект, представляющий условие, имеет методы `acquire()` и `release()`, которые вызывают соответствующие методы ассоциированного объекта, реализующего блокировку. Он также имеет методы `wait()`, `notify()` и `notifyAll()`, которые должны вызываться только после того, как будет вызван метод `acquire()`. При вызове метода `wait()` объект-условие освобождает ассоциированный объект и блокирует выполнение до тех пор, пока из другого потока не будет вызван метод `notify()` или `notifyAll()`. После пробуждения метод снова захватывает ассоциированный объект. Вы можете указать максимальное время ожидания.

Методы `notify()` и `notifyAll()` не освобождают ассоциированный объект, реализующий блокировку. Таким образом, поток или потоки, вызвавшие метод `wait()`, будут пробуждены только после того, как поток, вызвавший `notify()` или `notifyAll()` окончательно освободит ассоциированный объект.

Приведем простой пример с производством и потреблением объектов. Пусть функция `make_an_item_available()` производит объект, `get_an_available_item()` его потребляет и `an_item_is_available()` возвращает истину, если есть в наличии произведенные (но еще не употребленные) объекты. Тогда следующие фрагменты кода могут быть использованы в потоках производителя и потребителя соответственно:

```
# Производим один объект
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()

# Потребляем один объект
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()
```

acquire([...])
release()
: Вызывают одноименные методы ассоциированного объекта, реализующего блокировку.

wait([*timeout*])
: Освобождает ассоциированный объект, реализующий блокировку, ожидает, пока другим потоком не будет вызван метод `notify()` или `notifyAll()`, и снова захватывает ассоциированный объект. Если задан и не равен `None` аргумент *timeout* (вещественное число), он указывает максимальное время ожидания в секундах. По умолчанию время ожидания не ограничено.

  Если для блокировки используется объект, возвращаемый функцией `RLock()`, этот метод использует его детали реализации для полного освобождения и восстановления уровня захвата объекта.

notify()
: Пробуждает один поток, ожидающий выполнения условия, если таковой имеется.

notifyAll()
: Пробуждает все потоки, ожидающие выполнения условия.

22.2.3 Семафоры

Объекты, возвращаемые функцией `Semaphore()` имеют такой же набор методов, как и объекты, реализующие блокировку (см. раздел 22.2.1). Семафор управляет внутренним счетчиком, который уменьшается при каждом вызове метода `acquire()` и увеличивается при каждом вызове метода `release()`. Начальное значение счетчика определяет аргумент, переданный конструктору `Semaphore()`. Значение счетчика не может стать меньше нуля: вызов метода `acquire()` при значении счетчика равным нулю блокирует выполнение до тех пор, пока значение не увеличится, то есть пока другой поток не вызовет метод `release()`.

22.2.4 События

Объекты, возвращаемые функцией `Event()` реализуют простейший механизм взаимодействия потоков: один поток подает сигнал о возникновении события и один или более потоков ожидают его. Объект-событие имеет внутренний флаг. Если этот флаг является истиной, событие считается наступившим. Изначально (при создании объекта) флаг является ложью.

isSet()
: Возвращает `1`, если событие наступило (внутренний флаг является истиной).

set()
: Устанавливает наступление события (внутренний флаг становится истинным). Все потоки, ожидающие события, пробуждаются.

clear()
: Отменяет наступление события (внутренний флаг становится ложным).

wait([*timeout*])
: Ожидает наступления события, то есть, если внутренний флаг является ложью, блокирует выполнение потока до тех пор, пока другой поток не вызовет метод `set()`. Если задан и не равен `None` аргумент *timeout* (вещественное число), он указывает максимальное время ожидания в секундах. По умолчанию время ожидания не ограничено.

22.2.5 Объекты, представляющие потоки

Экземпляры класса `Thread` представляют действия, выполняемые в отдельном потоке. Существует два способа определить эти действия: передача объекта, поддерживающего вызов, в качестве аргумента с именем `target` или переопределение метода `run()` в производных классах. В производных классах не следует переопределять другие методы, за исключением, возможно, конструктора. Если Вы переопределяете конструктор, не забудьте вызвать конструктор дочернего класса (`Thread.__init__()`) до того, как Вы будете что-либо делать с потоком.

После создания объекта, представляющего поток, Вы можете запустить его с помощью метода `start()`. После запуска поток считается активным до тех пор, пока не закончится выполнение метода `run()`.

Каждый поток, реализованный с помощью `Thread` или производного от него класса имеет имя. Имя может быть задано при инициализации или с помощью метода `setName()` и извлечено с помощью метода `getName()`.

Поток может "присоединиться" к другому потоку, вызвав метод `join()` объекта, представляющего этот поток. При этом выполнение потока, вызвавшего метод `join()` приостанавливается до тех пор, пока не завершится выполнение потока, к объекту которого был применен метод.

Поток может быть помечен как фоновый (daemon). Выполнение всей программы будет завершено только после того, как активными останутся только фоновые потоки. Начальное значение наследуется от текущего потока. Этот флаг может быть изменен с помощью метода `setDaemon()` и извлечен с помощью метода `getDaemon()`.

Существует также объект, соответствующий основному потоку. Основной поток не является фоновым. Возможно также создание объектов, соответствующих "посторонним" потокам, то есть потокам, созданным без помощи модуля `threading`. Такие объекты имеют ограниченную функциональность. Они всегда считаются активными, фоновыми и к ним нельзя присоединиться с помощью метода `join()`.

start()
: Запускает выполнение задачи (метод `run()` объекта) в отдельном потоке. Этот метод может быть вызван не более одного раза.

run()
: Этот метод представляет задачу, которая должна быть выполнена в отдельном потоке. Вы можете переопределить этот метод в производном классе. Стандартный метод `run()` вызывает функцию, указанную при инициализации объекта (аргумент с именем `target`).

join([*timeout*])
: Этот метод приостанавливает работу текущего потока (который вызвал метод) до окончания выполнения потока, к объекту которого метод применяется. Если задан и не равен `None` аргумент *timeout* (вещественное число), он указывает макси-

мальное время в секундах ожидания завершения работы потока. По умолчанию (и если аргумент *timeout* равен None) время ожидания не ограничено.

К одному потоку можно присоединиться несколько раз. Поток не может присоединиться к самому себе, так как это привело бы к (мертвой) блокировке самого себя. К потоку нельзя присоединиться до того, как он будет запущен.

getName()
: Возвращает имя потока.

setName(*name*)
: Устанавливает имя потока. Имя потока является строкой, которая может быть использована для его идентификации. Несколько потоков могут иметь одинаковое имя. Начальное имя устанавливается конструктором.

isAlive()
: Возвращает 1, если поток активен. Поток считается активным с момента вызова метода start() и до завершения выполнения задачи (метода run()).

isDaemon()
: Возвращает значение внутреннего флага, который является истиной, если поток выполняется в фоновом режиме.

setDaemon(*daemonic*)
: Устанавливает значение внутреннего флага, определяющего, в каком режиме выполняется поток, равным *daemonic*: если флаг является истиной, поток будет выполняться в фоновом режиме. Этот метод должен вызываться до запуска потока. Начальное значение наследуется от потока, создавшего данный поток.

Работа всей программы может быть завершена только после завершения работы всех нефоновых потоков.

22.3 Queue — синхронизированные очереди

Модуль Queue определяет класс, реализующий очередь (FIFO), доступ к которой может осуществляться из нескольких потоков. Доступность этого модуля зависит от наличия модуля thread.

Queue(*maxsize*)
: Создает и возвращает объект-очередь. Аргумент *maxsize* (целое число) определяет максимальное количество элементов, которое можно поместить в очередь. По умолчанию, а также если *maxsize* меньше или равен нулю, размер очереди не ограничен.

Empty
: Исключения этого класса генерируются при попытке извлечь из пустой или заблокированной очереди элемент с помощью метода get_nowait() (или get() с аргументом *block* равным нулю).

Full

Исключения этого класса генерируются при попытке добавить в полную или заблокированную очередь элемент с помощью метода `put_nowait()` (или `put()` с аргументом *block* равным нулю).

Экземпляры класса `Queue` имеют следующие методы:

qsize()

Возвращает количество элементов в очереди (в момент вызова метода).

empty()

Возвращает `1`, если очередь (в момент вызова метода) пуста, иначе возвращает `0`.

full()

Возвращает `1`, если очередь (в момент вызова метода) содержит максимальное количество элементов, иначе возвращает `0`.

put(*item* [, *block*])

Добавляет элемент в очередь. Если аргумент *block* опущен или является истиной, выполнение потока при необходимости приостанавливается до тех пор, пока в очереди не появится свободное место. Если аргумент *block* является ложью, элемент добавляется в очередь только, если это можно сделать немедленно, в противном случае генерируется исключение `Full`.

put_nowait(*item*)

Эквивалентно вызову `put(item, 0)`.

get([*block*])

Извлекает (удаляет) элемент из очереди и возвращает его. Если аргумент *block* опущен или является истиной, выполнение потока при необходимости приостанавливается до тех пор, пока в очереди не появится элемент. Если аргумент *block* является ложью, элемент извлекается из очереди только, если это можно сделать немедленно, в противном случае генерируется исключение `Empty`.

get_nowait()

Эквивалентно вызову `get(0)`.

Не следует полагаться на значения, возвращаемые методами `qsize()`, `empty()` и `full()`: к моменту их получения ситуация может уже измениться.

Глава 23

Работа с базами данных

Модули, описанные в этой главе, предоставляют доступ к простым базам данных, реализуя интерфейс, аналогичным словарям. Во всех случаях в качестве ключа и значения могут использоваться только обычные строки[1]. Если Вы хотите хранить объекты языка Python другого типа, воспользуйтесь модулем shelve.

anydbm Универсальный интерфейс к базам данных в стиле DBM.

dumbdbm Медленная, но переносимая реализация интерфейса DBM.

dbhash Интерфейс в стиле DBM к библиотеке баз данных BSD.

dbm Интерфейс к библиотеке (n)dbm.

gdbm Интерфейс к библиотеке gdbm.

whichdb Определяет, какой модуль с интерфейсом DBM был использован для создания файла базы данных.

bsddb Интерфейс к библиотеке баз данных BSD.

23.1 Интерфейс к базам данных в стиле DBM

Модуль anydbm предоставляет универсальный интерфейс в стиле DBM к различным вариантам баз данных: dbhash (требует наличие модуля bsddb), gdbm или dbm. Если ни один из этих модулей не доступен, будет использована медленная, но полностью переносимая реализация на языке Python в модуле dumbdbm.

Модуль dbhash доступен в операционных системах UNIX и Windows и предоставляет интерфейс в стиле DBM к библиотеке BSD db. Для работы этого модуля необходим модуль bsddb.

Модули dbm и gdbm доступны только в операционных системах UNIX.

Заметим, что форматы файлов, создаваемых различными модулями не совместимы друг с другом.

[1] Реализация некоторых из приведенных здесь модулей такова, что они позволяют использовать любые объекты с интерфейсом буфера. Однако использование этой возможности приведет к непереносимости базы данных, а также может привести к конфликтам ключей.

23.1.1 Общая для всех модулей часть интерфейса

Все перечисленные выше модули определяют функцию-конструктор и исключение:

open(*filename* [, *flag* [, *mode*]])

Открывает файл с именем *filename* базы данных и возвращает соответствующий ему объект. Если файл уже существует, функция `anydbm.open()` использует для определения его типа модуль `whichdb` и для файла используется соответствующий модуль. При создании нового файла `anydbm.open()` использует первый модуль из перечисленных выше, который может быть импортирован.

Аргумент *flag* может иметь одно из следующих значений:

`'r'`
: Открыть существующий файл только для чтения (используется по умолчанию).

`'w'`
: Открыть существующий файл для чтения и записи.

`'c'`
: Открыть файл для чтения и записи. Если файл не существует, он будет создан.

`'n'`
: Создать новую пустую базу данных (старый файл перезаписывается) и открыть ее для чтения и записи.

В функции `dbhash.open()` на платформах, поддерживающих блокировку, к флагу может быть добавлен символ '`l`', если блокировка должна быть использована. В функции `gdbm.open()` к флагу может быть добавлен символ '`f`' для открытия файла в "быстром" режиме: в этом случае измененные данные не будут автоматически записываться в файл при каждом внесении изменений (см. также описание метода `sync()` в разделе 23.1.3).

Аргумент *mode* указывает биты разрешений, которые будут установлены для файла, принимая во внимание маску разрешений (umask). По умолчанию используется `0666`.

error

Класс исключений, которые генерируются в случае возникновения различных ошибок, характерных для данного модуля. Заметим, что если в базе данных отсутствует запись с определенным ключом, генерируется исключение `KeyError`.

Имя `dbhash.error` является псевдонимом для класса `bsddb.error`.

В модуле `anydbm` эта переменная ссылается на кортеж классов исключений, которые могут быть сгенерированы непосредственно модулем `anydbm` (первый элемент) или одним из модулей, реализующим доступ к базе данных. Это не вносит каких-либо неудобств, так как кортежи с классами исключений, указанные после ключевого слова `except`, могут быть вложенными произвольным образом.

Объекты, возвращаемые функциями `open()` всех перечисленных модулей, поддерживают большинство операций, характерных для отображений: извлечение, установка и

удаление значений по ключу, а также методы `has_key()` и `keys()`. Ключи и значения должны быть обычными строками. Кроме того, эти объекты имеют еще один метод:

close()
Закрывает файл с базой данных.

Объекты, возвращаемые функциями `dbhash.open()` и `gdbm.open()`, имеют дополнительные методы, описанные ниже.

23.1.2 Дополнительные методы объектов, возвращаемых функцией `dbhash.open()`

first()
Возвращает кортеж с ключом и значением первой записи в базе данных. Если база данных не содержит ни одной записи, генерируется исключение `KeyError`. Используя методы `first()` и `next()` Вы можете обойти все записи в базе данных.

next()
Возвращает кортеж с ключом и значением следующей записи. Например, следующий код выводит все записи, содержащиеся в базе данных, не создавая их список в памяти:

```
k, v = db.first()
try:
    print k, v
    k, v = db.next()
except KeyError:
    pass
```

sync()
Сбрасывает на диск все не записанные данные.

23.1.3 Дополнительные методы объектов, возвращаемых функцией `gdbm.open()`

firstkey()
Возвращает ключ первой записи в базе данных. Вы можете перебрать все записи базы данных с помощью методов `firstkey()` и `nextkey()`.

nextkey(*key*)
Возвращает ключ к записи, следующей за записью с ключом *key*. Следующий пример выводит ключи для всех записей, не создавая списка ключей в памяти:

```
k = db.firstkey()
while k != None:
    print k
    k = db.nextkey(k)
```

reorganize()

Реорганизует файл базы данных, удаляя из него неиспользуемые участки. Может быть полезен после удаления большого количества записей (`reorganize()` — единственный метод, который может привести к уменьшению размера файла).

sync()

Сбрасывает на диск все не записанные данные.

23.2 whichdb — определение формата файла базы данных

Этот модуль определяет единственную функцию, которая пытается определить, какой модуль должен быть использован для открытия базы данных: `dbm`, `gdbm` или `dbhash`.

whichdb(*filename*)

Возвращает `None` — если файл *filename* отсутствует или к нему нет доступа на чтение, пустую строку (`''`) — если формат файла неизвестен. В остальных случаях возвращает строку с именем модуля (например, `'gdbm'`), который должен быть использован для открытия файла.

23.3 bsddb — интерфейс к библиотеке баз данных BSD

Модуль `bsddb` предоставляет интерфейс к библиотеке BSD `db`[2]. С помощью этого модуля Вы можете создавать и работать с файлами баз данных типа “Hash”, “BTree” и “Record”. Объекты, возвращаемые функциями модуля, ведут себя аналогично словарям.

Первые три аргумента всех функций одинаковы: *filename* — имя файла базы данных, *flag* — флаги, определяющие способ открытия файла, и *mode* — биты разрешений (по умолчанию используется `0666`), которые будут установлены для файла, принимая во внимание маску разрешений (umask). Аргумент *flag* может иметь одно из следующих значений:

`'r'`

Открыть существующий файл только для чтения (используется по умолчанию).

`'w'`

Открыть существующий файл для чтения и записи.

`'c'`

Открыть файл для чтения и записи. Если файл не существует, он будет создан.

[2] Обычно используется библиотека версии 1.85. Начиная с Python 2.0, версия библиотеки может быть указана при сборке. Следует помнить, что версии 1.85 и 2.0 библиотеки `db` несовместимы.

'n'
: Создать новую пустую базу данных (старый файл перезаписывается) и открыть ее для чтения и записи.

На платформах, поддерживающих блокировку, к флагу может быть добавлен символ 'l', если блокировка должна быть использована.

Остальные аргументы используются достаточно редко, Вы можете узнать об их назначении из документации к функции dbopen() библиотеки BSD db.

hashopen(*filename* [, *flag* [, *mode* [, *bsize* [, *ffactor* [, *nelem* [, *cachesize* [, *hash* [, *lorder*]]]]]]]])
: Открывает файл базы данных в формате "Hash" и возвращает соответствующий ему объект.

btopen(*filename* [, *flag* [, *mode* [, *btflags* [, *cachesize* [, *maxkeypage* [, *minkeypage* [, *psize* [, *lorder*]]]]]]]])
: Открывает файл базы данных в формате "BTree" и возвращает соответствующий ему объект.

rnopen(*filename* [, *flag* [, *mode* [, *rnflags* [, *cachesize* [, *psize* [, *lorder* [, *reclen* [, *bval* [, *bfname*]]]]]]]]])
: Открывает файл базы данных в формате "Record" и возвращает соответствующий ему объект.

error
: Класс исключений, которые генерируются в случае возникновения различных ошибок, характерных для этого модуля. Заметим, что если в базе данных отсутствует запись с определенным ключом, генерируется исключение KeyError.

Объекты, возвращаемые функциями hashopen(), btopen() и rnopen(), поддерживают большинство операций, характерных для отображений: извлечение, установка и удаление значений по ключу, а также методы has_key() и keys() (ключи и значения должны быть обычными строками). Кроме того, доступны следующие методы:

close()
: Закрывает файл с базой данных.

set_location(*key*)
: Устанавливает указатель на запись с ключом *key* и возвращает кортеж, содержащий ключ и значение для этой записи. Этот метод не может быть применен к объектам, созданным с помощью функции hashopen().

first()
: Возвращает кортеж с ключом и значением первой записи в базе данных. Если база данных не содержит ни одной записи, генерируется исключение KeyError. Используя методы first() и next() Вы можете обойти все записи в базе данных.

next()

Возвращает кортеж с ключом и значением следующей записи. Например, следующий код выводит все записи, содержащиеся в базе данных, не создавая их списка в памяти:

```
k, v = db.first()
try:
    print k, v
    k, v = db.next()
except KeyError:
    pass
```

last()

Возвращает кортеж с ключом и значением последней записи в базе данных. Если база данных не содержит ни одной записи, генерируется исключение `KeyError`. Используя методы `last()` и `previous()` Вы можете обойти все записи в базе данных в обратном порядке. Этот метод не может быть применен к объектам, созданным с помощью функции `hashopen()`.

previous()

Возвращает кортеж с ключом и значением предыдущей записи. Этот метод не может быть применен к объектам, созданным с помощью функции `hashopen()`.

sync()

Сбрасывает на диск все не записанные данные.

Пример:

```
>>> import bsddb
>>> db = bsddb.btopen('/tmp/spam.db', 'c')
>>> for i in xrange(10): db['%d' % i] = '%d' % (i*i)
...
>>> db['3']
'9'
>>> db.keys()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> db.first()
('0', '0')
>>> db.next()
('1', '1')
>>> db.last()
('9', '81')
>>> db.set_location('2')
('2', '4')
>>> db.previous()
('1', '1')
>>> db.sync()
0
```

Глава 24

Сжатие данных

zlib Интерфейс низкого уровня к операциям сжатия, с использованием алгоритма, совместимого с **gzip**.

gzip Работа с файлами, сжатыми программой **gzip**.

zipfile Чтение и запись **zip**-архивов.

24.1 zlib — алгоритм сжатия, совместимый с gzip

Функции, определенные в этом модуле, позволяют упаковывать (сжимать) и распаковывать данные с помощью библиотеки zlib (http://www.info-zip.org/pub/infozip/zlib/, RFC 1950). Версия используемой библиотеки zlib языка C определена в виде константы:

ZLIB_VERSION
Строка с версией библиотеки.

Аргумент *level* в описанных ниже функциях должен быть равным Z_DEFAULT_COMPRESSION, 0 (Z_NO_COMPRESSION) или целым числом от 1 (Z_BEST_SPEED) до 9 (Z_BEST_COMPRESSION) и определяет уровень сжатия:

Z_NO_COMPRESSION
Сохранить данные без сжатия.

Z_BEST_SPEED
Обеспечивает максимальную скорость при меньшем сжатии.

Z_BEST_COMPRESSION
Обеспечивает наилучшее сжатие с меньшей скоростью.

Z_DEFAULT_COMPRESSION
Использует уровень, обеспечивающий компромисс между скоростью и степенью сжатия. В текущей версии библиотеки соответствует уровню 6.

Аргумент *wbits* (в текущей версии может быть целым числом от 8 до 15) определяет размер окна буфера истории равным 2***wbits*. Большее значение обеспечивает

лучшее сжатие при больших затратах памяти. При использовании отрицательных значений подавляется стандартный заголовок (эта недокументированная возможность позволяет использовать модуль `zlib` для работы с **zip**-файлами). Использование значения отличного от `MAX_WBITS` при распаковке может привести к возникновению ошибки. По умолчанию всегда используется значение `MAX_WBITS`.

Текущая версия библиотеки позволяет использовать единственный метод (аргумент *method*) сжатия:

Z_DEFLATED
Метод DEFLATE (см. RFC 1951).

Модуль определяет следующие исключение и функции:

error
Исключения этого класса генерируются в случае возникновения ошибок при упаковке или распаковке.

adler32(*string* [, *value*])
Вычисляет контрольную сумму Adler-32 строки *string*. Контрольная сумма, вычисленная по этому алгоритму практически настолько же надежна, как и CRC32, но вычисляется гораздо быстрее. Если задано число *value*, оно используется в качестве начального значения контрольной суммы, в противном случае используется фиксированное значение по умолчанию. Это позволяет вычислять контрольную сумму длинного потока по частям. Алгоритм Adler-32 недостаточно сильный, чтобы использовать его в криптографических целях.

compress(*string* [, *level*])
Возвращает строку, полученную в результате упаковки строки *string*.

compressobj([*level* [, *method* [, *wbits* [, *mem_level* [, *strategy*]]]]])
Возвращает объект, реализующий упаковку потока данных. Аргумент *mem_level* определяет использование памяти: `1` — использовать минимум памяти, но что сильно замедляет процесс, `9` — использовать максимум памяти, что дает максимальную скорость. По умолчанию используется `DEF_MEM_LEVEL` (`8`). Аргумент *strategy* может иметь значение `Z_DEFAULT_STRATEGY`, `Z_FILTERED` или `Z_HUFFMAN_ONLY` и определяет стратегию сжатия (см. документацию к библиотеке `zlib` языка C).

crc32(*string* [, *value*])
Вычисляет контрольную сумму CRC32 (Cyclic Redundancy Check) строки *string*. Если задано число *value*, оно используется в качестве начального значения контрольной суммы, в противном случае используется фиксированное значение по умолчанию. Это позволяет вычислять контрольную сумму длинного потока по частям. Алгоритм CRC32 недостаточно сильный, чтобы использовать его в криптографических целях.

decompress(*string* [, *wbits* [, *bufsize*]])
Возвращает строку, полученную в результате распаковки строки *string*. Аргумент *bufsize* определяет начальный размер буфера вывода (при необходимости буфер будет увеличен).

decompressobj([*wbits*])
Возвращает объект, реализующий распаковку потока данных. Аргумент *wbits* имеет такое же значение, как и для функции decompress().

Объекты, реализующие упаковку потока данных, имеют следующие методы:

compress(*string*)
Упаковывает строку *string* и возвращает строку с упакованными данными как минимум для части строки *string*. Часть данных может быть оставлена во внутреннем буфере для дальнейшей обработки. Возвращаемая строка должна быть добавлена к строкам, полученным в результате предыдущих вызовов метода.

flush([*mode*])
Обрабатывает содержимое внутреннего буфера и возвращает строку с упакованным остатком данных.

В качестве аргумента *mode* метода flush() может быть использована одна из следующих констант:

Z_SYNC_FLUSH
Обрабатывает содержимое внутреннего буфера. Данные могут быть использованы для распаковки до текущего места.

Z_FULL_FLUSH
Аналогично Z_SYNC_FLUSH, но также сбрасывает состояние. Распаковка может быть перезапущена с этого уровня, если предыдущие сжатые данные оказались поврежденными или необходим произвольный доступ к сжатым данным.

Z_FINISH
Полностью завершает сжатие, используется по умолчанию. После этого объект не может быть использован для упаковки данных.

Объекты, реализующие распаковку потока данных, имеют следующие атрибут данных и методы:

unused_data
Строка с неиспользованными данными. Должна быть пустой, если объекту была передана вся строка с упакованными данными.

decompress(*string*)
Распаковывает строку *string* и возвращает строку с распакованными данными как минимум для части строки *string*. Часть данных может быть оставлена во внутреннем буфере для дальнейшей обработки. Возвращаемая строка должна быть добавлена к строкам, полученным в результате предыдущих вызовов метода.

flush()
: Обрабатывает содержимое внутреннего буфера и возвращает строку с распакованным остатком данных. После вызова этого метода объект не может быть использован для распаковки данных.

24.2 gzip — работа с файлами, сжатыми программой gzip

Метод сжатия, предоставляемый модулем zlib, совместим с методом, используемым программой GNU **gzip**. Модуль gzip реализован поверх модуля zlib и предоставляет класс GzipFile, реализующий чтение и запись файлов в **gzip**-формате (RFC 1952). Упаковка и распаковка данных происходит автоматически, так что экземпляры класса GzipFile ведут себя аналогично обычным файловым объектам. Заметим, что этот модуль не позволяет работать с файлами дополнительных форматов (созданных программами **compress** и **pack**), которые могут быть распакованы программой **gzip** (**gunzip**).

GzipFile(*filename* [, *mode* [, *compresslevel* [, *fileobj*]]])
: Возвращает объект аналогичный файловому, обеспечивающий прозрачную упаковку и распаковку данных. Экземпляры класса GzipFile имеют большинство методов, характерных для файловых объектов, кроме seek() и tell().

Если задан и не равен None аргумент *fileobj* (файловый или подобный объект), он будет использован для чтения/записи (аргумент *filename* в этом случае будет использован только в заголовке **gzip**-файла и может быть равен None), иначе будет открыт файл с именем *filename*.

Аргумент *mode* должен иметь одно из трех значений: 'rb', 'ab' или 'wb'. По умолчанию используется 'rb'.

Уровень сжатия устанавливается аргументом *compresslevel*, который должен быть целым числом от 1 до 9. По умолчанию используется значение 9, обеспечивающее максимальную степень сжатия.

Вызов метода close() экземпляра класса GzipFile не закрывает файловый объект *fileobj*. Это позволяет добавить дополнительные данные, а также извлечь результат, используя метод getvalue() экземпляра класса StringIO.

open(*filename* [, *mode* [, *compresslevel*]])
: Эквивалентно вызову 'GzipFile(*filename*, *mode*, *compresslevel*)'.

24.3 zipfile — работа с zip-архивами

Этот модуль доступен, начиная с версии 1.6, и позволяет записывать и читать **zip**-архивы.

error
: Исключения этого класса генерируются, если указанный файл не является **zip**-архивом или поврежден.

is_zipfile(*path*)
: Возвращает 1, если *path* является **zip**-архивом, иначе возвращает 0. Текущая реализация модуля не позволяет работать с архивами, содержащими комментарий.

ZipFile(*filename* [, *mode* [, *compression*]])
: Возвращает объект, реализующий автоматическую упаковку/распаковку при чтении/записи файла с именем *filename*. Аргумент *mode* может иметь одно из трех значений: 'r' (чтение), 'w' (запись в новый архив) или 'a' (добавление к существующему архиву). По умолчанию используется 'r'. В качестве аргумента *compression* может быть использована одна из приведенных ниже констант, при попытке использования недоступного метода генерируется исключение RuntimeError.

PyZipFile(*filename* [, *mode* [, *compression*]])
: Этот класс является производным от класса ZipFile и реализует один дополнительный метод, позволяющий создавать архивы файлов библиотек и пакетов языка Python.

ZipInfo([*filename* [, *date_time*]])
: Создает и возвращает объект, описывающий файл в архиве. Аргумент *filename* определяет имя файла (по умолчанию используется 'NoName'), *date_time* — дату и время создания файла (кортеж из целых чисел: год, месяц, число, час, минута, секунда).

В настоящий момент модуль поддерживает только два метода сжатия (аргумент *compression* конструктора ZipFile):

ZIP_STORED
: Упаковка данных без сжатия (используется по умолчанию).

ZIP_DEFLATED
: Наиболее часто используемый в **zip**-файлах метод сжатия. Этот метод требует наличия модуля zlib.

Экземпляры классов ZipFile и PyZipFile имеют следующие атрибуты данных и методы:

namelist()
: Возвращает список имен файлов, содержащихся в архиве.

infolist()
: Возвращает список экземпляров класса ZipInfo, представляющих описание файлов, содержащихся в архиве.

printdir()
Выводит в стандартный поток вывода (`sys.stdout`) оглавление архива.

testzip()
Считывает все файлы архива и проверяет для них контрольные суммы. Возвращает имя первого файла, для которого контрольная сумма не сходится или `None`, если тест прошел успешно.

getinfo(*name*)
Возвращает экземпляр класса `ZipInfo`, представляющий описание файла *name*.

read(*name*)
Считывает файл с именем *name* из архива и возвращает его содержимое в виде строки.

write(*filename* [, *arcname* [, *compress_type*]])
Помещает файл с именем *filename* в архив под именем *arcname* (по умолчанию используется имя исходного файла) используя метод *compress_type* (по умолчанию используется метод, указанный при инициализации экземпляра).

writestr(*zinfo*, *bytes*)
Помещает файл в архив. В качестве содержимого файла используется аргумент *bytes*, информация об имени файла, времени его создания, методе сжатия и т. д. берется из объекта *zinfo*, который должен быть экземпляром класса `ZipInfo`.

close()
Закрывает файл (если архив был открыт для записи, дописывает завершающую запись).

Экземпляры класса `PyZipFile` имеют дополнительный метод:

writepy(*pathname* [, *basename*])
Если *pathname* является каталогом пакета, рекурсивно добавляет все '.py'-файлы в архив, если *pathname* является обычным каталогом — добавляет в архив файлы, находящиеся в этом каталоге, в противном случае *pathname* должен быть '.py'-файлом, который будет добавлен в архив. Все модули ('.py'-файлы) добавляются в архив в компилированном виде ('.pyo' или '.pyc').

Экземпляры класса `ZipInfo` имеют следующие (основные) атрибуты, описывающие файл в архиве, и метод:

filename
Имя файла.

date_time
Дата и время создания файла, кортеж из целых чисел: год, месяц, число, час, минута, секунда.

compress_type
Метод сжатия.

comment
Комментарий (для каждого файла).

extra
Строка дополнительных данных.

CRC
Контрольная сумма CRC32 распакованного файла.

compress_size
Размер сжатого представления файла.

file_size
Размер распакованного файла.

FileHeader()
Печатает строку информации о файле в архиве.

Глава 25

Отладка и оптимизация кода на языке Python

Средства, описанные в этой главе, позволяют отлаживать и замерять производительность кода на языке Python. Эти средства представлены двумя (основными) модулями, которые также могут быть использованы в качестве готовых программ.

pdb Средства отладки кода на языке Python.

profile Замер производительности кода на языке Python.

pstats Обработка статистических данных и вывод отчетов.

25.1 Отладчик кода на языке Python

Модуль pdb реализует интерактивный отладчик программ, написанных на языке Python. Он позволяет устанавливать точки останова, использовать пошаговое выполнение строк исходного кода, исследовать кадры стека, выводить исходный код и выполнять инструкции языка Python в контексте любого кадра стека. Этот модуль также позволяет осуществить “вскрытие трупа” программы или интерактивных инструкций, выполнение которых завершилось генерацией исключения.

Вы можете расширить возможности отладчика: все описанные в этом разделе функции реализованы с помощью класса Pdb (в настоящий момент не документирован).

Запуск под контролем отладчика производится следующим образом (в качестве приглашения отладчика используется ‘(Pdb) ’):

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: "There is no ... named 'spam'"
> <string>(1)?()
(Pdb)
```

Или Вы можете осуществить “вскрытие” уже после возникновения ошибки:

```
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "mymodule.py", line 2, in test
    print spam
NameError: There is no variable named 'spam'
>>> import pdb
>>> pdb.pm()
> mymodule.py(2)test()
-> print spam
(Pdb)
```

Файл ‘pdb.py’ может быть также использован в качестве программы для отладки других программ:

```
/usr/local/lib/python1.5/pdb.py myscript.py
```

25.1.1 Функции запуска отладчика

Модуль pdb определяет следующие функции, запускающие отладчик различными способами:

run(*statements* [, *globals* [, *locals*]])
Выполняет инструкции в строке *statements* (аналогично инструкции exec) под контролем отладчика. Приглашение отладчика появляется перед выполнением первой инструкции: Вы можете установить точки останова или использовать пошаговый режим выполнения. Аргументы *globals* и *locals* определяют окружение, в котором выполняются инструкции; по умолчанию используются пространства имен модуля __main__ (см. также раздел 10.3.11).

runeval(*expression* [, *globals* [, *locals*]])
Вычисляет выражение в строке *expression* (аналогично встроенной функции eval()) под контролем отладчика и (если вычисление прошло успешно) возвращает его значение. В остальном поведение этой функции аналогично поведению функции run().

runcall(*function* [, *arg1* ...])
Вызывает функцию *function* (или другой объект, поддерживающий вызов) с указанными аргументами под контролем отладчика и (если вызов функции прошел успешно) возвращает значение, возвращаемое этой функцией. Приглашение отладчика появляется перед выполнением первой инструкции тела функции.

set_trace()
: Запускает отладчик в кадре стека блока, из которого эта функция вызывается. Может быть полезна для жестко запрограммированных точек останова в точках, где, например, не выполняются отладочные утверждения.

post_mortem(*tb*)
: Производит "вскрытие" объекта *tb* (типа `traceback`).

pm()
: Производит "вскрытие" объекта `sys.last_traceback`.

25.1.2 Команды отладчика

Ниже описаны команды, распознаваемые отладчиком. Большинство команд можно ввести в сокращенном виде; например, запись '`h(elp)`' означает, что для выполнения данной команды следует ввести '`h`' или '`help`' (но не '`he`', '`hel`', '`H`', '`Help`' или '`HELP`'). Аргументы команды должны быть отделены символами пропуска. Необязательные аргументы команды указаны в квадратных скобках ('[*аргумент*]'), альтернативные варианты разделены вертикальной чертой ('|').

При вводе пустой строки отладчик повторяет последнюю выполненную команду. Исключение: если последней была команда '`list`', при вводе пустой строки выводятся следующие 11 строк исходного кода.

Все не распознанные отладчиком команды воспринимаются как инструкции языка Python, которые выполняются в контексте отлаживаемой программы. Команды, начинающиеся с восклицательного знака ('`!`'), всегда воспринимаются как инструкции языка Python. Выполнение произвольных инструкций является мощным средством отладки: Вы можете изменить значение переменной или вызвать функцию. Если при выполнении такой инструкции генерируется исключение, выводится информация об исключении, но состояние отладчика не изменяется.

Вы можете ввести несколько команд отладчика в одной строке, используя в качестве разделителя '`;;`' ('`;`' используется в качестве разделителя инструкций языка Python). Заметим, что отладчик не располагает "интеллектом", достаточным для того, чтобы распознать '`;;`' в середине строк, заключенных в кавычки.

Если в текущем и/или домашнем каталоге пользователя присутствует конфигурационный файл '.pdbrc', его строки выполняются при запуске отладчика, как если бы Вы вводили их в ответ на приглашение интерпретатора (сначала читается файл в домашнем каталоге пользователя, затем — в текущем). Такая возможность особенно полезна для определения псевдонимов.

h(elp) [*command*]
?[*command*]
: При использовании этой команды без аргументов выводит список всех возможных команд. С аргументом — выводит подсказку об использовании команды *command*. Команда '`help pdb`' выводит полностью файл документации отладчика, используя

программу постраничного просмотра, заданную в переменной окружения PAGER. Так как аргумент *command* должен быть идентификатором, для получения подсказки об использовании команды '!' необходимо ввести 'help exec'.

w(here)
Выводит информацию о месте, на котором выполнение было приостановлено.

d(own)
Перемещается на один кадр вниз по стеку (к более новому кадру стека).

u(p)
Перемещается на один кадр вверх по стеку (к более старому кадру стека).

b(reak) [([*filename*:]*lineno* | *function*) [, *condition*]]
Устанавливает точку останова в строке с номером *lineno* в файле *filename* (по умолчанию в текущем файле) или на первой инструкции функции *function*. Если указан аргумент *condition*, точка останова будет принята во внимание только в тех случаях, когда выражение *condition* является истиной. Каждой точке останова присваивается порядковый номер, который может быть использован для ее снятия.

Без аргументов выводит информацию обо всех установленных точках останова.

tbreak [([*filename*:]*lineno* | *function*) [, *condition*]]
Эта команда работает аналогично команде 'break', но устанавливает временную точку останова, которая будет снята после первого использования.

cl(ear) [*bpnumber* [*bpnumber* ...]]
Если задан список номеров точек останова (*bpnumber*), снимает их. При использовании без аргументов снимает все точки останова, предварительно запросив подтверждение.

disable [*bpnumber* [*bpnumber* ...]]
Временно отключает точки останова с номерами, указанными в списке аргументов. Вы можете снова включить их с помощью команды 'enable'.

enable [*bpnumber* [*bpnumber* ...]]
Включает точки останова, номера которых указаны в списке аргументов.

ignore *bpnumber* [*count*]
Указывает, что точка останова с номером *bpnumber* должна быть проигнорирована *count* (по умолчанию 0) раз.

condition *bpnumber* [*condition*]
Устанавливает условие, при котором точка останова будет приниматься во внимание: точка останова будет срабатывать только, если выражение *condition* является истиной. Если аргумент *condition* опущен, точка останова становится безусловной.

s(tep)
Выполняет инструкции в текущей строке и останавливается при первом удобном случае: либо в вызываемой функции, либо на следующей строке текущей функции.

n(ext)

Продолжает выполнение до следующей строки в текущей функции или до возврата из нее. Разница между 'next' и 'step' состоит в том, что команда 'step' выполняет в пошаговом режиме вызываемые функции, в то время как команда 'next' выполняет их без остановок.

r(eturn)

Продолжает выполнение до возврата из текущей функции.

c(ont(inue))

Продолжает выполнение до первой (активной) точки останова.

l(ist) [*first* [, *last*]]

Выводит строки исходного кода. При использовании без аргументов выводит 11 строк вокруг текущей, с одним аргументом — вокруг указанной. Если указаны оба аргумента, выводит строки в диапазоне от *first* до *last* или, если число *last* меньше *first*, *last* строк, начиная с *first*.

a(rgs)

Выводит список аргументов текущей функции.

p *expression*

Вычисляет значение выражения *expression* и выводит его значение. Заметим, что для этих целей Вы можете использовать инструкцию print языка Python, однако 'print' не является командой отладчика.

alias [*name* [*command*]]

Создает псевдоним *name* для команды *command*. Команда *не* должна быть заключена в кавычки. Фрагменты '%1', '%2' и т. д. заменяются первым, вторым и т. д. аргументами, переданными команде *name*, а '%*' — на все аргументы. Если аргумент *command* опущен, выводит команду для псевдонима *name*. При вызове без аргументов выводит список всех псевдонимов.

Команда *command* может использовать все, что можно вводить в ответ на приглашение интерпретатора, в том числе другие псевдонимы. Замена псевдонима производится только для первого слова команды, все остальное остается без изменений.

Приведем примеры двух полезных псевдонимов, которые можно поместить в конфигурационный файл '.pdbrc': команда 'alias pi for k, v in %1.__dict__.items(): print "%1.", k, -", v' создает псевдоним 'pi', выводящий атрибуты указанного экземпляра, а команда 'alias ps pi self' создает псевдоним 'ps', выводящий атрибуты экземпляра, в теле метода которого псевдоним используется.

unalias *name*

Удаляет псевдоним *name*.

[**!**]*statements*

Выполняет одну строку *statements* с инструкциями языка Python в текущем кадре стека. Восклицательный знак может быть опущен, за исключением случаев,

когда первое слово инструкции является командой (или псевдонимом) отладчика. Следует помнить, что для изменения глобальных переменных необходимо использовать инструкцию global, например:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

q(uit)
Выходит из отладчика. Выполнение отлаживаемой программы прерывается.

25.2 Замер производительности

Модули profile и pstats, описанные в этом разделе, позволяют осуществлять замер производительности программ на языке Python, анализировать результат и выводить отчет.

Copyright © 1994, by InfoSeek Corporation, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

25.2.1 Введение

Настоящее введение предоставляет краткий обзор основных средств. Он позволит Вам быстро осуществить замер производительности, не вникая в подробности.

Для того, чтобы вывести профиль выполнения функции 'foo()' (время выполнения 'foo()' и вызываемых из нее функций), следует выполнить следующие инструкции:

```
import profile
profile.run('foo()')
```

Чтобы сохранить информацию в файле (например, для сравнения с профилями для других вариантов реализации), укажите имя файла в качестве второго аргумента функции `profile.run()`:

```
import profile
profile.run('foo()', 'fooprof')
```

Вы можете также использовать файл '**profile.py**' в качестве программы для замера производительности другой программы, например:

```
python /usr/local/lib/python1.5/profile.py myscript.py
```

Для обработки статистических данных профиля большой программы могут быть полезны средства модуля `pstats`. Чтобы загрузить профиль из файла, воспользуйтесь следующим кодом:

```
import pstats
p = pstats.Stats('fooprof')
```

Созданный экземпляр класса `pstats.Stats` имеет множество методов для обработки и вывода статистических данных. Перед выводом отчета функция `profile.run()` сортирует данные в соответствии со строкой, содержащей имя модуля, номер строки и имя функции. Вы можете воспользоваться другим критерием. Например, следующий код выведет данные, отсортированные по имени функции:

```
p.sort_stats('name')
p.print_stats()
```

А следующий пример выведет 10 наиболее важных строк, отсортированных по совокупному затраченному времени (методы экземпляров класса `pstats.Stats` возвращают ссылку на экземпляр, что позволяет вызывать последовательно несколько методов в одной инструкции):

```
p.sort_stats('cumulative').print_stats(10)
```

Именно эти данные позволят Вам понять, на выполнение каких действий уходит больше всего времени.

Вы можете ограничить вывод с помощью регулярного выражения, например, вывести статистику для конструкторов классов:

```
p.print_stats('__init__')
```

Если же Вас интересует, из каких функций вызываются конструкторы, или наоборот, какие функции используются в конструкторах, воспользуйтесь следующими инструкциями соответственно:

```
p.print_callers('__init__')
p.print_callees('__init__')
```

25.2.2 `profile` — замер производительности

Основная функция модуля:

run(*string* [, *filename* ...])

Выполняет инструкции в строке *string* (с помощью инструкции `exec`) и выводит статистическую информацию в файл с именем *filename* (по умолчанию используется `sys.stdout`). В заголовке выводимого профиля указывается строка с инструкциями *string*, общее число вызовов функций, число примитивных вызовов (без рекурсии) и затраченное время. Далее следует шесть колонок с подробной информацией для каждой функции:

ncalls
: число вызовов (может быть два числа: общее число вызовов функции, число примитивных вызовов);

tottime
: суммарное время, затраченное на выполнение функции, исключая время, затраченное на вызовы из нее других функций;

percall
: среднее время, затрачиваемое на один вызов функции, исключая время, затраченное на вызовы из нее других функций (tottime/ncalls);

cumtime
: суммарное время, затраченное на выполнение функции и вызовы из нее других функций;

percall
: среднее время, затрачиваемое на один вызов функции и на вызовы из нее других функций (cumtime/ncalls);

filename:lineno(function)
: имя файла, номер строки и имя функции.

Для реализации функции `run()` используется определенный в этом модуле класс `Profile`. Кроме того, модуль `profile` определяет класс `HotProfile` производный от `Profile`, который обеспечивает более быстрый замер производительности, но не подсчитывает зависимости между функциями и суммарное время, затраченное на выполнение функции и вызовы из нее других функций (колонка cumtime).

Для сортировки собранных статистических данных и вывода отчета функция `run()` использует класс `pstats.Stats`.

25.2.3 `pstats` — обработка статистических данных и вывод отчетов

`Stats``([`*`filename`*` ...])`
: Создает и возвращает объект, реализующий обработку статистических данных о производительности из файла (или нескольких файлов, в этом случае данные объединяются) *`filename`*. Файлы должны быть созданы модулем `profile`, имеющим соответствующую версию (совместимость формата файлов для различных версий не гарантируется). После создания объекта дополнительные файлы могут быть прочитаны с помощью метода `add()`.

Экземпляры класса `Stats` имеют следующие методы:

`strip_dirs``()`
: Удаляет начало пути для всех имен файлов (оставляет только основные имена). Эта операция может быть полезна для того, чтобы уменьшить размер вывода, но может привести (редко) к неразличимости имен функций.

`add``([`*`filename`*` ...])`
: Считывает статистическую информацию из указанных файлов и объединяет ее с ранее считанной.

`sort_stats``([`*`key`*` ...])`
: Сортирует статистические данные объекта в соответствии с указанными критериями. Если указано более одного ключа, дополнительные ключи используются в качестве вторичного и т. д. критериев. Например, ‘`sort_stats('name', 'file')`’ сортирует данные по имени функции и данные для функций с одинаковыми именами по имени файла, в которой функция определена.

  В качестве ключей можно использовать неполные строки, если сокращение может быть разрешено однозначно. Метод воспринимает следующие ключи (если не указано иного, сортировка производится по возрастанию или в алфавитном порядке):

  `'calls'`
  : число вызовов (по убыванию);

  `'cumulative'`
  : суммарное время, затраченное на выполнение функции, исключая время, затраченное на вызовы из нее других функций (по убыванию);

  `'file'`
  : имя файла;

  `'module'`
  : имя модуля;

`'pcalls'`
: число примитивных вызовов (по убыванию);

`'line'`
: номер строки;

`'name'`
: имя функции;

`'nfl'`
: имя функции, имя файла, номер строки (вызовы '`sort_stats('nfl')`' и '`sort_stats('name', 'file', 'line')`' эквивалентны);

`'stdname'`
: стандартное имя, содержащее имя файла, номер строки и имя функции;

`'time'`
: суммарное время, затраченное на выполнение функции, исключая время, затраченное на вызовы из нее других функций (по убыванию).

`reverse_order``()`
: Меняет порядок следования данный на обратный.

`print_stats``([`*`restriction`*` ...])`
: Выводит отчет (аналогично функции `profile.run()`). Порядок следования данных определяется последним вызовом метода `sort_stats()`. Аргументы могут быть использованы для ограничения вывода информации. Каждый аргумент может быть целым числом (максимальное количество строк с данными), вещественное число от 0.0 до 1.0 (доля выводимой информации) или строка с регулярным выражением (выводятся только строки, ему удовлетворяющие). Если указано несколько ограничений, они применяются последовательно.

`print_callers``([`*`restriction`*` ...])`
: Для каждой записи выводит имя функции, которая вызвала данную. Для удобства в скобках указывается число таких вызовов и, затем, совокупное время. В остальном поведение этого метода идентично поведению метода `print_stats()`.

`print_callees``([`*`restriction`*` ...])`
: Для каждой записи выводит имя функций, которые данная функция вызывает. В остальном поведение этого метода идентично поведению метода `print_stats()`.

Все описанные здесь методы возвращают ссылку на экземпляр, поэтому несколько операций могут быть записаны одной инструкцией.

Глава 26

Выполнение в защищенном режиме

Обычно программы на языке Python имеют полный доступ к операционной системе через различные функции и объекты. Например, программа может открыть любой файл для чтения и записи с помощью встроенной функции `open()` (при условии, что у Вас достаточно полномочий). В большинстве случаев — это то, что Вам нужно.

С другой стороны, существует класс приложений, для которых такая "открытость" нежелательна. Представьте: Web-броузер (например, Grail) загружает код, написанный на языке Python, с различных адресов в Internet и выполняет его на локальной машине. Так как автор кода неизвестен, очевидно, что Вы не можете доверить ему все ресурсы своей машины.

При выполнении кода в защищенном режиме (restricted execution) происходит разделение потенциально опасного кода и безопасного кода. Идея состоит в создании "надсмотрщиком" окружения с ограниченными полномочиями и выполнении в нем кода, которому Вы не доверяете. Заметим, что код, выполняемый в таком окружении, может создать новое окружения с меньшими (но не большими) полномочиями.

Особенность модели защищенного режима в языке Python состоит в том, что интерфейс, представленный потенциально опасному коду, обычно такой же, как и в обычном режиме. Таким образом, нет необходимости в изучении каких-либо особенностей для написания кода, который будет выполняться в защищенном режиме. Так как окружение для защищенного режима создается "надсмотрщиком", Вы можете накладывать различные ограничения в зависимости от назначения приложения. Например, в определенной ситуации Вы можете посчитать безопасным чтение файлов в определенном каталоге, но не запись в них. В этом случае "надсмотрщик" может переопределить встроенную функцию `open()` таким образом, чтобы она генерировала исключение `IOError`, если открываемый файл находится за пределами разрешенного каталога или файл открывается для записи.

Интерпретатор языка Python для каждого блока проверяет переменную `__builtins__` и, если она не является ссылкой на встроенный модуль `__builtin__` или словарь, представляющий пространство его имен, считает, что блок кода выполняется в защищенном режиме. В защищенном режиме действуют некоторые ограничения, которые не позволяют программе выйти из окружения с ограниченными полномочиями. Например, становятся недоступными атрибут функций `func_globals` и атрибут классов и экземпляров `__dict__`.

Следующие два модуля предоставляют средства для настройки окружения защищенного режима:

rexec Основные средства настройки защищенного режима.

Bastion Ограничивает доступ к экземпляру класса.

26.1 rexec — основные средства настройки защищенного режима

Этот модуль определяет класс RExec, предоставляющий методы r_eval(), r_execfile(), r_exec() и r_import(), являющиеся ограниченными версиями встроенных функций eval() и execfile() и инструкций exec и import. Код, выполняемый в созданном этим классом окружении, имеет доступ только к модулям и функциям, которые считаются безопасными. Вы можете определить производный от RExec класс, если необходимо добавить или убрать какие-либо возможности.

Заметим, что класс RExec предотвращает использование кодом опасных операций (например, чтение и запись файлов или использование сетевых соединений), но не может защитить от использования слишком большого количества памяти или процессорного времени.

RExec([*hooks* [, *verbose*]])

Создает и возвращает объект, предоставляющий окружение с ограниченными возможностями. Аргумент *hooks* должен быть экземпляром RHooks (в настоящий момент не документирован) или производного от него класса, который реализует загрузку модулей. Если аргумент *hooks* опущен или равен None, автоматически создается новый экземпляр (используются аргументы по умолчанию). Если задан и является истиной аргумент *verbose*, на стандартный поток вывода будет выводиться отладочная информация.

Используя альтернативный объект в качестве аргумента *hooks*, Вы можете обеспечить переадресацию запросов к файловой системе через механизм RPC (Remote Procedure Call) или реализовать загрузку модулей с определенного URL (как это сделано в браузере Grail).

Класс RExec имеет следующие атрибуты, которые используются при создании экземпляра. Присваивание новых значений одноименным атрибутам экземпляра не окажет никакого влияния. Присваивание следует проводить атрибутам производного класса в его определении (но не в конструкторе). Все эти атрибуты являются кортежами из строк.

nok_builtin_names

Имена встроенных функций, которые не будут доступны в защищенном режиме. Для класса RExec это '('open', 'reload', '__import__')'. Добавлять имена в производных классах следует следующим способом:

```
class MyRExec(RExec):
    ...
    nok_builtin_names = RExec.nok_builtin_names +
```

Это позволит избежать неприятностей, если в будущих версиях языка будут добавлены новые опасные функции.

ok_builtin_modules

Имена встроенных и динамически подгружаемых модулей, использование которых безопасно. Для класса `RExec` это ‘`('audioop', 'array', 'binascii', 'cmath', 'errno', 'imageop', 'marshal', 'math', 'md5', 'operator', 'parser', 'regex', 'pcre', 'rotor', 'select', 'strop', 'struct', 'time')`’.

ok_path

Каталоги, в которых будет производиться поиск модулей при импортировании в защищенном режиме. Для класса `RExec` берется значение `sys.path` на момент инициализации модуля `rexec`.

ok_posix_names

Имена, определенные в модуле `os`, которые будут доступны в защищенном режиме. Для класса `RExec` это ‘`('error', 'fstat', 'listdir', 'lstat', 'readlink', 'stat', 'times', 'uname', 'getpid', 'getppid', 'getcwd', 'getuid', 'getgid', 'geteuid', 'getegid')`’.

ok_sys_names

Имена, определенные в модуле `sys`, которые будут доступны в защищенном режиме. Для класса `RExec` это ‘`('ps1', 'ps2', 'copyright', 'version', 'platform', 'exit', 'maxint')`’.

Экземпляры класса `RExec` поддерживают следующие методы. Отличие методов с приставками ‘`r_`’ и ‘`s_`’ состоит в том, что последние после выполнения кода восстанавливают исходные значения переменных `sys.stdin`, `sys.stderr` и `sys.stdout` в созданном окружении.

r_eval(*code*)

s_eval(*code*)

Вычисляет и возвращает значение выражения языка Python, представленного строкой или объектом кода *code*. Вычисление производится в модуле `__main__` созданного окружения.

r_exec(*code*)

s_exec(*code*)

Выполняет в модуле `__main__` созданного окружения инструкции языка Python, представленные строкой или объектом кода *code*.

r_execfile(*filename*)

s_execfile(*filename*)

Выполняет в модуле __main__ созданного окружения код языка Python, содержащийся в файле *filename*.

Класс RExec также определяет следующие методы, которые неявно вызываются при выполнении кода в защищенном режиме. Вы можете переопределить их в производном классе, чтобы изменить накладываемые ограничения (достаточно переопределить метод с приставкой 'r_').

r_import(*modulename* [, *globals* [, *locals* [, *fromlist*]]])

s_import(*modulename* [, *globals* [, *locals* [, *fromlist*]]])

Этот метод используется вместо встроенной функции __import__() (см. главу 12) для импортирования модуля. Должен генерировать исключение ImportError, если считается небезопасным.

r_open(*filename* [, *mode* [, *bufsize*]])

s_open(*filename* [, *mode* [, *bufsize*]])

Этот метод используется вместо встроенной функции open() (см. главу 12). По умолчанию метод r_open() позволяет открыть любой файл только для чтения (режимы 'r' и 'rb'). Смотрите пример в конце раздела, реализующий метод r_open() с меньшими ограничениями.

r_reload(*module*)

s_reload(*module*)

Этот метод используется вместо встроенной функции reload() (см. главу 12) для перезагрузки модуля.

r_unload()

s_unload()

Вы можете использовать эти методы для выгрузки модуля (удаления его из словаря sys.modules созданного окружения).

Приведем пример класса, позволяющего открывать файлы на запись в каталоге '/tmp':

```
from rexec import RExec
from os.path import normpath, islink

class TmpWriterRExec(RExec):
    def r_open(self, file, mode='r', bufsize=-1):
        if mode not in ('r', 'rb'):
            file = normpath(file)
```

```
        if file[:5]!='/tmp/' or islink(file):
            raise IOError(
                  "Can't write outside /tmp")
    return open(file, mode, bufsize)
```

26.2 Bastion — ограничение доступа к экземплярам классов

Этот модуль предназначен для использования совместно с модулем rexec и предоставляет возможность запретить доступ ко всем атрибутам данных и небезопасным методам экземпляра класса.

Bastion(*object* [, *filter* [, *name* [, *bastionclass*]]])
Возвращает экземпляр класса *bastionclass* (по умолчанию используется BastionClass), который ведет себя аналогично объекту *object*, но предоставляет доступ только к методам, для имен *attr_name* которых *filter*(*attr_name*) является истиной. Функция *filter*, используемая по умолчанию, запрещает доступ к методам, имена которых начинаются с символа подчеркивания (‘_’). При попытке доступа к атрибутам данных или запрещенным методам генерируется исключение AttributeError. Строковым представлением (результатом применения встроенной функции repr() или при заключении в обратные кавычки) защищенного объекта будет '<Bastion for *name*>'. Если аргумент *name* опущен или равен None, вместо *name* используется repr(*object*).

BastionClass(*getfunc*, *name*)
Этот (или производный от него) класс используется в качестве аргумента *bastionclass* функции Bastion() и реализует защиту объекта. Аргумент *getfunc* должен быть функцией (тип function), возвращающей значение атрибута с именем, переданным в качестве единственного аргумента, если атрибут считается безопасным, и генерирующей исключение AttributeError, если доступ к атрибуту запрещен. Аргумент *name* используется при конструировании строкового представления объекта.

Глава 27

Поддержка протоколов Internet

Стандартная библиотека предоставляет набор модулей, осуществляющих поддержку большинства распространенных протоколов Internet. Мы остановимся лишь на наиболее часто используемых — тех, которые предоставляют интерфейс высокого уровня.

cgi Протокол CGI (общий шлюзовой интерфейс), используемый для интерпретации форм HTML на стороне сервера.

urllib Чтение произвольных ресурсов по URL.

urlparse Операции над URL.

27.1 `cgi` — протокол CGI

Этот модуль предоставляет средства, которые будут полезны при написании программ, использующих интерфейс CGI (Common Gateway Interface, общий шлюзовой интерфейс), на языке Python.

27.1.1 Введение

CGI-программа вызывается HTTP-сервером, обычно для обработки данных, переданных пользователем через элементы '<FORM>' и '<ISINDEX>' языка HTML. HTTP-сервер помещает информацию о запросе (имя узла клиента, запрашиваемый URL, строка параметров запроса и др.) в переменные окружения программы, выполняет программу и пересылает клиенту его вывод. Часть данных от клиента может также поступать на стандартный поток ввода программы. Модуль `cgi` берет на себя заботу обо всех возможных способах передачи данных и предоставляет их программе через простой интерфейс. Модуль также предоставляет набор средств, которые будут полезны при отладке.

Вывод CGI-программы должен состоять из двух частей, разделенных пустой строкой. Первая часть содержит набор заголовков, которые описывают тип данных, следующих во втором разделе. Простейшая программа может выглядеть следующим образом:

```
# Заголовок: далее следует HTML-документ
print "Content-Type: text/html"
```

```
# Пустая строка: конец заголовков
print

# Вторая часть: HTML-документ
print "<html>"
print "<title>Вывод CGI-программы</title>"
print "<body>"
print "<h1>Это моя первая CGI-программа</h1>"
print "Привет всему миру!"
print "</body>"
print "</html>"
```

27.1.2 Использование модуля `cgi`

Начните с инструкции '`import cgi`'. Никогда не используйте '`from cgi import *`' — модуль определяет множество имен для внутреннего использования и для совместимости с предыдущими версиями, появление которых нежелательно в вашем пространстве имен.

`FieldStorage`([`**`*keyword_args*])

В этом классе сосредоточена наиболее часто используемая функциональность модуля `cgi`. При инициализации его без аргументов происходит обработка данных со стандартного потока ввода и/или из переменных окружения в соответствии со стандартом CGI[1]. Так как при этом поглощаются данные со стандартного ввода, следует создавать только один экземпляр класса `FieldStorage` за все время работы программы.

Конструктор класса воспринимает следующие именованные аргументы (*keyword_args*):

`fp`

Указывает альтернативный файловый (или подобный) объект, из которого будут считываться данные. По умолчанию используется `sys.stdin`. Этот аргумент игнорируется при использовании метода GET.

`headers`

Отображение с информацией о HTTP-заголовках. По умолчанию извлекается из переменных окружения.

`environ`

Отображение с информацией о переменных окружения. По умолчанию используется `os.environ`.

`keep_blank_values`

По умолчанию создаваемый экземпляр класса `FieldStorage` не содержит записи для полей, значения которых являются пустыми строками; для того,

[1] С одним исключением: при использовании метода POST не обрабатываются данные в строке запроса (часть URL после символа '`?`').

чтобы такие значения сохранялись, необходимо при инициализации указать этот аргумент равным истине.

strict_parsing
По умолчанию ошибки, возникающие при обработке данных, молча игнорируются. Если же задан и является истиной аргумент strict_parsing, при возникновении ошибок будет генерироваться исключение ValueError.

Помимо представления всей формы, экземпляры класса FieldStorage используются для представления полей формы, переданных в виде 'multipart/form-data'.

MiniFieldStorage
Экземпляры этого класса используются для представления полей формы, переданных в виде 'application/x-www-form-urlencoded'.

Экземпляры классов FieldStorage и MiniFieldStorage имеют следующие общие атрибуты данных:

name
Имя поля или None.

filename
Имя файла, указанное клиентом, или None.

value
Значение поля в виде строки, список значений (если используется несколько полей с одинаковым именем) или None. Если объект соответствует загружаемому файлу, при каждом обращении к этому атрибуту производится чтение всего содержимого файла.

file
Файловый (или подобный) объект, из которого Вы можете считывать данные, или None, если данные представлены строкой.

type
Тип содержимого поля (заголовок 'Content-Type') или None, если тип не указан.

type_options
Словарь параметров, указанных в заголовке 'Content-Type'.

disposition
Размещение содержимого поля (заголовок 'Content-Disposition') или None.

disposition_options
Словарь параметров, указанных в заголовке 'Content-Disposition'.

headers
Отображение, содержащее записи для всех заголовков.

Экземпляры класса FieldStorage являются отображениями имен полей к представляющим их объектам и поддерживают основные операции, характерные для отображений, а также методы has_key() и keys(). Помимо этого они имеют следующие методы:

getvalue(*key* [, *default*])
: Возвращает строковое значение поля с именем *key* или *default*, если нет поля с таким именем. Если форма содержит несколько полей с таким именем, возвращает список их строковых значений. По умолчанию в качестве аргумента *default* используется None.

make_file(*binary*)
: Этот метод используется реализацией класса для создания временного хранилища данных. По умолчанию используется tempfile.TemporaryFile(). Вы можете переопределить этот метод в производном классе, предоставив альтернативный способ временного хранения данных. В качестве аргумента *binary* используется строка 'b', если файл должен быть открыт в двоичном режиме, в противном случае он равен пустой строке. Метод должен возвращать файловый (или подобный) объект, доступный для чтения и записи.

Приведем простой пример CGI-программы, который проверяет, чтобы были заполнены поля 'name' и 'addr':

```
import cgi

print """\
Content-Type: text/html

<html>
  <body>"""

form = cgi.FieldStorage()
if form.has_key("name") and form.has_key("addr"):
    print """\
    <p>Имя: %s</p>
    <p>Адрес: %s</p>"""
else:
    print """\
    <h1>Ошибка</h1>
    <p>Введите, пожалуйста, имя и адрес.</p>"""

print """\
  </body>
</html>"""
```

Поля формы, доступные через form[*key*], могут быть представлены экземплярами класса FieldStorage, MiniFieldStorage (в зависимости от способа кодирования) или списком экземпляров, если форма содержит несколько полей с указанным

именем. В последнем случае метод getvalue() также возвращает список строковых значений полей. Если в Вашем случае возможно присутствие нескольких полей с одинаковым именем, используйте встроенную функцию type() для определения типа:

```
value = form.getvalue("username", "")
if type(value) is type([]):
    # Заполнено несколько полей с именем пользователя
    usernames = ",".join(value)
else:
    # Заполнено не более одного поля с именем
    # пользователя
    usernames = value
```

Если объект представляет загружаемый файл, при обращении к атрибуту value весь файл считывается в память в виде строки. Такое поведение не всегда желательно. Вы можете определить, представляет ли объект загружаемый файл, по значению атрибута filename или file:

```
fileitem = form["userfile"]
if fileitem.file:
    # Объект fileitem представляет загружаемый файл.
    # Подсчитываем строки, не сохраняя весь файл в
    # памяти.
    linecount = 0
    while 1:
        if fileitem.file.readline()
            linecount += 1
        else:
            break
```

Проект стандарта по организации загрузки файлов принимает во внимание возможность загрузки нескольких файлов из одного поля HTML-формы (рекурсивно используя 'multipart/*'). В этом случае объект будет являться отображением, с которым можно работать аналогично объекту, представляющему всю форму. Вы можете определить такой объект по значению атрибута type ('*obj*.type and *obj*.type.startswith('multipart/')').

27.1.3 Дополнительные возможности модуля

Модуль cgi определяет также дополнительные функции, которые будут полезны, если необходим более тонкий контроль.

parse([***keyword_args*])
Обрабатывает данные со стандартного потока ввода и/или из переменных окружения в соответствии со стандартом CGI[1] и возвращает словарь, отображающий

имена полей к спискам значений. Функция воспринимает именованные аргументы `fp`, `environ`, `keep_blank_values` и `strict_parsing`, которые имеют такое же значения, как и для конструктора класса `FieldStorage` (см. раздел 27.1.2).

`parse_qs``(`*`qs`* `[, `*`**keyword_args`*`])`

Обрабатывает строку запроса, переданную в качестве аргумента *`qs`* (данные типа '`application/x-www-form-urlencoded`'), и возвращает словарь, отображающий имена полей к спискам значений. Функция воспринимает именованные аргументы `keep_blank_values` и `strict_parsing`, которые имеют такое же значение, как и для конструктора класса `FieldStorage` (см. раздел 27.1.2).

`parse_qsl``(`*`qs`* `[, `*`**keyword_args`*`])`

Работает аналогично функции `parse_qs()`, но возвращает список кортежей с именем поля и его значением.

`parse_multipart``(`*`fp`*`, `*`pdict`*`)`

Обрабатывает данные в виде '`multipart/form-data`' из потока *`fp`* и возвращает словарь, отображающий имена полей к спискам значений. Аргумент *`pdict`* должен быть словарем, содержащим параметры из заголовка '`Content-Type`'.

Заметим, что эта функция не обрабатывает вложенные '`multipart/*`' части. Кроме того, она полностью считывает файл в память, что может быть нежелательным при загрузке больших файлов. Класс `FieldStorage` не обладает этими недостатками.

`parse_header``(`*`string`*`)`

Обрабатывает MIME-заголовок в строке *`string`* и возвращает кортеж, содержащий основное значение и словарь параметров. Предполагается, что строка *`string`* не содержит самого имени заголовка, то есть для заголовка '`Content-Type: text/plain; encoding=koi8-r`' аргумент должен быть равен `'text/plain; encoding=koi8-r'`.

`test``()`

Эта функция реализует готовую CGI-программу, предназначенный для тестирования: выводит минимальный HTTP-заголовок и всю переданную программе информацию в формате HTML. Функция `test()` вызывается при использовании файла модуля '`cgi.py`' в качестве программы.

`print_environ``()`

Выводит информацию о переменных окружения в формате HTML.

`print_form``(`*`form`*`)`

Выводит значения полей формы, представленной объектом *`form`* (должен быть экземпляром класс `FieldStorage`), в формате HTML.

`print_directory``()`

Выводит содержимое текущего каталога в формате HTML.

`print_environ_usage``()`

Выводит список переменных окружения, которые могут быть использованы CGI-программой.

escape(*s* [, *quote*])
Заменяет символы '&', '<' и '>' в строке *s* на соответствующие сущности HTML. Используйте эту функцию для вывода текста, который может содержать такие символы. Если задан и является истиной аргумент *quote*, также заменяет символ двойной кавычки ('"'), что может быть полезно для вывода значений атрибутов (например, ''a href=" + escape(*url*, 1) + ' '').

Кроме того, модуль предоставляет средства для ведения log-файла:

logfile
Если эта переменная имеет непустое значение, она определяет имя log-файла. Используется только, если переменная logfp является ложью.

logfp
Файловый (или подобный) объект, соответствующий log-файлу.

log(format [, **args*])
Записывает строку 'format % *args* + '\n'' в log-файл, определяемый значениями переменных logfile и logfp при первом вызове функции. Если в момент первого вызова этой функции обе переменный являются ложью, этот и все последующие вызовы функции ничего не делают.

27.1.4 Вопросы безопасности

Существует одно важное правило: если Вы запускаете внешнюю программу или считывает или записываете файл, убедитесь, чтобы в качестве параметров соответствующих функций (например, os.system(), os.popen*(), os.exec*(), os.spawn*(), open()) не использовались произвольные строки, полученные от клиента. Не следует доверять даже частям URL и именам полей, так как они могут поступить вовсе не из Вашей формы.

Если Вам все же необходимо использовать строки, полученные из формы, для составления команды shell или пути к файлу, убедитесь, что они содержат только безопасные символы (не содержит символов, имеющих специальное значение).

27.1.5 Установка CGI-программы

Поместите Вашу программу в каталог, предназначенный для CGI-программ (обычно это 'cgi-bin'). Убедитесь, что пользователь, от имени которого запущен HTTP-сервер, имеет доступ на чтение и выполнение программы. В операционных системах UNIX первая строка программы должна начинаться с '#!', после чего должен следовать полный путь к интерпретатору, например:

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

Для систем, отличных от UNIX, потребуется дополнительная настройка HTTP-сервера для того, чтобы он “знал”, каким образом необходимо выполнять программы на языке Python.

Если Ваша программа должна читать или записывать файлы, убедитесь, что пользователь, от имени которого запущен HTTP-сервер, имеет соответствующий доступ к каталогам и файлам. Следует также помнить, что значения переменных окружения пользователя, от имени которого будет выполняться программа (в том числе переменных PATH и PYTHONPATH) могут отличаться от тех, которые имеете Вы сами.

Для того, чтобы импортировать модули из каталогов, которые не включены в пути поиска модулей по умолчанию, измените переменную `sys.path` перед тем, как импортировать другие модули. Например:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
```

Следует помнить, что поиск производится в каталогах в том порядке, в котором они следуют в списке `sys.path`. Таким образом, в приведенном примере поиск будет производиться сначала в ‘/usr/home/joe/lib/python’, а затем в путях по умолчанию.

27.1.6 Отладка

К сожалению, часто CGI-программа работает в командной строке и, при этом, ведет себя странно при запуске сервером. Существует как минимум одна причина, по которой Вам следует сначала попробовать запустить программу из командной строки: это позволит Вам исключить синтаксические ошибки.

Если Ваша программа не содержит синтаксических ошибок и все же не работает, попробуйте использовать этот модуль (файл ‘cgi.py’) в качестве CGI-программы, чтобы исключить распространенные ошибки, связанные с установкой программы и настройкой сервера. Например, если файл ‘cgi.py’ установлен в стандартном каталоге ‘cgi-bin’, Вы можете послать примерно следующий запрос из браузера:

```
http://hostname/cgi-bin/cgi.py?name=Joe&addr=At+Home
```

Если при этом Вы получаете ошибку 404 — сервер не может найти программу; вероятно, необходимо установить его в другой каталог. Если же Вы получаете другую ошибку (например, 500) — у Вас проблемы с установкой программы или настройкой сервера, которые необходимо устранить перед тем, как идти дальше. И, наконец, если Вы получили форматированный вывод переменных окружения и содержимого формы (в нашем примере это поле ‘addr’ со значением ‘At home’ и поле ‘name’ со значением ‘Joe’) — файл ‘cgi.py’ установлен правильно.

Следующим шагом вызовите функцию `test()` модуля `cgi` из Вашей программы, то есть замените его код следующими двумя инструкциями:

```
import cgi
cgi.test()
```

Вы должны получить точно такой же результат, как и при обращении к 'cgi.py'.

Когда обычная программа на языке Python генерирует исключение, которое не обрабатывается, интерпретатор выводит информацию об исключении и прерывает выполнение. Однако вывод информации об исключении производится на стандартный поток ошибок (sys.stderr), который перенаправляется в log-файл сервера или вовсе игнорируется. Чтобы этого не происходило, Вы можете поместить в начало программы инструкцию

```
import sys
sys.stderr = sys.stdout
```

или поместить основной текст программы в блок try ... except и самостоятельно выводить информацию об исключении, например:

```
import cgi

print 'Content-Type: text/html\n'

try:
    # Основной текст CGI-программы
    ...
except:
    cgi.print_exception(limit=10)
```

27.2 urllib — чтение произвольных ресурсов по URL

Этот модуль предоставляет средства высокого уровня для чтения сетевых ресурсов, используя различные протоколы. В частности функция urlopen() ведет себя аналогично встроенной функции open(), но воспринимает URL (Uniform Resource Locator, единообразный указатель ресурса) вместо имени файла. При этом, естественно, налагаются некоторые ограничения: Вы можете открыть ресурс только для чтения и полученный "файловый" объект не имеет метода seek().

Определенные в этом модуле средства позволяют обращаться к ресурсам через proxy-сервер, не требующий аутентификации. Если proxy-сервера не указаны явно, они определяются через переменные окружения, системный реестр (Windows) или Internet Config (Macintosh). В операционных системах UNIX для использования proxy необходимо перед запуском интерпретатора установить переменные окружения (регистр букв имен переменных может быть произвольным) http_proxy, ftp_proxy и gopher_proxy равными URL, указывающим на proxy-сервера для соответствующих протоколов.

urlopen(*url* [, *data*])

Создает и возвращает объект, реализующий чтение ресурса *url*. Если URL не содержит идентификатора протокола (scheme identifier) или в качестве идентификатора используется '**file**', открывается локальный файл. В остальных случаях открывается сетевое соединение. Если соединение не может быть установлено или сервер сообщает об ошибке, генерируется исключение `IOError`.

По умолчанию для протокола HTTP используется метод GET. Для того, чтобы использовался метод POST, необходимо указать аргумент *data* с данными в формате '`application/x-www-form-urlencoded`' (см. описание функции `urlencode()` ниже).

urlretrieve(*url* [, *filename* [, *reporthook* [, *data*]]])

Копирует ресурс *url* в локальный файл. Если URL указывает на локальный файл или кэш уже содержит свежую копию ресурса, копирование не производится. Функция возвращает кортеж вида '`(`*filename*`,` *headers*`)`', где *filename* — имя локального файла с копией ресурса и *headers* — `None`, если *url* ссылается на локальный файл, иначе — объект, возвращаемый методом `info()` объекта, реализующего чтение ресурса.

Аргумент *filename* указывает имя локального файла, в который будет производиться копирование. По умолчанию имя файла генерируется с помощью функции `tempfile.mktemp()`. Задав аргумент *reporthook* Вы можете отслеживать процесс копирования. Функция (или другой объект, поддерживающий вызов) *reporthook* вызывается при установлении сетевого соединения и после загрузки каждого блока ресурса с тремя аргументами: число загруженных блоков, размер блока в байтах и размер файла (`-1`, если размер неизвестен). Аргумент *data* имеет такое же значение, как и для функции `urlopen()`.

urlcleanup()

Очищает кэш, созданный предыдущими вызовами функции `urlretrieve()`[2].

quote(*string* [, *safe*])

Заменяет специальные символы в строке *string* на последовательности вида '`%`*xx*' и возвращает результат. Преобразованию никогда не подвергаются буквы, цифры и символы '`_`', '`,`', '`.`' и '`-`'. Аргумент *safe* указывает дополнительные символы, которые должны быть оставлены без изменений; по умолчанию он равен `'/'`.

quote_plus(*string* [, *safe*])

Работает аналогично функции `quote()`, но также заменят пробелы на '+', как это необходимо для обработки значений полей форм. Если символ '+' не содержится в строке *safe*, то он заменяется на '`%2b`'.

unquote(*string*)

Заменяет специальные последовательности вида '`%`*xx*' на соответствующие символы и возвращает результат.

[2] В текущих реализациях модуля кэширование по умолчанию отключено, так как еще не реализован механизм проверки срока хранения копий ресурсов.

unquote_plus(*string*)

Работает аналогично функции unquote(), но также заменяет символы '+' пробелами, как это необходимо для восстановления значений полей форм.

urlencode(*dict*)

Возвращает строку с данными *dict* (отображение строк-имен переменных к строкам-значениям) в формате 'application/x-www-form-urlencoded'. Результат может быть использован в качестве аргумента *data* функций urlopen() и urlretrieve() и метода open() экземпляров URLopener и производных от него классов. Эта функция полезна для преобразования значений полей формы в строку запроса. Возвращаемая строка состоит из последовательности фрагментов вида '*key*=*value*' для каждой записи в словаре, разделенных символом '&', где *key* и *value* обрабатываются с помощью функции quote_plus().

Объекты, реализующие чтение ресурса, помимо read(), readline(), readlines(), fileno() и close(), характерных для файловых объектов (см. раздел 11.7), имеют следующие методы:

info()

Возвращает экземпляр класса mimetools.Message, содержащий метаинформацию о ресурсе. При использовании протокола HTTP этот объект содержит информацию обо всех заголовках. При использовании протокола FTP заголовок 'Content-Length' будет присутствовать, если сервер посылает информацию о длине файла. При обращении к локальному файлу возвращаемый этим методом объект будет содержать заголовок 'Date' с датой и временем последнего изменения, заголовок 'Content-Length' с размером файла и заголовок 'Content-Type' с предполагаемым типом файла.

geturl()

Возвращает истинный URL ресурса. В некоторых случаях HTTP-сервер перенаправляет клиента на другой URL. Такие перенаправления автоматически обрабатываются, и Вы получаете ресурс с новым URL, узнать который можно с помощью этого метода.

Функции urlopen() и urlretrieve() создают и используют экземпляр класса FancyURLopener. Созданный объект сохраняется для дальнейшего использования этими функциями. Если Вам необходим более тонкий контроль — используйте FancyURLopener, URLopener или производные от них классы.

URLopener([*proxies* [, ***x509*]])

Базовый класс, реализующий чтение ресурсов. Если Вам не нужна поддержка каких-либо дополнительных протоколов — используйте класс FancyURLopener. Если задан и не равен None аргумент *proxies*, он должен быть отображением идентификаторов протоколов (scheme identifier) к URL соответствующих proxy-серверов и будет использован для определения способа соединения. По умолчанию URL proxy-серверов определяются по схеме, описанной в начале раздела.

Для аутентификации при использовании протокола HTTPS должны быть указаны именованные аргументы (*x509*) `key_file` и `cert_file`.

FancyURLopener([*proxies* [, ***x509*]])

Этот класс является производным от класса `URLopener` и предоставляет обработку HTTP-ответов с кодами 301, 302 и 401. При получении ответов с кодами 301 и 302 используется заголовок '`Location`' для переадресации на другой URL. Для ответов с кодом 401 используется базовая аутентификация. Аргументы конструктора этого класса имеют такое же значение, как и для конструктора класса `URLopener`.

Экземпляры классов `URLopener` и `FancyURLopener` имеют следующие методы и атрибуты данных:

open(*url* [, *data*])

Открывает ресурс *url* и возвращает объект, аналогичный файловому, реализующий чтение ресурса (см. выше). Этот метод проверяет кэш, обрабатывает информацию о proxy-сервере и вызывает один из методов `open_`*scheme*`()`, где *scheme* заменяется идентификатором протокола (в котором символы '-' заменены на '_'), или метод `open_unknown()`, если используемый идентификатор протокола неизвестен. Аргумент *data* имеет такое же значение, как и для функции `urlopen()`.

open_unknown(*url* [, *data*])

Этот метод может быть переопределен в производном классе для обработки URL с неизвестным идентификатором протокола. Исходная реализация генерирует исключение `IOError`.

retrieve(*url* [, *filename* [, *reporthook* [, *data*]]])

Копирует ресурс *url* в локальный файл. Если URL указывает на локальный файл или кэш уже содержит свежую копию ресурса, копирование не производится. Метод возвращает кортеж вида '`(`*filename*`, `*headers*`)`', где *filename* — имя локального файла с копией ресурса и *headers* — `None`, если *url* ссылается на локальный файл, иначе — объект, возвращаемый методом `info()` объекта, реализующего чтение ресурса (см. выше).

Аргумент *filename* указывает имя локального файла, в который будет производиться копирование. По умолчанию имя файла генерируется с помощью функции `tempfile.mktemp()`. Задав аргумент *reporthook* Вы можете отслеживать процесс копирования. Функция (или другой объект, поддерживающий вызов) *reporthook* вызывается при установлении сетевого соединения и после загрузки каждого блока ресурса с тремя аргументами: число загруженных блоков, размер блока в байтах и размер файла (`-1`, если размер неизвестен). Аргумент *data* имеет такое же значение, как и для функции `urlopen()`.

version

Этот атрибут класса является строкой с именем программы клиента, которое будет использовано при конструировании HTTP-запросов. По умолчанию используется строка `'Python-urllib/`*urllib_ver*`'`, где *urllib_ver* — версия модуля

urllib. Вы можете переопределить атрибут version в определении производного класса, чтобы изменить имя программы, посылаемое серверу.

Для совместимости с предыдущими версиями, если URL без идентификатора протокола (scheme identifier) не соответствует существующему локальному файлу, производится попытка использовать его для FTP протокола. Это может привести к сбивающим с толку сообщениям об ошибках.

При установлении сетевого соединения может произойти задержка на произвольное время. Таким образом, использование средств модуля urllib в однопоточных программах сильно затруднено.

Следующий пример использует метод GET для загрузки ресурса по протоколу HTTP:

```
>>> import urllib
>>> params = urllib.urlencode({'spam' : 1,
...                            'eggs' : 2,
...                            'bacon': 0})
>>> f = urllib.urlopen(
...     "http://www.musi-cal.com/cgi-bin/query?" +
...     params)
>>> print f.read()
```

И тот же самый ресурс, используя метод POST:

```
>>> import urllib
>>> params = urllib.urlencode({'spam' : 1,
...                            'eggs' : 2,
...                            'bacon': 0})
>>> f = urllib.urlopen(
...     "http://www.musi-cal.com/cgi-bin/query?",
...     params)
>>> print f.read()
```

27.3 urlparse — операции над URL

Этот модуль определяет средства для разбиения URL (Uniform Resource Locator, единообразный указатель ресурса, RFC 1738) на компоненты, конструирования URL из компонент и преобразования относительных URL в абсолютные (RFC 1808).

urlparse(*urlstring* [, *default_scheme* [, *allow_fragments*]])
Разбивает URL на компоненты и возвращает кортеж из шести строк: идентификатор протокола (scheme identifier), положение в сети, путь, параметры, строка

запроса и идентификатор фрагмента. То есть применение функции к строке общего вида

'*scheme*://*netloc*/*path*;*parameters*?*query*#*fragment*'

дает '(*scheme*, *netloc*, *path*, *parameters*, *query*, *fragment*)'. Если URL не содержит какого-либо компонента, соответствующий элемент возвращаемого кортежа будет равен пустой строке. Разделители компонент не включаются в результат, за исключением косой черты в начале пути:

```
>>> import urlparse
>>> urlparse.urlparse(
...     'http://www.cwi.nl:80/%7Eguido/Python.html')
('http', 'www.cwi.nl:80', '/%7Eguido/Python.html', '',
'', '')
```

Если задан аргумент *default_scheme*, он будет использован в качестве идентификатора протокола (scheme identifier) по умолчанию — в том случае, если указанный URL не содержит идентификатора протокола.

Если задан и равен нулю аргумент *allow_fragments*, использование идентификатора фрагмента в URL (последний компонент) считается недопустимым.

urlunparse(*tuple*)

Восстанавливает и возвращает URL из компонент, переданных в кортеже *tuple*. При последовательном применении функций urlparse() и urlunparse() Вы можете получить другой (например, если исходный URL содержит излишние разделители), но эквивалентный URL.

urljoin(*base*, *rel_url* [, *allow_fragments*])

Конструирует и возвращает полный URL, комбинируя базовый URL *base* и относительный URL *rel_url*. Например:

```
>>> urljoin(
...     'http://www.cwi.nl/%7Eguido/Python.html',
...     'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

Если задан и равен нулю аргумент *allow_fragments*, использование идентификатора фрагмента в URL (последний компонент) считается недопустимым.

Глава 28

Поддержка форматов, используемых в Internet

В этой главе описаны модули, обеспечивающих поддержку форматов, обычно используемых для передачи данных через Internet. Модули, предназначенные для обработки структурной разметки, описаны отдельно — в главе 29.

rfc822	Обработка заголовков электронных писем (в стиле RFC 822).
mimetools	Обработка сообщений в формате MIME.
MimeWriter	Средства для записи в формате MIME.
multifile	Чтение сообщений, состоящих из нескольких частей (например, в формате MIME).
xdrlib	Преобразование данных в формат XDR и обратно.

28.1 `rfc822` — обработка заголовков электронных писем

Модуль `rfc822` определяет класс `Message`, представляющий набор заголовков электронного письма (см. описание стандарта RFC 822). Он может быть использован в различных ситуациях, обычно для чтения таких заголовков из файла. Этот модуль также определяет класс `AddressList`, предназначенный для обработки адресов в соответствии с RFC 822, и несколько вспомогательных функций.

Message(*file* [, *seekable*])

При создании экземпляров класса `Message` в качестве аргумента *file* должен быть указан файловый или подобный объект (достаточно, чтобы этот объект имел метод `readline()`). При инициализации считываются и сохраняются заголовки из *file* вплоть до разделительной пустой строки.

Наличие методов `seek()` и `tell()` у объекта *file* позволяет использовать метод `rewindbody()` экземпляров класса `Message`. Кроме того, при наличии этих методов или метода `unread()` некорректные строки будут возвращены в поток. Таким образом, этот класс может быть использован для чтения буферизованных потоков.

Если аргумент *seekable* опущен или равен `1`, наличие методов `seek()` и `tell()` определяется путем вызова `tell()`. Однако такой механизм не всегда хорошо работает. Поэтому, для обеспечения максимальной переносимости следует установить *seekable* равным нулю, если объект `file` не поддерживает установку указателя (например, если он представляет сетевое соединение).

Строки, считываемые из файла, могут заканчиваться комбинацией CR-LF или одним символом LF; перед сохранением комбинация CR-LF заменяется на LF.

AddressList(*field*)
При инициализации экземпляров этого класса используется единственный аргумент — строка, в которой, в соответствии с RFC 822, через запятую перечислены адреса. При использовании в качестве аргумента *field* значения `None`, созданный экземпляр будет представлять пустой список.

parsedate(*date*)
Пытается проанализировать строку с датой в соответствии с правилами, приведенными в RFC 822 (например, `'Mon, 20 Nov 1995 19:12:08 -0500'`). В некоторых случаях эта функция срабатывает и для строк, в которых дата записана не в соответствии со стандартом. В случае удачного анализа возвращает кортеж из девяти элементов, пригодный для использования в качестве аргумента функции `time.mktime()` (назначение элементов кортежа смотрите в описании модуля `time`), иначе возвращает `None`. Заметим, что 6, 7 и 8 элементы возвращаемого кортежа не имеют полезных значений.

parsedate_tz(*date*)
Работает аналогично функции `parsedate()`, но возвращает либо `None`, либо кортеж из десяти элементов. Первые девять элементов такие же, десятый — сдвиг часового пояса в секундах относительно универсального времени. Заметим, что знак этого сдвига противоположен знаку `time.timezone` для того же часового пояса (`time.timezone` следует стандарту POSIX, в то время как данная функция — RFC 822). Если строка *date* не содержит информации о часовом поясе, последний элемент кортежа будет равен `None`. Заметим, что 6, 7 и 8 элементы возвращаемого кортежа не имеют полезных значений.

mktime_tz(*tuple*)
Преобразует кортеж из десяти элементов (например, возвращаемый функцией `parsedate_tz()`) в число секунд, пройденных с начала эпохи (UTC timestamp). Если последний элемент кортежа *tuple* равен `None`, считается, что время в *tuple* представлено для локального часового пояса. Эта функция может давать небольшую ошибку в точках перехода на летнее и зимнее время.

formatdate([*tuple*])
Возвращает строку с временем *tuple* (кортеж из девяти элементов — значение элементов смотри в описании модуля `time`), представленным в соответствии с RFC 822 и RFC 1123.

Экземпляры класса `Message` являются отображениями имен заголовков к их значениям (с доступом только на чтение) и поддерживают большинство характерных для

отображений методов (has_key(), keys(), values(), items(), get()). Сопоставление имен заголовков производится независимо от используемого регистра букв, то есть выражения *m*['From'], *m*['from'] и *m*['FROM'] полностью эквивалентны. Значением заголовка считается текст после двоеточия в первой строке заголовка плюс текст из строк продолжения с убранными символами пропуска в начале и конце. Кроме того, эти объекты имеют следующие методы и атрибуты данных:

rewindbody()
: Перемещает указатель на начало сообщения. Этот метода работает только в том случае, если используемый файловый объект имеет методы tell() и seek().

isheader(*line*)
: Возвращает канонизированное имя заголовка, если *line* является соответствующим RFC 822 заголовком, иначе возвращает None (предполагая, что анализ должен быть остановлен в этом месте и строка должна быть помещена обратно в поток). Вы можете переопределить этот метод в производном классе для обеспечения поддержки расширенных форматов заголовков.

islast(*line*)
: Возвращает истину, если строка *line* является разделителем, на котором обработка заголовков должна быть остановлена, иначе возвращает ложь. При чтении разделительная строка поглощается, и указатель помещается непосредственно после нее. Вы можете переопределить этот метод в производном классе. Исходная реализация проверяет, является ли данная строка пустой.

iscomment(*line*)
: Возвращает истину, если строка *line* должна быть полностью проигнорирована, иначе возвращает ложь. Вы можете переопределить этот метод в производном классе, исходная реализация всегда возвращает ложь.

getallmatchingheaders(*name*)
: Возвращает список строк, составляющих все заголовки с именем *name*. Каждая строка, в том числе являющаяся продолжением заголовка, является отдельным элементом этого списка. Если нет заголовков с указанным именем, возвращает пустой список.

getfirstmatchingheader(*name*)
: Возвращает список строк, составляющих первый заголовок с именем *name*. Если нет заголовков с указанным именем, возвращает None.

getrawheader(*name*)
: Возвращает строку, составляющую значение первого заголовка с именем (текст после двоеточия, включая строки продолжения) *name*, в исходном виде. Необработанное значение включает символы пропуска в начале, завершающий символ перехода на новую строку и дополнительные символы перехода на новую строку и пропуска перед текстом, взятым из строк продолжения. Если нет заголовков с указанным именем, возвращает None.

getheader(*name* [, *default*])
: Псевдоним стандартного метода get() отображений.

getaddr(*name*)

Возвращает кортеж из двух строк (возможно пустых) с полным именем и адресом электронной почты, взятых из первого заголовка с именем *name*. Если нет заголовков с указанным именем, возвращает '(None, None)'. Например, если первый заголовок, сохраненный объектом *m*, содержит строку 'jack@cwi.nl (Jack Jansen)' (или 'Jack Jansen <jack@cwi.nl>'), *m*.getaddr('From') даст пару '('Jack Jansen', 'jack@cwi.nl')'.

getaddrlist(*name*)

Работает аналогично методу getaddr(), но обрабатывает заголовок (или заголовки) с именем *name* как содержащий список адресов (например, заголовок 'To' или серия заголовков 'Cc') и возвращает список кортежей с полным именем и адресом электронной почты. Если нет заголовков с указанным именем, возвращает пустой список.

getdate(*name*)

Анализирует первый заголовок с именем *name* на предмет содержания даты и времени и возвращает кортеж из девяти элементов (см. описание функции parsedate()) или None, если нет заголовков с указанным именем или первый такой заголовок не содержит даты (не может быть обработан функцией parsedate()).

getdate_tz(*name*)

Работает аналогично методу getdate(), но возвращает кортеж из десяти элементов; последний элемент равен сдвигу в секундах часового пояса относительно универсального времени (см. описание функции parsedate_tz()) или None.

headers

Список всех строк заголовков в порядке их следования во входном файле. Каждая строка содержит завершающий символ перехода на новую строку. Разделительная строка (сигнализирующая конец заголовков) в этот список не включается.

fp

Файловый (или подобный) объект, который был использован при инициализации. Может быть использован для чтения содержимого сообщения.

Экземпляры класса AddressList (обозначенные ниже как *a* и *b*) по поведению схожи с множествами (см. раздел 9.7.3) и поддерживают следующие операции:

len(*a*)

Количество адресов в списке.

str(*a*)

Канонизированное строковое представление списка (через запятую) адресов. Каждый адрес представляется в виде '"*name*" <*e_mail*>'.

a **+** *b*

Дает экземпляр класса AddressList с адресами из обоих списков, но без дубликатов (объединение множеств).

a **-** *b*
Дает экземпляр класса AddressList с адресами, которые содержаться в списке *a*, но не содержаться в списке *b* (разница множеств).

Кроме того, экземпляры класса AddressList имеют один публичный атрибут:

addresslist
Список кортежей (по одному на каждый адрес) из двух строк: канонизированного имени и электронного адреса.

28.2 mimetools — обработка сообщений в формате MIME

Этот модуль определяет класс Message, производный от rfc822.Message, а также набор функций, полезных при обработке сообщений в формате MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения электронной почты).

Message(*fp* [, *seekable*])
Этот класс является производным от класса rfc822.Message и определяет несколько дополнительный методов, описанных ниже. Аргументы имеют такое же значение, как и для класса rfc822.Message (см. описание модуля rfc822).

choose_boundary()
Возвращает уникальную строку, которая с высокой вероятностью может быть использована в качестве разделителя частей. Строка составляется из IP-адреса, идентификатора пользователя, идентификатора процесса, числа секунд, пройденных с начала эпохи, и некоторого случайного числа, разделенных точкой.

decode(*input*, *output*, *encoding*)
Считывает данные в MIME-кодировке *encoding* из потока *input* и записывает раскодированные данные в поток *output*. Аргумент *encoding* может иметь одно из следующих значений: 'base64', 'quoted-printable', 'uuencode' (для большей совместимости воспринимаются также 'x-uuencode', 'uue' и 'x-uue'), '7bit' или '8bit'.

encode(*input*, *output*, *encoding*)
Считывает данные из потока *input*, преобразует в MIME-кодировку *encoding* и записывает в поток *output*. Поддерживаемые значения аргумента *encoding* такие же, как и для функции decode().

copyliteral(*input*, *output*)
Считывает строки текста из потока *input* (до конца файла) и записывает их в поток *output*.

copybinary(*input*, *output*)

Считывает блоки двоичных данных из потока *input* (до конца файла) и записывает их в поток *output*. В текущих реализациях размер блоков фиксирован и равен 8192 байт.

Экземпляры класса `Message` помимо унаследованных от `rfc822.Message` имеют следующие методы:

getplist()

Возвращает список параметров (строк) из заголовка 'content-type'. Для параметров вида '*name*=*value*' имя *name* преобразуется к нижнему регистру. Например, если сообщение содержит заголовок 'Content-type: text/html; spam=1; Spam=2; Spam', метод `getplist()` вернет список `['spam=1', 'spam=2', 'Spam']`.

getparam(*name*)

Возвращает значение (*value*) первого параметра вида '*name*=*value*' с указанным именем (сопоставление производится без учета регистра букв) из заголовка 'content-type'. Если значение помещено в двойные кавычки '"*value*"' или угловые скобки '<*value*>', то они удаляются.

getencoding()

Возвращает строку с кодировкой сообщения, указанного в заголовке 'content-transfer-encoding'. Если сообщение не содержит такого заголовка, возвращает `'7bit'`. Буквы в возвращаемой строке приводятся к нижнему регистру.

gettype()

Возвращает тип содержимого сообщения в виде строки '*основной_тип*/*подтип*', указанный в заголовке `content-type`. Если сообщение не содержит такого заголовка, возвращает `'text/plain'`. Буквы в возвращаемой строке приведены к нижнему регистру.

getmaintype()

Возвращает строку с основным типом содержимого сообщения, указанного в заголовке `content-type`. Если сообщение не содержит такого заголовка, возвращает `'text'`. Буквы в возвращаемой строке приведены к нижнему регистру.

getsubtype()

Возвращает строку с подтипом содержимого сообщения, указанного в заголовке `content-type`. Если сообщение не содержит такого заголовка, возвращает `'plain'`. Буквы в возвращаемой строке приведены к нижнему регистру.

28.3 `MimeWriter` — средства для записи в формате MIME

Этот модуль определяет класс `MimeWriter`, реализующий запись сообщений, состоящих из нескольких частей, в MIME-формате:

MimeWriter(*fp*)
Экземпляры этого класса реализуют запись в MIME-формате в поток *fp* (файловый или подобный объект).

Экземпляры класса MimeWriter имеют следующие методы:

addheader(*name*, *value* [, *prefix*])
Добавляет к сообщению заголовок *name* со значением *value*. Необязательный аргумент *prefix* указывает место, в которое заголовок будет добавлен: 0 (используется по умолчанию) — добавить в коней, 1 — вставить в начало.

flushheaders()
При вызове этого метода все накопленные (с помощью метода addheader()) заголовки записываются в поток и забываются. Такая операция может быть полезна, если Вы не собираетесь записывать основную часть (тело) сообщения, например, для блоков типа 'message/rfc822', которые иногда используется для представления информации, аналогичной заголовкам.

startbody(*ctype* [, *plist* [, *prefix*]])
Возвращает объект, аналогичный файловому, который может быть использован для записи основной части (тела) сообщения. Аргумент *ctype* указывает значение, которое будет использоваться в заголовке 'content-type' (тип данных). Если задан аргумент *plist*, он должен быть списком пар вида '(name, *value*)', которые будут использованы в качестве дополнительных параметров в заголовке 'content-type'. Назначение аргумента *prefix* такое же, как и для метода addheader(), но по умолчанию он равен 0 (добавить в начало).

startmultipartbody(*subtype* [, *boundary* [, *plist* [, *prefix*]]])
Возвращает объект, аналогичный файловому, который может быть использован для записи тела сообщения, состоящего из нескольких частей. Аргумент указывает подтип сообщения типа 'multipart', *boundary* может быть использован для указания дружественного разделителя частей (по умолчанию используется mimetools.choose_boundary()), список *plist* задает дополнительные параметры заголовка 'content-type'. Аргумент *prefix* имеет такое же значение, как и для метода startbody(). Подчасти должны создаваться с помощью метода nextpart().

nextpart()
Возвращает новый экземпляр класса MimeWriter, представляющий часть в сообщении типа 'multipart/*'. Этот объект может быть использован, в том числе, и для вложенных сообщений, состоящих из нескольких частей. Перед использованием метода nextpart() должен быть вызван метод startmultipartbody().

lastpart()
Это метод *всегда* должен быть использован после записи сообщений, состоящих из нескольких частей (типа 'multipart/*').

28.4 `multifile` — чтение сообщений, состоящих из нескольких частей

Этот модуль определяет класс `MultiFile`, позволяющий читать текст, состоящий из нескольких частей: метод `readline()` его экземпляров возвращает пустую строку при достижении заданного разделителя. Исходная реализация класса рассчитана на чтения сообщений в формате MIME, однако, переопределив методы в производном классе, Вы можете адаптировать его на общий случай.

`MultiFile`(*fp* [, *seekable*])
Создает и возвращает объект, реализующий чтение частей текста. Аргумент *fp* должен быть файловым (или подобным) объектом, из которого будет производиться чтение исходного текста. На самом деле достаточно наличия у *fp* методов `readline()`, `seek()` и `tell()`. Последние два метода требуются для обеспечения произвольного доступа к частям текста. Если задан и является ложью аргумент *seekable*, методы `seek()` и `tell()` объекта *fp* использоваться не будут.

`Error`
Это исключение генерируется в случае возникновения характерных для модуля ошибок.

С точки зрения реализации класса `MultiFile` исходный текст состоит из трех видов строк: данных, разделителей частей и метки конца. Этот класс приспособлен к обработке сообщений, части которых состоят из несколько вложенных частей. В каждом таком случае должен быть свой шаблон для разделительных строк и метки конца.

Экземпляры класса `MultiFile` имеют следующие методы:

`push`(*str*)
Записывает в стек разделительную строку. Если эта строка, соответствующим образом выделенная, будет найдена во входном потоке, она будет интерпретирована как разделитель частей или метка конца. Все последующие попытки чтения из экземпляра класса `MultiFile` будут возвращать пустую строку до тех пор, пока разделительная строка не будет удалена из стека с помощью метода `pop()` или ее использование не будет повторно разрешено с помощью метода `next()`.

Стек может содержать более одной разделительной строки — использоваться в качестве шаблона будет та, которая находится на вершине стека, то есть была записана последней (если в тексте попадется другая строка из стека, будет сгенерировано исключение `Error`).

`readline`()
Считывает и возвращает очередную строку из текущей части. При достижении разделителя частей или метки конца возвращает пустую строку. Если считываемая строка удовлетворяет другому разделителю в стеке или поток с исходным текстом выдает конец файла (при непустом стеке), генерирует исключение `Error`.

readlines()
: Возвращает список всех оставшихся строк для данной части.

read()
: Считывает и возвращает все оставшиеся данные для текущей части.

next()
: Считывает строки до конца текущей части и, если достигнута метка конца, возвращает 0, в противном случае переходит к следующей части (разрешает повторное использование разделителя на вершине стека) и возвращает 1.

pop()
: Удаляет с вершины стека разделительную строку.

seek(*pos* [, *whence*])
: Перемещает указатель по текущей части. Аргументы имеют такое же значение, как и для метода seek() файловых объектов (см. раздел 11.7).

tell()
: Возвращает текущую позицию в текущей части.

is_data(*str*)
: Возвращает 1, если строка *str* точно не является разделителем или меткой конца (то есть является строкой с данными), иначе (если строка может быть разделителем или меткой конца) возвращает 0. Вы можете переопределить этот метод в производном классе, исходная реализация проверяет наличие '--' в начале строки (для формата MIME).

  Этот метод предназначен для быстрой проверки: если он будет всегда возвращать 0, это лишь замедлит обработку, но не приведет к ошибкам.

section_divider(*str*)
: Преобразует разделительную строку *str* (в том виде, в котором она записывается в стек) в строку, которая используется для разделения частей. Вы можете переопределить этот метод в производном классе, исходная реализация добавляет '--' в начало (для формата MIME). Нет необходимости добавлять в конец символ перехода на новую строку, так как при сравнении символы пропуска в конце строки игнорируются.

end_marker(*str*)
: Преобразует разделительную строку *str* (в том виде, в котором она записывается в стек) в строку, которая используется в качестве метки конца. Вы можете переопределить этот метод в производном классе, исходная реализация добавляет '--' в начало и конец (для формата MIME). Нет необходимости добавлять в конец символ перехода на новую строку, так как при сравнении символы пропуска в конце строки игнорируются.

Кроме того, экземпляры класса MultiFile имеют два публичных атрибута данных:

level
Уровень вложенности текущей части.

last
Является истиной (1), если последний разделитель являлся меткой конца.

В качестве примера определим функцию, которая находит в сообщении (в формате MIME) часть, имеющую определенный тип (если сообщение не содержит частей такого типа, возвращает пустую строку):

```
import mimetools
import multifile
import StringIO

def extract_mime_part_matching(stream, mimetype):

    msg = mimetools.Message(stream)
    msgtype = msg.gettype()
    params = msg.getplist()

    data = StringIO.StringIO()
    if msgtype.startswith('multipart/'):

        file = multifile.MultiFile(stream)
        file.push(msg.getparam('boundary'))
        while file.next():
            submsg = mimetools.Message(file)
            try:
                data = StringIO.StringIO()
                mimetools.decode(file, data,
                                 submsg.getencoding())
            except ValueError:
                continue
            if submsg.gettype() == mimetype:
                break
        file.pop()
    return data.getvalue()
```

28.5 xdrlib — представление данных в формате XDR

Модуль xdrlib предоставляет средства для работы с данными в формате XDR (eXternal Data Representation, аппаратно-независимое представление данных) в соответствии с RFC 1014[1] и поддерживает большинство типов данных, описанных в стандарте. Другие

[1] Формат XDR разработан фирмой Sun Microsystems в 1987 году, новая версия формата описана в RFC 1832.

(менее переносимые) способы получения представления объектов описаны в главе 20. Модуль определяет два класса: для упаковки данных в XDR-представление и для их распаковки.

Packer()
Экземпляры этого класса реализуют упаковку объектов в XDR-представление.

Unpacker(*data*)
Экземпляры этого класса позволяют получить объекты из их XDR-представления, заданного в строке (или другом объекте, имеющем интерфейс буфера) *data*.

Экземпляры класса `Packer` имеют следующие методы:

get_buffer()
Возвращает текущее содержимое буфера с упакованными данными в виде строки (данные их буфера не удаляются).

reset()
Очищает буфер.

pack_uint(*value*)
pack_int(*value*)
pack_enum(*value*)
pack_bool(*value*)
pack_uhyper(*value*)
pack_hyper(*value*)
pack_float(*value*)
pack_double(*value*)
Записывают в буфер XDR-представление данных *value* соответствующих простых типов.

pack_fstring(*n*, *s*)
pack_fopaque(*n*, *s*)
Записывают в буфер XDR-представление строки *s* фиксированной длины *n*. Заметим, что длина строки не сохраняется в представлении. Если необходимо, строка дополняется нулевыми байтами для обеспечения выравнивания к 4 байтам.

pack_string(*s*)
pack_opaque(*s*)
pack_bytes(*s*)
Записывают в буфер XDR-представление строки *s* переменной длины (сначала упаковывается длина строки в виде беззнакового целого, затем сама строка).

pack_list(*list*, *pack_item*)
Записывают в буфер XDR-представление гомогенной последовательности *list*,

длина которой заранее неизвестна. Для каждого элемента последовательности сначала упаковывается `1` (как беззнаковое целое), затем значение элемента, с помощью метода (функции) *pack_item*. По достижении конца списка упаковывается `0` (как беззнаковое целое).

Например, для упаковки списка целых чисел можно использовать примерно следующий код:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

pack_farray(*n*, *array*, *pack_item*)

Записывают в буфер XDR-представление гомогенной последовательности *array* фиксированной длины *n*. Длина последовательности не сохраняется, но, если указанная длина последовательности не равна `len(`*array*`)`, генерируется исключение `ValueError`. Для упаковки каждого элемента используется метод (функция) *pack_item*.

pack_array(*array*, *pack_item*)

Записывают в буфер XDR-представление гомогенной последовательности *array* переменной длиной. Сначала сохраняется длина последовательности, затем, с помощью метода (функции) *pack_item* упаковываются элементы последовательности.

Экземпляры класса `Unpacker` предоставляют методы для выполнения обратного преобразования:

reset(*data*)

Устанавливает буфер с упакованными данными равным строке *data*.

get_position()

Возвращает текущую позицию в буфере.

set_position(*pos*)

Устанавливает указатель на позицию *pos* в буфере. Будьте осторожны, используя этот метод.

get_buffer()

Возвращает текущее значение буфера с упакованными данными в виде строки.

done()

Указывает на завершение распаковки данных. Генерирует исключение, если не все данные были распакованы.

unpack_uint()

unpack_int()

unpack_enum()

unpack_bool()

unpack_uhyper()

unpack_hyper()

unpack_float()

unpack_double()

Интерпретирует данные в буфере как XDR-представление значения соответствующего простого типа и возвращает распакованное значение.

unpack_fstring(*n*)

unpack_fopaque(*n*)

Интерпретирует данные в буфере как XDR-представление строки фиксированной длины *n*. Ожидается, что представление строки дополнено нулевыми байтами для обеспечения выравнивания к 4 байтам. Возвращает распакованное значение в виде строки.

unpack_string()

unpack_opaque()

unpack_bytes()

Интерпретирует данные в буфере как XDR-представление строки переменной длины. Ожидается, что длина строки упакована в виде целого беззнакового числа. Возвращает распакованное значение в виде строки.

unpack_list(*unpack_item*)

Интерпретирует данные в буфере как XDR-представление последовательности неизвестной длины. Ожидается, что перед каждым элементом последовательности упакована `1` в виде целого беззнакового, а в конце последовательности упакован `0`. Для распаковки элементов последовательности использует метод (функцию) *unpack_item*, распакованную последовательность возвращает в виде списка.

unpack_farray(*n*, *unpack_item*)

Интерпретирует данные в буфере как XDR-представление последовательности фиксированной длины *n*. Для распаковки элементов последовательности использует метод (функцию) *unpack_item*, распакованную последовательность возвращает в виде списка.

unpack_array(*unpack_item*)

Интерпретирует данные в буфере как XDR-представление последовательности переменной длины. Ожидается, что длина последовательности упакована в виде целого беззнакового числа. Для распаковки элементов последовательности использует метод (функцию) *unpack_item*, распакованную последовательность возвращает в виде списка.

Модуль также определяет иерархию исключений:

Error

Базовый класс для всех исключений модуля. Его экземпляры имеют один публичный атрибут данных `msg`, содержащий пояснение к ошибке.

ConversionError

Исключения этого класса генерируются методами `unpack_*()` экземпляров класса `Unpacker`, если данные в буфере с XDR-представлением не соответствуют ожидаемому формату, и методами `pack_*()` класса `Packer`, если указанное значение не может быть упаковано данным методом.

Глава 29

Средства работы с языками структурной разметки

Модули, описанные в этой главе, предоставляют средства для работы с различными языками структурной разметки: SGML (Standard Generalized Markup Language, стандартный язык обобщенной разметки), HTML (Hypertext Markup Language, язык разметки гипертекста) и XML (Extensible Markup Language, расширяемый язык разметки).

sgmllib	Минимальные средства работы с SGML (то, что необходимо для обработки HTML).
htmllib	Обработка HTML-документов.
htmlentitydefs	Определения сущностей HTML.
xml.parsers.expat	Быстрая обработка XML-документов (без проверки на соответствие DTD) с помощью библиотеки `Expat` языка C.
xml.sax	SAX2 интерфейс к синтаксическим анализаторам XML-документов.
xml.sax.handler	Базовые классы для обработчиков SAX-событий.
xml.sax.saxutils	Вспомогательные средства для приложений, использующих SAX.
xml.sax.xmlreader	Интерфейс объектов, реализующих преобразование XML-документов в поток SAX-событий.
xmllib	Обработка XML-документов.

29.1 `sgmllib` — обработка SGML-документов

Этот модуль определяет класс `SGMLParser`, реализующий обработку текста в формате SGML (Standard Generalized Markup Language, стандартный язык обобщенной разметки). На самом деле он воспринимает только ту часть синтаксиса SGML, которая используется в HTML (модуль `sgmllib` служит базой для модуля `htmllib`).

`SGMLParser()`

Возвращает объект, реализующий обработку текста в формате SGML. Воспринимаются следующие конструкции:

- Открывающие и закрывающие теги в виде '<*тег атрибут*="*значение*"...>' и '</*тег*>' соответственно.
- Ссылки на символы в виде '&#*код*;'.
- Ссылки на сущности (entity) в виде '&*имя*;'.
- Комментарии в виде '<!--*текст*-->'.

Экземпляры класса `SGMLParser` имеют следующие интерфейсные методы (переопределяя эти методы в производных классах, Вы контролируете обработку размеченного текста, то есть, определяете тип документа — DTD):

reset()
Возвращает экземпляр в исходное состояние. Все необработанные данные теряются. Этот метод неявно вызывается при инициализации.

setnomoretags()
Останавливает обработку тегов. Весь последующий текст воспринимается без обработки (CDATA). Этот метод необходим для реализации обработки тега '<PLAINTEXT>' в HTML.

setliteral()
Временно приостанавливает обработку тегов. Последующий текст воспринимается без обработки (CDATA).

feed(*data*)
Передает экземпляру размеченный текст *data* для обработки. Обрабатываются только завершенные данные, остальное остается в буфере до тех пор, пока не будет передана следующая порция данных или не будет вызван метод `close()`.

close()
Завершает обработку всех данных в буфере. Вы можете переопределить этот метод в производном классе, добавив дополнительные действия. Переопределенный метод всегда должен вызывать исходную версию метода.

get_starttag_text()
Возвращает текст последнего открывающего тега. Этот метод может быть полезен для регенерации исходного размеченного текста с минимальными изменениями.

handle_starttag(*tag*, *method*, *attributes*)
Вызывается для обработки открывающих тегов, для которых определен метод `start_`*tag*`()` или `do_`*tag*`()`. Аргумент *tag* является строкой с именем тега, приведенном к нижнему регистру, *method* — метод, который должен быть вызван для интерпретации открывающего тега. Атрибут *attributes* является списком пар '(*name*, *value*)' для всех атрибутов, найденных внутри угловых скобок тега, где *name* — имя атрибута, приведенное к нижнему регистру, и *value* — его значение. Исходная реализация этого метода вызывает *method*(*attributes*).

handle_endtag(*tag*, *method*)
Вызывается для обработки закрывающих тегов, для которых определен метод end_*tag*(). Аргумент *tag* является строкой с именем тега, приведенном к нижнему регистру, *method* — метод, который должен быть вызван для интерпретации закрывающего тега. Исходная реализация этого метода просто вызывает *method*().

handle_data(*data*)
Вызывается для обработки простого текста. Вам следует переопределить его в производном классе: исходная реализация ничего не делает.

handle_charref(*ref*)
Вызывается для обработки ссылок на символы в виде '&#*ref*;'. Исходная реализация преобразует число в диапазоне от 0 до 255 в символ с помощью встроенной функции chr() и вызывает с ним метод handle_data(). Если аргумент *ref* неверен или выходит за пределы указанного диапазона, вызывает unknown_charref(*ref*) для обработки ошибки.

handle_entityref(*ref*)
Вызывается для обработки ссылок на сущности в виде '&*ref*;'. Исходная реализация использует атрибут экземпляра (или класса) entitydefs, являющийся отображением имен сущностей к строкам. Если это отображение содержит запись с ключом *ref*, вызывает метод handle_data() с результатом в качестве аргумента, в противном случае вызывает unknown_entityref(*ref*). По умолчанию атрибут entitydefs содержит записи с ключами 'amp', 'apos', 'gt', 'lt' и 'quot'.

handle_comment(*comment*)
Вызывается для обработки комментария. В качестве аргумента *comment* используется строка с текстом, содержащимся между '<!--' и '-->'. Исходная реализация ничего не делает.

report_unbalanced(*tag*)
Этот метод вызывается, если в размеченном тексте найден закрывающий тег *tag*, для которого не было открывающего тега.

unknown_starttag(*tag*)
Вызывается для обработки неизвестного открывающего тега (если нет соответствующего метода start_*tag*() или do_*tag*()). Вам следует переопределить этот метод в производном классе: исходная реализация ничего не делает.

unknown_endtag(*tag*)
Вызывается для обработки неизвестного закрывающего тега (если нет соответствующего метода end_*tag*()). Вам следует переопределить этот метод в производном классе: исходная реализация ничего не делает.

unknown_charref(*ref*)
Вызывается для обработки неверной ссылки на символ в виде '&#*ref*;' (см. описание метода handle_charref()). Вам следует переопределить этот метод в производном классе: исходная реализация ничего не делает.

unknown_entityref(*ref*)
: Вызывается для обработки ссылки на сущность с неизвестным именем в виде '&*ref*;' (см. описание метода handle_entityref()). Вам следует переопределить этот метод в производном классе: исходная реализация ничего не делает.

Помимо переопределения некоторых из перечисленных выше методов, в производных классах Вам следует определить методы для обработки тегов. Имена тегов в размеченном тексте могут содержать буквы в любом регистре: перед вызовом метода имя будет приведено к нижнему регистру (то есть *tag* в именах методов, описанных ниже, является именем тега, приведенного к нижнему регистру). В качестве атрибута *attributes* используется список пар '(*name*, *value*)' для всех атрибутов, найденных внутри угловых скобок тега, где *name* — имя атрибута, приведенное к нижнему регистру, и *value* — его значение.

start_*tag*(*attributes*)
: Вызывается для обработки открывающего тега *tag*. Этот метод имеет преимущество над методом do_*tag*().

do_*tag*(*attributes*)
: Вызывается для обработки открывающего тега *tag*, для которого не должно быть соответствующего закрывающего тега.

end_*tag*()
: Вызывается для обработки закрывающего тега *tag*.

Обработчик хранит стек открытых элементов, для которых еще не было закрывающих тегов. В этот стек помещаются только теги, для которых есть соответствующий метод start_*tag*(). Определение метода end_*tag*() не является обязательным. Для тегов, обрабатываемых методами do_*tag*() и unknown_tag() нет смысла определять соответствующие методы end_*tag*() — они не будут использованы.

29.2 htmllib — обработка HTML-документов

Этот модуль определяет класс HTMLParser (производный от sgmllib.SGMLParser), который может быть использован в качестве базового для класса, реализующего обработку текста в формате HTML (Hypertext Markup Language, язык разметки гипертекста). В настоящий момент этот класс поддерживает HTML 2.0[1]. Для получения вывода экземпляры HTMLParser и производных от него классов вызывают методы указанного объекта, реализующего форматирование текста. Стандартный модуль formatter предоставляет два варианта таких объектов (смотрите описание модуля для получения информации об интерфейсе объектов, реализующих форматирование).

[1] http://www.w3.org/hypertext/WWW/MarkUp/html-spec/html-spec_toc.html

HTMLParser(*formatter*)
Создает и возвращает объект, реализующий обработку текста в формате HTML. Этот класс поддерживает все имена сущностей, требуемых спецификацией HTML 2.0 (определены в модуле htmlentitydefs), и определяет обработчики для всех элементов HTML 2.0 и многих элементов HTML 3.0 и 3.2.

В дополнение к методам, реализующий обработку тегов (как того требует интерфейс класса sgmllib.SGMLParser), класс HTMLParser предоставляет следующие дополнительные методы и атрибуты данных:

formatter
Объект, реализующий форматирование текста, который был передан в качестве аргумента при инициализации.

nofill
Если этот флаг является ложью, идущие друг за другом символы пропуска будут свернуты в один пробел методом handle_data() или save_end(), в противном случае будут оставлены без изменений. Обычно он должен быть истинным только внутри элемента <pre>.

anchor_bgn(*href*, *name*, *type*)
Вызывается в начале региона, помеченного якорем. В качестве аргументов используются значения одноименных атрибутов тега <a> (если тег не содержит какого-либо атрибута, в качестве аргумента для него используется пустая строка). Исходная реализация сохраняет все ссылки, содержащиеся в документе, в списке, доступном через атрибут anchorlist.

anchor_end()
Вызывается в конце региона, помеченного якорем. Исходная реализация добавляет текстовую ссылку в виде '[*index*]', где *index* — индекс соответствующего элемента в списке ссылок (атрибут anchorlist).

handle_image(*src*, *alt* [, *ismap* [, *align* [, *width* [, *height*]]]])
Вызывается для обработки вставок изображений. В качестве аргументов используются значения одноименных атрибутов тега <img> (аргументы *width* и *height* являются целыми числами). Исходная реализация вызывает handle_data(*alt*).

save_bgn()
Устанавливает режим, в котором все символьные данные сохраняются во внутреннем буфере вместо того, чтобы быть переданными объекту, реализующему форматирование (аргумент *formatter* конструктора). Вы можете извлечь сохраненные данные с помощью метода save_end().

save_end()
Заканчивает работу в режиме, в котором все символьные данные сохраняются во внутреннем буфере, и возвращает все сохраненные данные. Если атрибут nofill является ложью, последовательности символов пропуска заменяются одним пробелом.

29.3 htmlentitydefs — определения сущностей HTML

Этот модуль определяет один словарь, который используется модулем htmllib в качестве значения атрибута entitydefs класса htmllib.HTMLParser. Этот словарь содержит записи для всех сущностей (entity) HTML 2.0, отображая их имена в символы в кодировке Latin-1 (ISO-8859-1). Сущности, которые не могут быть представлены литерально в рамках Latin-1, отображаются в запись вида '&#*ref*;', где *ref* — код символа в Unicode.

entitydefs

Словарь, отображающий имена сущностей HTML 2.0 в символ кодировки Latin-1 или ссылку на символ Unicode в виде '&#*ref*;'.

29.4 xml.parsers.expat — быстрая обработка XML-документов с помощью библиотеки Expat

Модуль xml.parsers.expat доступен, начиная с версии 2.0 и предоставляет интерфейс библиотеке Expat, реализующей быструю обработку XML-документов без проверки на соответствие DTD[2]. Модуль определяет следующие две функции, исключение и подмодуль:

ErrorString(*err_code*)

Возвращает строку с пояснением к ошибке с кодом *err_code*.

ParserCreate([*encoding* [, *namespace_separator*]])

Создает и возвращает объект, реализующий обработку текста в формате XML. Если задан и не равен None, он должен быть строкой, указывающей кодировку: 'UTF-8', 'UTF-16' или 'ISO-8859-1' (по умолчанию обработчик автоматически выбирает между UTF-8 и UTF-16).

Если задан и не равен None аргумент *namespace_separator* (должен быть строкой из одного символа), включается обработка пространств имен. В этом случае имена элементов и атрибутов, определенных в каком-либо пространстве имен, будут передаваться функциям StartElementHandler() и EndElementHandler() в виде '*namespace_URI* + *namespace_separator* + *name*', где *namespace_URI* — URI пространства имен и *name* — имя элемента или атрибута. Если аргумент *namespace_separator* равен '\000', в качестве разделителя используется пустая строка.

Например, пусть в качестве аргумента *namespace_separator* используется ' ' и обрабатывается следующий документ:

[2]Модуль xml.parsers.expat является надстройкой над модулем pyexpat. Использовать модуль pyexpat напрямую не рекомендуется.

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

В этом случае функция `StartElementHandler()` получит следующие строки в качестве аргументов:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

error

Исключения этого класса генерируются, если XML-документ содержит ошибки.

errors

Подмодуль, в котором определены константы вида `XML_ERROR_*` для возможных ошибок обработки. Вы можете использовать их для сравнения со значением атрибута `ErrorCode` объектов, возвращаемых функцией-конструктором `ParserCreate()`.

XMLParserType

Тип объектов, возвращаемых функцией `ParserCreate()`.

Объекты, возвращаемые функцией-конструктором `ParserCreate()`, имеют следующие методы:

Parse(*data* [, *isfinal*])

Обрабатывает содержимое строки *data*, вызывая соответствующие функции. При последнем вызове аргумент *isfinal* должен являться истиной.

ParseFile(*file*)

Считывает данные из файла, представленного файловым или подобным объектом *file* (должен поддерживать вызов метода `read()` с одним аргументом, возвращающим пустую строку, если больше нет данных).

SetBase(*base*)

Устанавливает базовый адрес, который должен быть использован для относительных URI в системных объявлениях: он будет передан в качестве аргумента *base* функциям `ExternalEntityRefHandler`, `NotationDeclHandler` и `UnparsedEntityDeclHandler`.

GetBase()

Возвращает строку с базовым адресом, установленным ранее с помощью метода `SetBase()`, или `None`, если метод `SetBase()` не был использован.

Объекты, возвращаемые функцией-конструктором `ParserCreate()`, имеют следующие атрибуты:

returns_unicode

Если этот атрибут равен `1` (по умолчанию), функциям-обработчикам, описанным ниже, будут передаваться строки Unicode, если же он равен `0` — обычные строки, содержащие данные в кодировке UTF-8. Вы можете изменять значение этого атрибута в любое время, задавая таким образом тип результата.

Значения следующих атрибутов относятся к последней ошибке. Эти атрибуты имеют корректные значения только после того, как метод `Parse()` или `ParseFile()` сгенерировал исключение `error`.

ErrorByteIndex

Индекс (в байтах), указывающий на место, в котором возникла ошибка.

ErrorCode

Числовой код ошибки. Вы можете использовать это значение в качестве аргумента функции `ErrorString()` или сравнивать его со значениями констант, определенных в подмодуле `errors`.

ErrorColumnNumber

Позиция в строке, в которой возникла ошибка.

ErrorLineNumber

Номер строки, в которой возникла ошибка.

Ниже приведен список обработчиков, которые Вы можете установить для объекта *obj* с помощью инструкций вида '*obj*.*handler* = *func*', где *handler* должен быть одним из перечисленных атрибутов и *func* — объектом, поддерживающим вызов с указанным количеством аргументов. Все аргументы, если не указано иного, являются строками.

StartElementHandler(*name*, *attributes*)

Вызывается для обработки открывающего тега каждого элемента. Аргумент *name* содержит имя элемента и *attributes* — словарь, отображающий имена атрибутов к их значениям.

EndElementHandler(*name*)

Вызывается для обработки закрывающего тега каждого элемента. Аргумент *name* содержит имя элемента.

ProcessingInstructionHandler(*name*, *data*)

Вызывается для каждой инструкции обработки (PI, Processing Instruction) в виде '<?*name* *data*?>' (но не для '<?xml *data*?>').

CharacterDataHandler(*data*)

Вызывается для обработки простого текста.

UnparsedEntityDeclHandler(*entityName*, *base*, *systemId*, *publicId*, *notationName*)

Вызывается для необрабатываемых объявлений сущностей (NDATA).

NotationDeclHandler(*notationName, base, systemId, publicId*)
Вызывается для обработки объявлений примечаний (‘<!NOTATION ...>’).

StartNamespaceDeclHandler(*prefix, uri*)
Вызывается, если элемент (открывающий тег) содержит объявление пространства имен.

EndNamespaceDeclHandler(*prefix*)
Вызывается для закрывающего тега элемента, содержащего объявление пространства имен.

CommentHandler(*data*)
Вызывается для комментариев.

StartCdataSectionHandler()
Вызывается в начале секции CDATA.

EndCdataSectionHandler()
Вызывается в конце секции CDATA.

DefaultHandler(*data*)
Вызывается для частей XML-документа, для которых не определен соответствующий обработчик. Части документа передаются в качестве аргумента *data* в исходном виде.

DefaultHandlerExpand(*data*)
Вызывается для частей XML-документа, для которых не определен соответствующий обработчик. Части документа передаются в качестве аргумента *data* после замены ссылок на сущности.

NotStandaloneHandler()
Вызывается, если XML-документ был объявлен как несамостоятельный (‘standalone="no"’ в заголовке).

ExternalEntityRefHandler(*context, base, systemId, publicId*)
Вызывается для обработки ссылок на внешние сущности.

Приведем простейший пример использования модуля:

```
from xml.parsers import expat

def start_element(name, attrs):
    print 'Начало элемента:', name, attrs
def end_element(name):
    print 'Конец элемента:', name
def char_data(data):
    print 'Символьные данные:', repr(data)
```

```
p = expat.ParserCreate()

p.returns_unicode = 0
p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""\
<?xml version="1.0"?>
<parent id="top">
  <child1 name="paul">Text goes here</child1>
  <child2 name="fred">More text</child2>
</parent>""")
```

Вывод этой программы будет следующим:

```
Начало элемента: parent {'id': 'top'}
Начало элемента: child1 {'name': 'paul'}
Символьные данные: 'Text goes here'
Конец элемента: child1
Символьные данные: '\012'
Начало элемента: child2 {'name': 'fred'}
Символьные данные: 'More text'
Конец элемента: child2
Символьные данные: '\012'
Конец элемента: parent
```

29.5 `xml.sax` — SAX2 интерфейс к синтаксическим анализаторам XML-документов

Этот модуль доступен, начиная с версии 2.0 и реализует SAX (Simple API for XML, простой программный интерфейс для XML) интерфейс.

Модуль `xml.sax` определяет следующие функции:

make_parser([*parser_list*])
Создает и возвращает объект с SAX интерфейсом (экземпляр `xml.sax.xmlreader.XMLReader` или производного от него класса), реализующий синтаксический анализ XML-документа. Для этого используется первый найденный анализатор. Поиск модулей синтаксических анализаторов производится в списке *parser_list* (список имен модулей), если он задан, затем в списке модулей по умолчанию.

parse(*filename_or_stream*, *handler* [, *error_handler*])
Создает объект, реализующий синтаксический анализ XML-документа,

и использует его для анализа файла *filename_or_stream*. Аргумент *filename_or_stream* должен быть файловым (или подобным) объектом или строкой с именем файла. Аргумент *handler* должен быть экземпляром класса ContentHandler, реализующим обработку событий для содержимого документа. С помощью аргумента *error_handler* (должен быть экземпляром класса ErrorHandler) Вы можете задать способ обработки ошибок, по умолчанию для всех ошибок будет генерироваться исключение SAXParseException.

parseString(*string*, *handler* [, *error_handler*])
Работает аналогично функции parse(), но обрабатывает XML-документ, содержащийся в строке *string*.

Типичное SAX-приложение состоит из объекта, реализующего чтение и синтаксический анализ документа, преобразуя его в поток SAX-событий, объектов-обработчиков событий и объекта, представляющего сам XML-документ. Для всех этих объектов существенными являются только интерфейсы, формально представленные различными классами. Классы InputSource, Locator, + AttributesImpl и XMLReader определены в модуле xml.sax.xmlreader, интерфейсные классы обработчиков событий — в модуле xml.sax.handler. Для удобства модуль xml.sax импортирует в свое глобальное пространство имен классы InputSource, ContentHandler и ErrorHandler.

Модуль также определяет иерархию исключений:

SAXException
Этот класс является базовым для всех исключений модуля. Экземпляры SAXException и производных от него классов заключают в себе сообщение об ошибке и, часто, реальное исключение, сгенерированное выбранным синтаксическим анализатором. Эта информация доступна через методы исключения getMessage() и getException() соответственно (последний может возвращать None).

SAXParseException
Исключения этого класса генерируются при обнаружении синтаксических ошибок в анализируемом XML-документе. Экземпляры класса SAXParseException имеют дополнительные методы getColumnNumber(), getLineNumber(), getPublicId() и getSystemId(), которые возвращают, соответственно, позицию в строке и номер строки, в которой возникла ошибка, и публичный и системный идентификаторы сущностей или None.

SAXNotRecognizedException
Исключения этого класса генерируются, если синтаксический анализатор сталкивается с неизвестной особенностью или свойством.

SAXNotSupportedException
Исключения этого класса генерируются при попытке использования возможности, которая не поддерживается данным синтаксическим анализатором.

SAXReaderNotAvailable
Исключения этого класса генерируются, если запрошенный анализатор не

может быть импортирован или не может выполнить внешнюю программу, необходимую для его работы.

29.6 xml.sax.handler — базовые классы для обработчиков SAX-событий

Этот модуль (доступен, начиная с версии 2.0) определяет базовые классы с интерфейса SAX (см. описание модуля xml.sax) для обработки содержимого XML-документа, его DTD (Document Type Definition, определение типа документа), ошибок и ссылок на сущности. В производных классах необходимо переопределить только методы для интересующих Вас событий.

ContentHandler()
: Этот класс определяет интерфейс обработчиков SAX-событий для содержимого XML-документа и является наиболее важным в SAX.

DTDHandler()
: Определяет интерфейс обработчика событий для DTD (объявлений необрабатываемых сущностей и примечаний).

ErrorHandler()
: Определяет интерфейс обработчика ошибок.

EntityResolver()
: Определяет интерфейс обработчика ссылок на внешние сущности.

Помимо интерфейсных классов модуль xml.sax.handler определяет символические константы для имен функциональных особенностей (feature_*) и свойств (property_*). Эти константы могут быть использованы в качестве аргументов методов getFeature(), setFeature(), getProperty() и setProperty() экземпляров xml.sax.xmlreader.XMLReader или производных от него классов. Заметим, что включение/выключение используемых особенностей возможно только до начала синтаксического анализа XML-документа.

feature_namespaces
: Выполнять обработку пространств имен (является истиной по умолчанию).

feature_namespace_prefixes
: Сообщать исходные имена с приставками элементов и атрибутов, используемые при объявлении пространств имен (по умолчанию является ложью).

feature_string_interning
: Обрабатывать с помощью встроенной функции intern() все приставки, имена элементов и атрибутов, URI пространств имен и т. д. (по умолчанию является ложью, то есть такая обработка не обязательна).

feature_validation
Сообщать о несоответствии документа DTD.

feature_external_ges
Подключать внешние объявления текстовых сущностей.

feature_external_pes
Подключать внешние объявления сущностей параметров, включая внешнее подмножество DTD.

all_features
Список, содержащий все известные особенности.

property_lexical_handler
Необязательный обработчик лексических событий (например, комментариев). В версии 2.0 не поддерживается.

property_declaration_handler
Необязательный обработчик событий, связанных с DTD (кроме необрабатываемых сущностей и примечаний). В версии 2.0 не поддерживается.

property_dom_node
До начала синтаксического анализа — корневой, во время анализа — текущий узел объектной модели документа (DOM). Изменение этого свойства возможно только до начала анализа документа.

property_xml_string
Строка, содержащая фрагмент исходного текста документа для текущего события. Свойство доступно только для чтения.

all_properties
Список всех известных имен свойств.

29.6.1 Интерфейс класса `ContentHandler`

Интерфейс класса `ContentHandler` составляют следующие методы:

setDocumentLocator(*locator*)
Вызывается синтаксическим анализатором XML-документа перед началом анализа. Объект *locator* (должен быть экземпляром `xml.sax.xmlreader.Locator`) предоставляет возможность получить информацию о позиции в документе соответствующей текущему событию. Эта информация обычно используется для вывода сообщений об ошибке. Заметим, что методы объекта *locator* должны возвращать корректный результат только во время обработки события.

startDocument()
Вызывается один раз для извещения о начале документа.

endDocument()
Вызывается один раз для извещения о конце документа (при достижении конца ввода или возникновении ошибки, не допускающей восстановления).

startPrefixMapping(*prefix*, *uri*)
Вызывается в начале области действия пространства имен. Обработка этого события не является необходимой: синтаксический анализатор XML-документа автоматически заменяет приставку *prefix* на URI *uri* в именах элементов и атрибутов, если включена возможность `feature_namespaces` (эта возможность включена по умолчанию). Метод `startPrefixMapping()` всегда вызывается до соответствующего метода `startElement()`, однако правильный порядок следования нескольких событий `startPrefixMapping()` не гарантируется.

endPrefixMapping(*prefix*)
Вызывается в конце области действия пространства имен. Метод `endPrefixMapping()` всегда вызывается до соответствующего метода `endElement()`, однако правильный порядок следования нескольких событий `endPrefixMapping()` не гарантируется.

startElement(*name*, *attrs*)
Вызывается в начале элемента (без учета пространств имен). В качестве аргумента *name* используется строка с необработанным именем элемента, в качестве *attrs* — экземпляр класса `xml.sax.xmlreader.AttributesImpl`, описывающий атрибуты элемента.

endElement(*name*)
Вызывается в начале элемента (без учета пространств имен). В качестве аргумента *name* используется строка с необработанным именем элемента.

startElementNS(*name*, *qname*, *attrs*)
Вызывается в начале элемента (с учетом пространств имен). В качестве аргумента *name* используется кортеж, содержащий URI и имя элемента, в качестве *qname* — строка с необработанным именем элемента (может быть `None`, если отключена возможность `feature_namespaces`) и в качестве *attrs* — экземпляр класса `xml.sax.xmlreader.AttributesNSImpl`, описывающий атрибуты элемента.

endElementNS(*name*, *qname*)
Вызывается в конце элемента (с учетом пространств имен). В качестве аргумента *name* используется кортеж, содержащий URI и имя элемента, в качестве *qname* — строка с необработанным именем элемента (может быть `None`, если отключена возможность `feature_namespaces`).

characters(*content*)
Вызывается для символьных данных (аргумент *content*). Синтаксический анализатор документа может передавать непрерывный поток символьных данных этому методу несколькими порциями. Все символы, переданные методу одной порцией должны быть из одного элемента. Аргумент *content* может быть как обычной строкой, так и строкой Unicode.

ignorableWhitespace(*whitespace*)
Вызывается для символов пропуска (аргумент *whitespace*) в содержимом элемента, которые могут быть проигнорированы. Синтаксический анализатор документа может передавать непрерывный поток символов пропуска этому методу несколькими порциями. Все символы пропуска, переданные методу одной порцией должны быть из одного элемента.

processingInstruction(*target*, *data*)
Вызывается для инструкций обработки вида '<?*target data*?>'. Этот метод не должен вызываться для объявлений вида '<?xml ...>'.

skippedEntity(*name*)
Вызывается для всех сущностей, обработка которых синтаксическим анализатором не выполняется (например, если сущность объявлена во внешнем подмножестве DTD).

29.6.2 Интерфейс класса DTDHandler

Интерфейс класса DTDHandler составляют следующие методы:

notationDecl(*name*, *publicId*, *systemId*)
Вызывается для обработки объявлений примечаний ('<!NOTATION ...>').

unparsedEntityDecl(*name*, *publicId*, *systemId*, *ndata*)
Вызывается для обработки объявлений необрабатываемых сущностей (NDATA).

29.6.3 Интерфейс класса ErrorHandler

Интерфейс класса ErrorHandler составляют следующие методы:

error(*exception*)
Вызывается для обработки ошибок, допускающих продолжение дальнейшей работы. Исходная реализация генерирует исключение *exception*.

fatalError(*exception*)
Вызывается для обработки фатальных ошибок, то есть, не допускающих продолжение дальнейшей работы. Исходная реализация генерирует исключение *exception*.

warning(*exception*)
Вызывается для предупреждений. Исходная реализация выводит на стандартный поток вывода (sys.stdout) строковое представление исключения *exception*.

29.6.4 Интерфейс класса `EntityResolver`

Интерфейс класса `EntityResolver` составляют следующие методы:

resolveEntity(*publicId, systemId*)
Вызывается для необрабатываемых сущностей. Должен возвращать либо строку с системным идентификатором, либо экземпляр класса `xml.sax.xmlreader.InputSource`, который будет использован для считывания данных. Исходная реализация возвращает *systemId*.

29.7 `xml.sax.saxutils` — вспомогательные средства для приложений, использующих SAX

Этот модуль (доступен, начиная с версии 2.0) определяет несколько классов и функций, которые обычно полезны при создании приложений, использующих SAX.

escape(*data* [, *entities*])
Заменяет символы '&', '<' и '>' в строке *data* соответствующими ссылками на сущности и возвращает полученную строку. Вы можете произвести дополнительные замены, передав в качестве аргумента *entities* словарь, ключи которого будут заменены соответствующими им значениями.

XMLGenerator([*out* [, *encoding*]])
Этот класс является производным от `ContentHandler`, его методы реализуют запись SAX-событий назад в XML-документ. Другими словами, при использовании экземпляра класса `XMLGenerator` в качестве объекта, реализующего обработку содержимого XML-документа, будет воспроизведен исходный документ. Запись документа производится в файловый (или подобный) объект *out*, по умолчанию используется *sys.stdout*. Аргумент *encoding* задает кодировку создаваемого документа (по умолчанию используется `'iso-8859-1'`).

XMLFilterBase(*base*)
Этот класс предназначен для использования в качестве "прослойки" между объектом *base* (экземпляр `xml.sax.xmlreader.XMLReader` или производного от него класса), реализующим синтаксический анализ XML-документа, и обработчиками SAX-событий. Исходная реализация просто переадресует события соответствующим обработчикам. В производных классах Вы можете переопределить определенные методы для того, чтобы изменять поток событий при их поступлении.

prepare_input_source(*source* [, *base*])
Возвращает экземпляр класса `xml.sax.xmlreader.InputSource`, готовый для чтения документа. Аргумент *source* может быть строкой с системным идентификатором, файловым (или подобным) объектом или экземпляром класса

xml.sax.xmlreader.InputSource. Аргумент *base* (по умолчанию равен пустой строке) используется в качестве основы URL (точки отсчета).

Функция prepare_input_source() обычно используется для реализации полиморфного восприятия аргумента *source* метода parse() объектов, реализующих чтение и синтаксический анализ XML-документов.

29.8 xml.sax.xmlreader — интерфейс объектов, реализующих чтение и синтаксический анализ XML-документов

Каждый синтаксический анализатор XML-документов с SAX-интерфейсом должен быть реализован в виде модуля на языке Python, в котором должна быть определена функция create_parser(), возвращающая объект-анализатор. Эта функция вызывается без аргументов функцией xml.sax.make_parser() для создания нового объекта. Все синтаксические анализаторы XML-документов должны иметь интерфейс, аналогичный классу XMLReader.

Модуль xml.sax.xmlreader определяет следующие классы:

XMLReader()
: Базовый класс для всех синтаксических анализаторов XML-документов. Его методы ничего не делают и лишь определяют SAX-интерфейс.

IncrementalParser()
: Метод parse() блокирует выполнение до тех пор, пока не будет прочитан весь документ. Данный класс предоставляет несколько дополнительных методов, которые позволяют подавать документ анализатору по частям. Класс IncrementalParser() также определяет метод parse(), реализованный с использованием этих методов. Этот класс является наиболее удобным для использования в качестве базового при написании синтаксических анализаторов.

Locator()
: Интерфейсный класс объектов, которые используются для ассоциации SAX-событий с позицией в документе.

InputSource([*systemId*])
: Экземпляры этого класса заключают в себе информацию о документе, необходимую для его анализа: публичный и системный идентификаторы, файловый объект, информацию о кодировке. Синтаксический анализатор не должен вносить изменения в этот объект, но может сделать его копию.

AttributesImpl(*attrs*)
: Экземпляры этого класса используются для представления атрибутов элемента при вызове метода startElement() обработчика событий. Аргумент *attrs* должен быть отображением имен атрибутов к их значениям.

AttributesNSImpl(*attrs*, *qnames*)

Этот класс является производным от AttributesImpl, его экземпляры используются для представления атрибутов элемента с учетом пространства имен при вызове метода startElement() обработчика событий. Аргументы *attrs* и *qnames* должны быть отображениями, в качестве ключей в которых используются кортежи с URI пространства имен и локальным именем, а в качестве значений — значения и полные имена атрибутов соответственно.

29.8.1 Интерфейс класса XMLReader

Интерфейс класса XMLReader составляют следующие методы:

parse(*source*)

Анализирует XML-документ, формируя поток событий. Аргумент *source* должен быть строкой с системным идентификатором (обычно имя файла или URL), файловым (или подобным) объектом или экземпляром класса InputSource.

getContentHandler()

Возвращает текущий объект, реализующий обработку событий, касающихся содержимого документа (экземпляр xml.sax.handler.ContentHandler или производного от него класса).

setContentHandler(*handler*)

Запоминает *handler* как объект, реализующий обработку событий, касающихся содержимого документа (экземпляр xml.sax.handler.ContentHandler или производного от него класса). По умолчанию (до вызова этого метода) используется объект, игнорирующий все события.

getDTDHandler()

Возвращает текущий объект, реализующий обработку событий, касающихся DTD (экземпляр xml.sax.handler.DTDHandler или производного от него класса).

setDTDHandler(*handler*)

Запоминает *handler* как объект, реализующий обработку событий, касающихся DTD (экземпляр xml.sax.handler.DTDHandler или производного от него класса). По умолчанию (до вызова этого метода) используется объект, игнорирующий все события.

getEntityResolver()

Возвращает текущий объект, реализующий обработку ссылок на внешние сущности (экземпляр xml.sax.handler.EntityResolver или производного от него класса).

setEntityResolver(*resolver*)

Запоминает *resolver* как объект, реализующий обработку ссылок на внешние сущности (экземпляр xml.sax.handler.EntityResolver или производного от

него класса). По умолчанию (до вызова этого метода) используется объект, открывающий документ, на который указывает системный идентификатор (отсутствие в определении сущности системного идентификатора приведет к ошибке).

getErrorHandler()
Возвращает текущий обработчик ошибок (экземпляр `xml.sax.handler.ErrorHandler` или производного от него класса).

setErrorHandler(*handler*)
Устанавливает *handler* в качестве текущего обработчика ошибок. По умолчанию (до вызова этого метода) используется объект, генерирующий исключение в случае ошибок и выводящий предупреждения на стандартный поток вывода (`sys.stdout`).

setLocale(*locale*)
Устанавливает язык и кодировку сообщений об ошибках и предупреждений.

getFeature(*featurename*)
Возвращает текущие установки для особенности с именем *featurename*. Если особенность неизвестна, генерирует исключение `SAXNotRecognizedException`. Символические константы для наиболее известных имен особенностей определены в модуле `xml.sax.handler`.

setFeature(*featurename*, *value*)
Изменяет установки для особенности с именем *featurename*. Для неизвестных особенностей генерирует исключение `SAXNotRecognizedException`, если синтаксический анализатор не поддерживает указанную особенность — генерирует исключение *SAXNotSupportedException*.

getProperty(*propertyname*)
Возвращает текущие установки для свойства *propertyname*. Если свойство неизвестно, генерирует исключение `SAXNotRecognizedException`. Символические константы для наиболее известных имен свойств определены в модуле `xml.sax.handler`.

setProperty(*propertyname*, *value*)
Изменяет установки для свойства *propertyname*. Для неизвестных свойств генерирует исключение `SAXNotRecognizedException`, если синтаксический анализатор не поддерживает указанное свойство — генерирует исключение *SAXNotSupportedException*.

29.8.2 Интерфейс класса `IncrementalParser`

Класс `IncrementalParser` является производным от `XMLReader`, его интерфейс имеет несколько дополнительных методов:

feed(*data*)
Анализирует порцию данных *data*.

close()
Вызов этого метода указывает на конец документа.

reset()
Этот метод должен быть вызван после метода `close()` для подготовки к анализу следующего документа.

29.8.3 Интерфейс класса Locator

Интерфейс класса `Locator` составляют следующие методы:

getColumnNumber()
Возвращает номер позиции в строке, в которой заканчиваются данные, соответствующие текущему событию (`-1`, если позиция неизвестна).

getLineNumber()
Возвращает номер строки, в которой заканчиваются данные, соответствующие текущему событию (`-1`, если номер строки неизвестен).

getPublicId()
Возвращает публичный идентификатор для текущего события (`None`, если публичный идентификатор не может быть установлен).

getSystemId()
Возвращает системный идентификатор для текущего события (`None`, если системный идентификатор не может быть установлен).

29.8.4 Экземпляры класса InputSource

Экземпляры класса `InputSource` имеют следующие методы:

setPublicId(*id*)
Устанавливает публичный идентификатор документа равным строке *id*.

getPublicId()
Возвращает публичный идентификатор документа.

setSystemId(*id*)
Устанавливает системный идентификатор документа равным строке *id*.

getSystemId()
Возвращает системный идентификатор документа.

setEncoding(*encoding*)
Устанавливает кодировку документа. Аргумент *encoding* должен быть строкой,

применимой для использования в объявлении кодировки XML-документа. Установленная кодировка будет проигнорирована синтаксическим анализатором документа, если установлен символьный поток, осуществляющий автоматическое преобразование.

getEncoding()
Возвращает кодировку документа или None, если кодировка неизвестна.

setByteStream(*bytefile*)
Устанавливает байтовый поток (файловый или подобный объект, не выполняющий преобразование байтов в символы), из которого будет производиться чтение документа. Кодировка потока может быть установлена с помощью метода setEncoding. Байтовый поток не будет использоваться синтаксическим анализатором документа, если установлен символьный поток, осуществляющий автоматическое преобразование байтов (в определенной кодировке) в символы Unicode.

getByteStream()
Возвращает байтовый поток, из которого должно производиться чтение документа. Кодировка этого потока может быть получена с помощью метода getEncoding.

setCharacterStream(*charfile*)
Устанавливает символьный поток (файловый или подобный объект, осуществляющий автоматическое преобразование байтов в определенной кодировке в символы Unicode), из которого должно производиться чтение документа.

getCharacterStream()
Возвращает символьный поток, из которого должно производиться чтение документа.

29.8.5 Экземпляры классов AttributesImpl и AttributesNSImpl

Экземпляры классов AttributesImpl и AttributesNSImpl являются отображениями, в качестве ключей в которых выступают именами атрибутов, и в качестве значений — их значения. Для экземпляров класса AttributesImpl имена атрибутов (а также аргумент *name* описанных ниже методов) являются строками, для экземпляров AttributesNSImpl — кортежами из двух строк: с URI пространства имен и локального имени атрибута. Полное имя атрибута (аргумент *qname* методов) в обоих случаях является строкой. Помимо общих для всех отображений операций и основных методов (copy(), get(), has_key(), items(), keys() и values()) они имеют следующие методы:

getLength()
Возвращает число атрибутов.

getNames()
Возвращает список имен атрибутов (ключей отображения).

getType(*name*)
Возвращает тип атрибута с именем *name* (обычно 'CDATA').

getValue(*name*)
Возвращает значение атрибута с именем *name*.

getValueByQName(*qname*)
Возвращает значения атрибута по его полному имени *qname*. Для экземпляров класса AttributesImpl этот метод эквивалентен методу getValue().

getNameByQName(*qname*)
Возвращает имя атрибута (строку или кортеж) по его полному имени *qname*. Для экземпляров класса AttributesImpl этот метод возвращает свой аргумент без изменений или генерирует исключение KeyError.

getQNameByName(*name*)
Возвращает полное имя атрибута с именем *name*. Для экземпляров класса AttributesImpl этот метод возвращает свой аргумент без изменений или генерирует исключение KeyError.

getQNames()
Возвращает список полных имен атрибутов. Для экземпляров класса AttributesImpl этот метод эквивалентен методу getNames().

29.9 xmllib — обработка XML-документов

Модуль xmllib считается устаревшим — используйте вместо него модуль xml.sax. Этот модуль определяет класс XMLParser, который может служить в качестве базового для класса, реализующего обработку размеченного текста в формате XML[3] (Extensible Markup Language, расширяемый язык разметки).

XMLParser([***keyword_args*])
Создает и возвращает объект, реализующий обработку текста в формате XML. В настоящий момент воспринимаются следующие именованные аргументы (*keyword_args*):

accept_unquoted_attributes
Если этот аргумент является истиной, обработчик будет воспринимать значения атрибутов, не заключенных в кавычки. По умолчанию равен 0.

accept_missing_endtag_name
Если этот аргумент является истиной, обработчик будет воспринимать закрывающие безымянные теги (‘</>’). По умолчанию равен 0.

[3] http://www.w3.org/TR/REC-xml, http://www.w3.org/XML/

map_case
: Если этот аргумент является истиной, обработчик будет приводить имена тегов и атрибутов к нижнему регистру. По умолчанию равен 0.

accept_utf8
: Если этот аргумент является истиной, обработчик допускает использование входной кодировки UTF-8. По умолчанию равен 0.

translate_attribute_references
: Если этот аргумент является истиной, обработчик будет заменять ссылки на символы и сущности в значениях атрибутов. По умолчанию равен 1.

Класс XMLParser имеет следующие интерфейсные атрибуты данных и методы, которые определяют тип документа (DTD):

attributes

Отображение, в котором в качестве ключей выступают имена элементов и в качестве значений — отображения имен возможных для данного элемента атрибутов к их значениям по умолчанию (или None, если атрибут не имеет значения по умолчанию). Вам следует переопределить этот атрибут в производном классе — в исходной реализации он равен пустому словарю.

elements

Отображение имен элементов к кортежам из функций (или других объектов, поддерживающих вызов), предназначенных для обработки открывающего и закрывающего тега для этого элемента или None, если должен быть вызван метод unknown_starttag() или unknown_endtag(). Вам следует переопределить этот атрибут в производном классе — в исходной реализации он равен пустому словарю.

entitydefs

Отображение имен сущностей к их значениям. В исходной реализации равен словарю, в котором есть записи с ключами 'lt', 'gt', 'amp', 'quot' и 'apos' (обязательный набор для всех типов документов).

reset()

Возвращает экземпляр в исходное состояние. Все необработанные данные теряются. Этот метод неявно вызывается при инициализации.

setnomoretags()

Останавливает обработку тегов. Весь последующий текст воспринимается без обработки (CDATA).

setliteral()

Временно приостанавливает обработку тегов. Последующий текст воспринимается без обработки (CDATA). Обработчик автоматически выходит из этого режима, как только дойдет до закрывающего тега для последнего открытого.

feed(*data*)

Передает экземпляру размеченный текст *data* для обработки. Обрабатываются

только завершенные данные, остальное остается в буфере до тех пор, пока не будет передана следующая порция данных или не будет вызван метод `close()`.

close()
Завершает обработку всех данных в буфере. Вы можете переопределить этот метод в производном классе, добавив дополнительные действия. Переопределенный метод всегда должен вызывать исходную версию метода.

translate_references(*data*)
Преобразует все ссылки на символы (‘&#*ref*;’) и сущности (‘&*ref*;’) в строке *data* и возвращает полученный текст.

getnamespace()
Возвращает отображение аббревиатур для (активных в настоящий момент) пространств имен к их URI (Uniform Resource Identifier, универсальный идентификатор ресурса).

handle_xml(*encoding*, *standalone*)
Вызывается для обработки тега ‘<?xml ...?>’. В качестве аргументов используются значения одноименных атрибутов тега (по умолчанию используются `None` и строка `'no'` соответственно). Исходная реализация ничего не делает.

handle_doctype(*tag*, *pubid*, *syslit*, *data*)
Вызывается для обработки объявления типа документа. Аргумент *tag* является строкой с именем корневого элемента, *pubid* — формальный публичный идентификатор (или `None`, если не задан), *syslit* — системный идентификатор и *data* — необработанный текст встроенного в документ DTD (Document Type Definition, определение типа документа) или `None`, если документ не содержит DTD. Исходная реализация ничего не делает.

handle_starttag(*tag*, *method*, *attributes*)
Вызывается для обработки открывающих тегов, для которых задан обработчик в атрибуте `elements`. Аргумент *tag* является строкой с именем тега, *method* — функция (метод), предназначенная для обработки открывающего тега, и *attributes* — словарь, отображающий имена атрибутов, найденных внутри угловых скобок тега, к их значениям (после обработки ссылок на символы и сущности). Например, для обработки открывающего тега ‘<A HREF="http://www.cwi.nl/">’ вызывается ‘*obj*.handle_starttag('A', *obj*.elements['A'][0], {'HREF': 'http://www.cwi.nl/'})’. Исходная реализация метода `handle_starttag` вызывает *method*(*attributes*).

handle_endtag(*tag*, *method*)
Вызывается для обработки закрывающих тегов, для которых задан обработчик в атрибуте `elements`. Аргумент *tag* является строкой с именем тега, *method* — функция (метод), предназначенная для обработки закрывающего тега. Например, для обработки закрывающего тега ‘</A>’ вызывается ‘*obj*.handle_endtag('A', *obj*.elements['A'][1])’. Исходная реализация метода `handle_endtag` просто вызывает *method*().

handle_data(*data*)
: Вызывается для обработки простого текста. Вам следует переопределить его в производном классе: исходная реализация ничего не делает.

handle_charref(*ref*)
: Вызывается для обработки ссылок на символы в виде '&#*ref*;'. Исходная реализация преобразует число в диапазоне от 0 до 255 в символ с помощью встроенной функции chr() и вызывает с ним метод handle_data(). Если аргумент *ref* неверен или выходит за пределы указанного диапазона, вызывает unknown_charref(*ref*) для обработки ошибки.

handle_comment(*comment*)
: Вызывается для обработки комментария. В качестве аргумента *comment* используется строка с текстом, содержащимся между '<!--' и '-->'. Исходная реализация ничего не делает.

handle_cdata(*data*)
: Вызывается для обработки элементов CDATA (текста между '<![CDATA[' и ']]>'). Вам следует переопределить его в производном классе: исходная реализация ничего не делает.

handle_proc(*name*, *data*)
: Вызывается для инструкций обработки (PI, Processing Instruction) в виде '<?*name* *data*?>'. Обратите внимание, что '<?xml *data*?>' обрабатывается отдельно — с помощью метода handle_xml(). Вам следует переопределить метод handle_proc() в производном классе: исходная реализация ничего не делает.

handle_special(*data*)
: Вызывается для обработки объявлений в виде '<!*data*>'. Обратите внимание, что '<!DOCTYPE ...>' в начале документа обрабатывается отдельно — с помощью метода handle_doctype(). Вам следует переопределить метод handle_special() в производном классе: исходная реализация ничего не делает.

syntax_error(*message*)
: Этот метод вызывается, если в документе обнаружена синтаксическая ошибка, при которой возможна дальнейшая обработка (при обнаружении фатальных ошибок этот метод не вызывается — сразу генерируется исключение RuntimeError). В качестве аргумента *message* используется строка с сообщением о том, что произошло. Исходная реализация этого метода генерирует исключение RuntimeError.

unknown_starttag(*tag*)
: Вызывается для обработки неизвестного открывающего тега. Вам следует переопределить этот метод в производном классе: исходная реализация ничего не делает.

unknown_endtag(*tag*)
: Вызывается для обработки неизвестного закрывающего тега. Вам следует переопределить этот метод в производном классе: исходная реализация ничего не делает.

unknown_charref(*ref*)

Вызывается для обработки неверной ссылки на символ в виде ‘&#*ref*;’. Вам следует переопределить этот метод в производном классе: исходная реализация ничего не делает.

unknown_entityref(*ref*)

Вызывается для обработки ссылки на сущность с неизвестным именем в виде ‘&*ref*;’. Вам следует переопределить этот метод в производном классе: исходная реализация вызывает метод `syntax_error()` для обработки ошибки.

Модуль `xmllib` поддерживает пространства имен XML. Полные имена тегов и атрибутов (то, что используется в качестве аргументов методов), определенных в пространстве имен, состоит URI, определяющего пространство имен, и самого имени тега или атрибута, разделенных пробелом. Например, запись ‘`<html xmlns='http://www.w3.org/TR/REC-html40'>`’ воспринимается как тег с именем `'http://www.w3.org/TR/REC-html40 html'` и запись ‘`<html:a href='http://frob.com'>`’ внутри этого элемента воспринимается как тег с именем `'http://www.w3.org/TR/REC-html40 a'` и атрибутом `'http://www.w3.org/TR/REC-html40 href'`.

Глава 30

Разное

fileinput	Перебор строк из нескольких входных потоков.
ConfigParser	Чтение конфигурационных файлов.
shlex	простой синтаксический анализатор для командных языков.
cmd	Создание построчных командных интерпретаторов.
calendar	Функции для работы с календарем, включая эмуляцию программы **cal**.

30.1 `fileinput` — перебор строк из нескольких входных потоков

Модуль `fileinput` определяет класс и функции, помогающие перебирать строки, полученные со стандартного потока и/или файлов, указанных списков. Чаще всего этот модуль используется примерно следующим образом:

```
import fileinput
for line in fileinput.input():
    process(line)
```

В этом случае перебираются строки файлов, имена которых указаны в качестве аргументов в командной строке (`sys.argv[1:]`), или стандартного потока ввода, если этот список пуст. Имя файла `'-'` считается ссылающимся на стандартный поток ввода `sys.stdin`. Вы можете указать альтернативный список имен файлов (или имя одного файла) в качестве первого аргумента функции `input()`.

Все файлы открываются в текстовом режиме, поэтому модуль может быть использован только для чтения файлов с текстом в однобайтовой кодировке.

input([*files* [, *inplace* [, *backup*]]])
Создает и возвращает экземпляр класса `FileInput`, который также сохраняется и используется другими функций этого модуля. Все аргументы передаются конструктору.

Следующие функции используют экземпляр класса `FileInput`, созданный и сохраненный функцией `input()`. При попытке использования этих функций без предварительного вызова `input()` или после вызова функции `close()` генерируется исключение `RuntimeError`. Текущим всегда считается файл, из которого была прочитана последняя строка.

filename()
: Возвращает имя файла, для которого была прочитана последняя строка. До того, как прочитана первая строка (в том числе непосредственно после вызова функции `nextfile()`), возвращает `None`. Для стандартного потока ввода возвращает `'<stdin>'`.

lineno()
: Возвращает суммарный номер последней прочитанной строки. До того, как прочитана первая строка (в том числе непосредственно после вызова функции `nextfile()`), возвращает `0`.

filelineno()
: Возвращает номер последней прочитанной строки в текущем файле. До того, как прочитана первая строка (в том числе непосредственно после вызова функции `nextfile()`), возвращает `0`.

isfirstline()
: Возвращает `1`, если последняя прочитанная строка является первой в текущем файле, иначе возвращает `0`.

isstdin()
: Возвращает `1`, если последняя строка была прочитана со стандартного потока ввода (`sys.stdin`), иначе возвращает `0`.

nextfile()
: Закрывает текущий файл и переходит к следующему. Строки, не прочитанные из этого файла, не будут учитываться при вычислении суммарного номера строки. До прочтения первой строки эта функция не дает никакого эффекта.

close()
: Вызывает метод `close()` экземпляра, сохраненного функцией `input()`.

Следующий класс реализует чтение из нескольких файлов и может быть использован для определения производных классов:

FileInput([*files* [, *inplace* [, *backup*]]])
: Создает и возвращает объект, обеспечивающий чтение строк из нескольких файлов. Аргумент *files* должен быть списком имен файлов или строкой с именем файла. По умолчанию используется `sys.argv[1:]` или стандартный поток ввода, если список аргументов командной строки пуст. Имя `'-'` является специальным, для него используется поток `sys.stdin`.

Если задан и является истиной аргумент *inplace*, файл переименовывается в резервную копию с именем, полученным добавлением суффикса *backup* (по умолчанию используется '.bak') к исходному имени файла, стандартный поток вывода перенаправляется в исходный файл, и перебираются строки резервной копии. Таким образом, одновременно с чтением строк Вы можете безопасно изменять исходный файл, выводя в него информацию, например, инструкцией print. Резервная копия удаляется при закрытии выходного файла. Такое редактирование файла (“in-place filtering”) невозможно при чтении из стандартного потока ввода.

Текущая реализация не позволяет работать с файловой системой MS-DOS, которая налагает сильные ограничения на имена файлов.

Экземпляры класса FileInput ведут себя аналогично последовательностям, но предоставляют доступ к элементам в определенном порядке (этого достаточно для использования в инструкции for), а также имеют следующие методы:

filename()
Возвращает имя файла, для которого была прочитана последняя строка. До того, как прочитана первая строка (в том числе непосредственно после вызова метода nextfile()), возвращает None. Для стандартного потока ввода возвращает '<stdin>'.

lineno()
Возвращает суммарный номер последней прочитанной строки. До того, как прочитана первая строка (в том числе непосредственно после вызова метода nextfile()), возвращает 0.

filelineno()
Возвращает номер последней прочитанной строки в текущем файле. До того, как прочитана первая строка (в том числе непосредственно после вызова метода nextfile()), возвращает 0.

isfirstline()
Возвращает 1, если последняя прочитанная строка является первой в текущем файле, иначе возвращает 0.

isstdin()
Возвращает 1, если последняя строка была прочитана со стандартного потока ввода (sys.stdin), иначе возвращает 0.

nextfile()
Закрывает текущий файл и переходит к следующему. Строки, не прочитанные из этого файла, не будут учитываться при вычислении суммарного номера строки. До прочтения первой строки эта функция не дает никакого эффекта.

close()
“Закрывает” объект.

readline()
Считывает и возвращает следующую строку. Этот метод увеличивает счетчик строк

на единицу, поэтому вызов *obj*.readline() внутри цикла 'for *line* in *obj*' приведет к ошибке (исключение RuntimeError).

30.2 ConfigParser — чтение конфигурационных файлов

Этот модуль определяет класс ConfigParser, реализующий чтение и обработку простых конфигурационных файлов, аналогичных по структуре '.ini'-файлам в Windows.

Конфигурационный файл состоит из разделов, начинающихся с заголовка вида '[*раздел*]' и содержащий строки вида '*имя*: *значение*'. Длинные значения могут быть записаны в несколько строк в стиле RFC 822. Также допускается использование строк вида '*имя*=*значение*'. Обратите внимание, что начальные символы пропуска из значения удаляются. Значения могут содержать строки формата, ссылающиеся на другие значения в этом же разделе или разделе DEFAULT (раздел значений по умолчанию):

```
foodir: %(dir)s/whatever
dir=frob
```

В приведенном примере последовательность '%(dir)s' будет заменена значением 'dir' (здесь оно равно 'frob'). Дополнительные значения по умолчанию могут быть указаны при инициализации и извлечении. Строки, начинающиеся с символа '#' или ';' считаются комментариями и игнорируются.

ConfigParser([*defaults*])
Возвращает объект, отвечающий за чтение конфигурационного файла. Отображение *defaults* определяет значения по умолчанию: ключи в нем должны быть строками, определяющие имена, значения могут содержать последовательности вида '%(*имя*)s'. Заметим, что имя __name__ всегда имеет значение, равное имени раздела.

MAX_INTERPOLATION_DEPTH
Определяет максимальную глубину рекурсии при раскрытии специальных последовательностей в значениях опций. Эту переменную следует считать доступной только для чтения: ее изменение не окажет никакого влияния на работу модуля.

Модуль также определяет следующую иерархию исключений:

Error
Базовый класс для всех исключений, которые генерируются при ошибках, характерных для данного модуля.

NoSectionError
Указанный раздел (атрибут section экземпляра исключения) не найден.

DuplicateSectionError
Раздел с таким именем (атрибут section экземпляра исключения) уже существует.

NoOptionError
Опция с указанным именем (атрибут option экземпляра исключения) не найдена в указанном разделе (атрибут section).

InterpolationError
Проблемы при раскрытии специальной последовательности вида '%(*имя*)s' (атрибут reference экземпляра исключения, имена раздела и опции доступны через атрибуты section и option).

InterpolationDepthError
Достигнута максимальная глубина рекурсии (MAX_INTERPOLATION_DEPTH) при раскрытии специальных последовательностей в значении опции (имена раздела и опции доступны через атрибуты section и option экземпляра исключения).

MissingSectionHeaderError
Конфигурационный файл не содержит заголовков. Имя файла, номер строки и сама строка, в которой обнаружена ошибка, доступны через атрибуты filename, lineno и line экземпляра исключения.

ParsingError
Ошибка при анализе конфигурационного файла. Атрибут filename экземпляра исключения содержит имя конфигурационного файла и атрибут errors — список пар из номера и содержимого строки для всех строк, в которых обнаружены ошибки.

Экземпляры класса ConfigParser имеют следующие методы:

defaults()
Возвращает отображение со значениями по умолчанию.

sections()
Возвращает список имен доступных разделов. Имя DEFAULT в список не включается.

add_section(*section*)
Добавляет раздел с именем *section*. Если раздел с таким именем уже существует, генерируется исключение DuplicateSectionError.

has_section(*section*)
Возвращает 1, если есть раздел с именем *section*, иначе возвращает 0. Имя раздела 'DEFAULT' не распознается.

options(*section*)
Возвращает имен доступных опций в разделе *section*.

has_option(*section*, *option*)
Возвращает 1, если существует раздел с именем *section* и содержит опцию с именем *option*, иначе возвращает 0. Этот метод доступен, начиная с версии 1.6.

read(*filenames*)
Считывает и обрабатывает конфигурационные файлы. Аргумент *filenames* может быть списком имен файлов или строкой, содержащей имя одного файла.

readfp(*fp* [, *filename*])
Считывает и обрабатывает конфигурационный файл, представленный файловым (или подобным) объектом *fp* (должен иметь метод `readline()`). Аргумент *filename* используется в качестве имени файла в сообщениях об ошибках. Если аргумент *filename* не задан, используется *fp*.name или '<???>', если объект *fp* не имеет атрибута name.

get(*section*, *option* [, *raw* [, *vars*]])
Возвращает значение опции *option* в разделе *section*. Если аргумент *raw* опущен или является ложью, метод `get()` раскрывает специальные последовательности, начинающиеся с символа '%'. При этом принимаются во внимание значения по умолчанию, переданные конструктору и этому методу в качестве аргумента *vars*. Значения, указанные в отображении *vars* имеют преимущество над значениями по умолчанию, переданными конструктору.

getint(*section*, *option*)
Возвращает значение опции *option* в разделе *section* в виде целого числа.

getfloat(*section*, *option*)
Возвращает значение опции *option* в разделе *section* в виде вещественного числа.

getboolean(*section*, *option*)
Возвращает значение опции *option* в разделе *section* в виде целого числа 0 или 1. В конфигурационном файле значение опции должно быть указано строкой '0' или '1', во всех остальных случаях генерируется исключение `ValueError`.

set(*section*, *option*, *value*)
Если существует раздел с именем *section*, устанавливает в нем значение опции *option* равным *value*. Если аргумент *section* является ложью или равен строке 'DEFAULT', *value* запоминается как значение опции *option* по умолчанию. В остальных случаях генерируется исключение `NoSectionError`. Метод `set()` доступен, начиная с версии 1.6.

write(*fp*)
Записывает текущее состояние объекта в конфигурационный файл, представленный файловым (или подобным) объектом *fp*. Этот метод доступен, начиная с версии 1.6.

remove_option(*section*, *option*)
Удаляет опцию *option* из раздела *section*. Если аргумент *section* является ложью или равен строке 'DEFAULT', удаляет значение опции *option* по умолчанию.

В остальных случаях, если раздел *section* отсутствует, генерируется исключение NoSectionError. Если опция присутствовала, возвращает 1, иначе возвращает 0. Метод remove_option() доступен, начиная с версии 1.6.

remove_section(*section*)
Полностью удаляет раздел *section*. Если раздел присутствовал, возвращает 1, иначе возвращает 0.

30.3 shlex — простой синтаксический анализатор

Класс shlex, определенный в этом модуле, значительно упрощает написание синтаксических анализаторов для командных языков с простым синтаксисом, подобных UNIX shell. Такая возможность часто полезна для написания мини-языков, используемых, например, для написания конфигурационных файлов (см. также описание модуля ConfigParser).

shlex([*stream* [, *file*]])
Инициализирует и возвращает объект, представляющий лексический анализатор. Аргумент *stream* должен быть файловым объектом (или подобным, с методами read() и readline()) и определяет поток ввода анализатора, по умолчанию используется sys.stdin. Если задан аргумент *file*, он определяет имя файла (используется для инициализации атрибута infile).

Экземпляры класса shlex имеют следующие методы:

get_token()
Возвращает очередную лексему. Лексема может быть считана из стека (если она была записана туда с помощью метода push_token()) или из входного потока. При достижении конца файла возвращает пустую строку.

push_token(*str*)
Записывает строку *str* в качестве лексемы в стек.

read_token()
Считывает из входного потока (игнорируя стек) и возвращает очередную лексему. Этот метод также игнорирует запросы на включения файла.

sourcehook(*filename*)
Этот метод вызывается для обработки запросов на включение файла. Аргумент *filename* является строкой, следующей за лексемой включения файла (атрибут source). Метод sourcehook() должен возвращать кортеж, содержащий имя файла и файловый (или подобный) объект.

error_leader([*file* [, *line*]])

Возвращает начало сообщения об ошибке в формате '"*file*", line *line*: '. Если аргументы *file* и/или *line* опущены, используются текущие имя файла и номер строки. Такой формат сообщения об ошибке является общепринятым и воспринимается редактором Emacs и другими инструментами UNIX.

Экземпляры класса shlex также имеют следующие атрибуты, управляющие процессом лексического анализа:

commenters

Строка символов, которые воспринимаются как начало комментария. Все символы от начала комментария до конца строки игнорируются. По умолчанию включает только символ '#'.

wordchars

Строка символов, последовательности из которых воспринимаются как одна лексема. По умолчанию включает цифры, символ подчеркивания и буквы латинского алфавита.

whitespace

Строка из символов, которые считаются разделителями лексем (символы пропуска). По умолчанию этот атрибут равен ' \t\r\n'.

quotes

Строка из символов, которые воспринимаются как кавычки. Лексема, начинающаяся с одного из символов кавычки, всегда заканчивается точно таким же символом (таким образом, строка в кавычках одного типа может содержать кавычки другого типа). По умолчанию включает одиночную и двойную кавычку.

infile

Имя текущего входного файла (или None, если имя файла не установлено). Полезен при конструировании сообщений об ошибке.

instream

Файловый объект, представляющий текущий входной поток.

source

По умолчанию этот атрибут равен None. Если ему присвоить строку, то эта строка будет восприниматься как лексема запроса на включение файла. Следующая после source лексема будет восприниматься как имя файла, содержимое которого должно быть обработано лексическим анализатором. После обработки включенного файла лексический анализатор продолжает обработку текущего потока.

debug

Целое число, определяющее, насколько подробной должна быть выводимая отладочная информация. Если атрибут debug равен 0, отладочная информация не выводится.

lineno

Номер текущей строки (число прочитанных символов перехода на новую строку плюс 1).

token

Буфер лексем. Может быть полезен при обработке исключений.

Обратите внимание, что символы, не входящие в строки wordchars, whitespace и quotes, воспринимаются как односимвольные лексемы. Кроме того, символы кавычек внутри слов теряют свое значение, а комментарии маскируют также и символ перехода на новую строку. Так, ввод ‘ain't’ и ain#d\n't будут восприняты как одна и та же лексема — строка "ain't".

30.4 cmd — создание командных интерпретаторов

Этот модуль определяет класс Cmd, который может служить основой при написании командных интерпретаторов, ориентированных на строки. Хорошим примером использования этого метода может послужить отладчик языка Python (модуль pdb).

Cmd()

Этот класс реализует основу интерпретатора. Производные от него классы должны предоставить методы, реализующие действия при вводе различных команд.

Экземпляры класса Cmd имеют следующие методы и атрибуты данных:

cmdloop([*intro*])

Выполняет в цикле следующие операции: выводит приглашение, воспринимает ввод, обрабатывает начало введенной команды и вызывает для ее выполнения соответствующий метод, передавая в качестве аргумента остаток строки. Аргумент *intro* (строка) определяет выводимый перед первым приглашением заголовок, по умолчанию используется атрибут intro.

Интерпретатор будет воспринимать команду ‘foo’ только, если объект имеет метод do_foo(). Символы ‘?’ и ‘!’ в начале строки воспринимаются как команды ‘help’ и ‘shell’ соответственно, конец файла — как команда ‘EOF’.

Предопределенный метод do_help с аргументом 'bar' вызывает метод help_bar(). Без аргументов do_help() выводит список всех доступных тем, для которых может быть получена помощь (определен соответствующий метод help_*()), и список недокументированных команд. Класс Cmd не определяет методы do_shell() и do_EOF().

Большинство методов do_*() должно возвращать None. Если возвращаемое значение является истиной (и метод postcmd() не переопределен), работа интерпретатора будет завершена.

onecmd(*str*)

Воспринимает строку *str*, как если бы она была введена в ответ на приглашение интерпретатора.

emptyline()
: Этот метод вызывается, если в ответ на приглашение введена пустая строка. По умолчанию повторяет последнюю введенную непустую команду.

default(*line*)
: Этот метод вызывается, если первая лексема команды не распознана (не имеет соответствующего метода `do_*()`). По умолчанию выводит сообщение об ошибке.

precmd(*line*)
: Этот метод вызывается для предварительной обработки введенной строки (аргумент *line*). По умолчанию возвращает строку без изменений.

postcmd(*stop*, *line*)
: Этот метод вызывается после обработки и выполнения команды. В качестве аргументов *stop* и *line* используются значение, возвращаемое соответствующим методом `do_*()`, и строка ввода, обработанная методом `precmd()`. Если значение, возвращаемое этим методом, является истиной, работа интерпретатора будет завершена. По умолчанию возвращает свой аргумент *stop* без изменений.

preloop()
: Этот метод вызывается один раз при запуске интерпретатора (вызове метода `cmdloop()`) перед выводом заголовка. По умолчанию ничего не делает.

postloop()
: Этот метод вызывается один раз при завершении работы интерпретатора. По умолчанию ничего не делает.

prompt
: Строка, выводимая в качестве приглашения. По умолчанию используется `'(Cmd) '`.

identchars
: Строка из символов, последовательность из которых воспринимается как один идентификатор. По умолчанию содержит цифры, символ подчеркивания и буквы[1].

lastcmd
: Строка, содержащая последнюю непустую команду (первую лексему введенной строки).

intro
: Строка, которая используется в качестве заголовка, выводимого при запуске интерпретатора. Этот атрибут может быть переопределен, если при вызове метода `cmdloop()` использован аргумент, отличный от `None`.

doc_header
: Заголовок к списку документированных команд (для которых есть соответствующие методы `do_*()` и `help_*()`).

[1]Текущая реализация использует данные в соответствии с национальными установками на момент инициализации модуля.

misc_header
Заголовок к списку доступных тем помощи, не являющихся командами (то есть, для которых есть метод `help_*()`, но нет соответствующего метода `do_*()`).

undoc_header
Заголовок к списку недокументированных команд (для которых есть соответствующий метод `do_*()`, но нет метода `help_*()`).

ruler
Символ, который используется для линий под заголовком помощи. Если является пустой строкой, линия не будет использована. По умолчанию атрибут `ruler` равен `'='`.

30.5 `calendar` — функции для работы с календарем

Этот модуль предоставляет функции для работы с календарем, а также позволяет выводить календарь аналогично программе **cal**. По умолчанию считается, что неделя начинается с понедельника. Чтобы установить иной день, например воскресенье, используйте функцию `setfirstweekday()`.

MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY
Константы со значениями от 0 (понедельник) до 6 (воскресенье), представляющие дни недели.

setfirstweekday(*weekday*)
Устанавливает использование *weekday* (целое число от 0 до 6) в качестве первого дня недели. Например, следующий фрагмент устанавливает использование в качестве первого дня недели воскресенья:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

firstweekday()
Возвращает целое число от 0 до 6, определяющее день недели, с которого (согласно текущим установкам) неделя начинается.

isleap(*year*)
Возвращает `1`, если год *year* (целое число) является високосным, иначе возвращает `0`.

leapdays(*y1*, *y2*)

Возвращает число високосных лет в промежутке с *y1* по *y2* - 1 год (включительно).

weekday(*year*, *month*, *day*)

Возвращает целое число, определяющее день недели для указанной даты. *year* — год (см. замечания в описании модуля time), *month* — месяц (от 1 до 12) и *day* — число (от 1 до 31). Например, следующий код позволяет вывести номера месяцев 2000 года, в которых 13 число приходится на пятницу:

```
>>> from calendar import weekday, FRIDAY
>>> for month in xrange(1, 13):
...     if weekday(2000, month, 13)==FRIDAY:
...         print month
...
10
```

monthrange(*year*, *month*)

Возвращает кортеж, содержащий первый день недели месяца (целое число от 0 до 6) и число дней в месяце для указанного года (аргумент *year*) и месяца (аргумент *month*).

monthcalendar(*year*, *month*)

Возвращает матрицу (список списков), представляющую календарь на месяц для указанного года (аргумент *year*) и месяца (аргумент *month*). Каждый элемент возвращаемого списка представляет собой список из семи элементов — чисел для каждого дня недели. Дни, выходящие за пределы месяца, представлены нулями. Неделя начинается с понедельника, если иной день не был установлен с помощью функции setfirstweekday().

month(*theyear*, *themonth* [, *datewidth* [, *linesperweek*]])

Возвращает строковое представление календаря на месяц для указанного года (аргумент *theyear*) и месяца (аргумент *themonth*). Аргументы *datewidth* и *linesperweek* определяют ширину поля даты и число строк, используемых для одной недели.

prmonth(*theyear*, *themonth* [, *datewidth* [, *linesperweek*]])

Выводит на стандартный поток вывода (sys.stdout) календарь на месяц. Аргументы имеют такое же значение, как и для функции month().

calendar(*year* [, *datewidth* [, *linesperweek* [, *spaces*]]])

Возвращает строковое представление в три колонки календаря на год *year*. Аргументы *datewidth*, *linesperweek* и *spaces* определяют ширину поля даты, число строк, используемых для одной недели, и количество пробелов между колонками с календарями на месяц.

prcal()

Выводит на стандартный поток вывода (sys.stdout) календарь на год. Аргументы имеют такое же значение, как и для функции calendar().

timegm(*time_tuple*)

Возвращает число секунд, прошедших с начала эпохи (независимо от системы начало эпохи считается равным началу 1970 года) до времени, которое представлено кортежем из девяти чисел *time_tuple* (см. описание модуля time).

Приложения

Приложение A

Параметры командной строки интерпретатора и переменные окружения

Для вызова интерпретатора Python должна быть набрана командная строка следующего формата:

python [*options*] [-c *command* | *file* | -] [*arguments*]

Где *python* — путь к исполняемому файлу интерпретатора (или просто имя, если каталог, в котором он находится, включен в пути поиска), *options* — опции, *command* — команда на языке Python, *file* — файл с программой на языке Python и *arguments* — список аргументов, передаваемых в `sys.argv[1:]`.

Интерпретатор поддерживает следующие опции:

`-d`

Выводить отладочную информацию при синтаксическом анализе. Этот режим может быть также включен с помощью переменной окружения PYTHONDEBUG (см. ниже).

`-i`

После выполнения программы перейти в интерактивный режим с выводом приглашения, даже если стандартный поток ввода интерпретатора не является терминалом. Этот режим может быть также включен с помощью переменной окружения PYTHONINSPECT (см. ниже).

`-O`

Выполнять оптимизацию генерируемого байт-кода. Оптимизация может быть также включена с помощью переменной окружения PYTHONOPTIMIZE (см. ниже).

`-OO`

Помимо оптимизации (опция `-O`) удалить из байт-кода строки документации.

`-S`

Не импортировать модуль `site` при инициализации.

`-t`

Выводить предупреждения о непоследовательном использовании символов табуляции в исходном коде.

`-tt`

Генерировать исключение `TabError` при обнаружении непоследовательного использования символов табуляции в исходном коде.

`-u`

Отключить буферизацию для стандартных потоков вывода и ошибок. Этот режим может быть также включен с помощью переменной окружения PYTHONUNBUFFERED (см. ниже). Отключение буферизации может быть необходимо для CGI-программ.

`-U`

Считать все строковые литеральные выражения строками Unicode, то есть воспринимать `'...'` как `u'...'`. Импортируемые модули, байт-код которых был получен без использования этой опции, будут откомпилированы заново. Возможность задания этой опции присутствует, начиная с версии 1.6.

`-v`

Выводить отладочную информацию при импортировании модулей. Этот режим может быть также включен с помощью переменной окружения PYTHONVERBOSE (см. ниже).

`-x`

Игнорировать первую строку исходного кода. Такой режим позволяет задавать способ выполнения программы в его первой строке на платформах, отличных от UNIX.

`-X`

Использовать строки вместо классов для стандартных исключений. Начиная с версии 1.6, эта опция не поддерживается.

`-h`

Вывести подсказку и завершить выполнение. Возможность задания этой опции присутствует, начиная с версии 2.0.

`-V`

Вывести номер версии интерпретатора и завершить выполнение. Возможность задания этой опции присутствует, начиная с версии 2.0.

Кроме того, на работу интерпретатора оказывают влияние значения следующих переменных окружения:

PYTHONDEBUG

Если эта переменная имеет непустое значение, интерпретатор будет выводить отладочную информацию при синтаксическом анализе исходного кода.

PYTHONINSPECT

Если эта переменная имеет непустое значение, интерпретатор после выполнения программы перейдет в интерактивный режим с выводом приглашения, даже если стандартный поток ввода интерпретатора не является терминалом.

PYTHONOPTIMIZE
Если эта переменная имеет непустое значение, интерпретатор будет оптимизировать генерируемый байт-код.

PYTHONUNBUFFERED
Если эта переменная имеет непустое значение, буферизация стандартных потоков вывода и ошибок будет отключена. Отключение буферизации может быть необходимо для CGI-программ.

PYTHONVERBOSE
Если эта переменная имеет непустое значение, интерпретатор будет выводить отладочную информацию при импортировании модулей.

PYTHONSTARTUP
Путь к файлу, который будет выполнен при запуске интерпретатора в интерактивном режиме.

PYTHONPATH
Список каталогов, разделенных символом '`;`', в которых будет производиться поиск модулей перед поиском в путях по умолчанию. Значение этой переменной окружения отражается на значении `sys.path` (см. описание модуля `sys`).

PYTHONHOME
Устанавливает альтернативное значение `sys.prefix` и, возможно, `sys.exec_prefix`. Значение переменной окружения должно иметь вид '*prefix*[`;`*exec_prefix*]' (см. описание модуля `sys`).

Приложение B

Грамматика языка

Здесь используются следующие обозначения для лексем:

`NAME`
: Имя (идентификатор).

`NUMBER`
: Числовое литеральное выражение.

`STRING`
: Строковое литеральное выражение.

`NEWLINE`
: Символ (или последовательность из двух символов), обозначающий переход на новую строку.

`ENDMARKER`
: Конец файла (потока).

`INDENT`
: Увеличение уровня отступа.

`DEDENT`
: Уменьшение уровня отступа.

Начальные символы грамматики:

`single_input`
: Одна интерактивная инструкция.

`file_input`
: Модуль или последовательность команд, считываемая из файла.

`eval_input`
: Инструкция-выражение, передаваемое функции `eval()` или ввод, получаемый функцией `input()`.

```
single_input:  NEWLINE |
               simple_stmt |
               compound_stmt NEWLINE
```

```
file_input:     (NEWLINE | stmt)* ENDMARKER

eval_input:     testlist NEWLINE* ENDMARKER


funcdef:        'def' NAME parameters ':' suite

parameters:     '(' [varargslist] ')'

varargslist:    (fpdef ['=' test] ',')*
                ('*' NAME [',' '**' NAME] | '**' NAME) |
                fpdef ['=' test] (',' fpdef ['=' test])*
                [',']

fpdef:          NAME | '(' fplist ')'

fplist:         fpdef (',' fpdef)* [',']


stmt:            simple_stmt | compound_stmt

simple_stmt:     small_stmt (';' small_stmt)* [';']
                 NEWLINE

small_stmt:      expr_stmt | print_stmt  | del_stmt |
                 pass_stmt | flow_stmt | import_stmt |
                 global_stmt | exec_stmt | assert_stmt

expr_stmt:       testlist (augassign testlist |
                 ('=' testlist)*)

augassign:       '+=' | '-=' | '*=' | '/=' | '%=' | '&='
                 | '|=' | '^=' | '<<=' | '>>=' | '**='

print_stmt:      'print' ( [ test (',' test)* [','] ] |
                 '>>' test [ (',' test)+ [','] ] )

del_stmt:        'del' exprlist

pass_stmt:       'pass'

flow_stmt:       break_stmt | continue_stmt |
                 return_stmt | raise_stmt

break_stmt:      'break'
```

```
continue_stmt:  'continue'

return_stmt:    'return' [testlist]

raise_stmt:     'raise' [test [',' test [',' test]]]

import_stmt:    'import' dotted_as_name
                (',' dotted_as_name)* |
                'from' dotted_name 'import'
                ('*' | import_as_name
                (',' import_as_name)*)

import_as_name: NAME [NAME NAME]

dotted_as_name: dotted_name [NAME NAME]

dotted_name:    NAME ('.' NAME)*

global_stmt:    'global' NAME (',' NAME)*

exec_stmt:      'exec' expr ['in' test [',' test]]

assert_stmt:    'assert' test [',' test]


compound_stmt:  if_stmt | while_stmt | for_stmt |
                try_stmt | funcdef | classdef

if_stmt:        'if' test ':' suite
                ('elif' test ':' suite)*
                ['else' ':' suite]

while_stmt:     'while' test ':' suite
                ['else' ':' suite]

for_stmt:       'for' exprlist 'in' testlist ':' suite
                ['else' ':' suite]

try_stmt:       ('try' ':' suite
                (except_clause ':' suite)+
                ['else' ':' suite] |
                'try' ':' suite
                'finally' ':' suite)

except_clause:  'except' [test [',' test]]
```

```
suite:          simple_stmt |
                NEWLINE INDENT stmt+ DEDENT



test:           and_test ('or' and_test)* | lambdef

and_test:       not_test ('and' not_test)*

not_test:       'not' not_test | comparison

comparison:     expr (comp_op expr)*

comp_op:        '<' | '>' | '==' | '>=' | '<=' | '<>'
                | '!=' | 'in' | 'not' 'in'| 'is'
                |'is' 'not'

expr:           xor_expr ('|' xor_expr)*

xor_expr:       and_expr ('^' and_expr)*

and_expr:       shift_expr ('&' shift_expr)*

shift_expr:     arith_expr (('<<' | '>>') arith_expr)*

arith_expr:     term (('+' | '-') term)*

term:           factor (('*' | '/' | '%') factor)*

factor:         ('+' | '-' | '~') factor | power

power:          atom trailer* ('**' factor)*

atom:           '(' [testlist] ')' |
                '[' [listmaker] ']' |
                '{' [dictmaker] '}' |
                '`' testlist '`' |
                NAME | NUMBER | STRING+

listmaker:      test ( list_for | (',' test)* [','] )

lambdef:        'lambda' [varargslist] ':' test

trailer:        '(' [arglist] ')' |
                '[' subscriptlist ']' | '.' NAME

subscriptlist:  subscript (',' subscript)* [',']

subscript:      '.' '.' '.' | test |
```

```
                [test] ':' [test] [sliceop]

sliceop:        ':' [test]

exprlist:       expr (',' expr)* [',']

testlist:       test (',' test)* [',']

dictmaker:      test ':' test (',' test ':' test)*
                [',']



classdef:       'class' NAME ['(' testlist ')'] ':'
                suite

arglist:        (argument ',')* (argument [',']| '*'
                test [',' '**' test] | '**' test)

argument:       [test '='] test



list_iter:      list_for | list_if

list_for:       'for' exprlist 'in' testlist
                [list_iter]

list_if:        'if' test [list_iter]
```

Указатель модулей

A

anydbm ... 327
array ... 252
atexit ... 200

B

Bastion ... 354
bisect ... 251
bsddb ... 330

C

calendar ... 419
cgi ... 355
cmath ... 247
cmd ... 417
codecs ... 232
ConfigParser ... 412
cPickle ... 258
cStringIO ... 231

D

dbhash ... 327
dbm ... 327
dumbdbm ... 327

E

errno ... 299

F

filecmp ... 286
fileinput ... 409
fnmatch ... 303

G

gc ... 198
gdbm ... 327
getopt ... 296
getpass ... 295
gettext ... 240
glob ... 302
gzip ... 336

H

htmlentitydefs ... 388
htmllib ... 386

I

imp ... 211

L

locale ... 236

M

marshal ... 263
math ... 245
mimetools ... 373
MimeWriter ... 374
mmap ... 315
multifile ... 376

O

operator ... 204
os ... 267
os.path ... 280

P

pdb ... 340
pickle ... 258
popen2 ... 288
pprint ... 212
profile ... 347
pstats ... 348

Q

Queue ... 325

R

random ... 249
re ... 220
repr ... 215
rexec ... 351
rfc822 ... 369

S

sched ... 294
select ... 313
sgmllib ... 383
shelve ... 262
shlex ... 415
shutil ... 304
signal ... 305
site ... 190
socket ... 307
stat ... 283
statvfs ... 285
string ... 217
StringIO ... 231

struct 264
sys 192

T

tempfile 298
thread 318
threading 320
time 289
traceback 208
types 201

U

urllib 363
urlparse 367
user 191
UserDict 257
UserList 256
UserString 255

W

whichdb 330
whrandom 250

X

xdrlib 378
xml.parsers.expat 388
xml.sax 392
xml.sax.handler 394
xml.sax.saxutils 398
xml.sax.xmlreader 399
xmllib 404

Z

zipfile 336
zlib 333

Предметный указатель

(), оператор 46, 107, 148, 149, 156
*
  оператор ...27, 108, 133, 135, 161, 204, 205
  передача позиционных аргументов ...43, 44, 46, 107
  в инструкции import 60, 66, 118
**
  оператор 108, 134, 161
  передача именованных аргументов ..44, 46, 108
**=, присваивание 114
*=, присваивание 114, 162
+
  бинарный оператор ..27, 108, 133, 135, 161, 204, 205
  унарный оператор 108, 134, 160, 204
+=, присваивание 114, 162
-
  бинарный оператор 108, 133, 161, 204
  унарный оператор 108, 134, 160, 204
-=, присваивание 114, 162
., оператор 83, 85, 86, 106, 151, 155
... .166
.pdbrc, файл 342
.pythonrc.py, файл 22, 191
/, оператор 108, 134, 161, 204
/=, присваивание 114, 162
<, оператор 109
<=, оператор 109
<>, оператор 109
<<, оператор 108, 134, 161, 205
<<=, присваивание 114, 162
=, присваивание 24, 113
==, оператор 109
>, оператор 109
>=, оператор 109
>>, оператор 108, 134, 161
>>=, присваивание 114, 162
[], оператор 28, 107, 135, 145–147, 157, 166, 206
%, оператор ..68, 108, 134, 137, 159, 161, 204
%=, присваивание 114
&, оператор 108, 134, 161, 205
&=, присваивание 114, 162
^, оператор 108, 134, 161, 205
^=, присваивание 114, 162
|, оператор 108, 134, 161, 205
|=, присваивание 114, 162
~, оператор 108, 134, 160, 205
_, встроенная переменная 25
__abs__()
  спец. метод 160
  в модуле operator 205
__add__()
  спец. метод 161
  в модуле operator 204
__all__ 65, 66
__and__()
  спец. метод 161
  в модуле operator 205
__bases__, атрибут объектов class 152
__builtins__ 129
__call__(), спец. метод 127, 151, 156
__class__, атрибут объектов instance 153
__cmp__(), спец. метод 109, 154
__coerce__(), спец. метод 161
__complex__(), спец. метод 160
__concat__(), в модуле operator 205
__contains__()
  спец. метод 159
  в модуле operator 205
__debug__ 61, 120
__del__(), спец. метод 153, 199
__delattr__(), спец. метод 115, 156
__delattr__()(), спец. метод 153
__delitem__()
  спец. метод 115, 157
  в модуле operator 206
__delslice__()
  спец. метод 115, 158
  в модуле operator 206
__dict__
  атрибут объектов class 149, 152, 350
  атрибут объектов instance ..91, 153, 156, 350
  атрибут объектов module 151
__div__()
  спец. метод 161
  в модуле operator 204
__divmod__(), спец. метод 161
__doc__
  атрибут объектов builtin_function_or_method ...150
  атрибут объектов class 152

атрибут объектов function 149
атрибут объектов instance method ..150
атрибут объектов module 151
атрибут объектов type 165
__file__, атрибут объектов module 151
__float__(), спец. метод 160
__getattr__(), спец. метод ..98, 153, 155, 159
__getinitargs__(), протокол копирования 259
__getitem__()
спец. метод 157
в модуле operator 206
__getslice__()
спец. метод 157
в модуле operator 206
__getstate__(), протокол копирования 259
__hash__(), спец. метод 155, 256
__hex__(), спец. метод 159
__iadd__(), спец. метод 162
__iand__(), спец. метод 162
__idiv__(), спец. метод 162
__ilshift__(), спец. метод 162
__imod__(), спец. метод 162
__import__(), встроенная функция 119, 170
__imul__(), спец. метод 162
__init__(), спец. метод ..86, 128, 151, 153
__init__.py, файл 64, 66
__int__(), спец. метод 160
__inv__(), в модуле operator 205
__invert__()
спец. метод 160
в модуле operator 205
__ior__(), спец. метод 162
__ipow__(), спец. метод 162
__irshift__(), спец. метод 162
__isub__(), спец. метод 162
__ixor__(), спец. метод 162
__len__(), спец. метод 157, 159, 175
__long__(), спец. метод 160
__lshift__()
спец. метод 161
в модуле operator 205
__members__, атрибут объектов встроенных типов 131
__methods__, атрибут объектов встроенных типов 131
__mod__()
спец. метод 161
в модуле operator 204
__module__, атрибут объектов class ..152
__mul__()
спец. метод 161
в модуле operator 204
__name__ 58
атрибут объектов builtin_function_or_method ...150
атрибут объектов class 152, 182
атрибут объектов function 149
атрибут объектов instance method ..150
атрибут объектов module 151
атрибут объектов type 165
__neg__()
спец. метод 160
в модуле operator 204
__nonzero__(), спец. метод 155
__not__(), в модуле operator 205
__oct__(), спец. метод 159
__or__()
спец. метод 161
в модуле operator 205
__pos__()
спец. метод 160
в модуле operator 204
__pow__(), спец. метод 161
__radd__(), спец. метод 162
__rand__(), спец. метод 162
__rcmp__(), спец. метод 109, 154
__rdiv__(), спец. метод 162
__rdivmod__(), спец. метод 162
__repeat__(), в модуле operator 205
__repr__(), спец. метод 159
__rlshift__(), спец. метод 162
__rmod__(), спец. метод 162
__rmul__(), спец. метод 162
__ror__(), спец. метод 162
__rpow__(), спец. метод 162
__rrshift__(), спец. метод 162
__rshift__(), спец. метод 161
__rsub__(), спец. метод 162
__rxor__(), спец. метод 162
__self__, атрибут объектов builtin_function_or_method ...151
__setattr__(), спец. метод ..98, 113, 153, 156
__setitem__()
спец. метод 113, 157
в модуле operator 206

__setslice__()
спец. метод 113, 158
в модуле operator 206
__setstate__(), протокол копирования 259
__stderr__, в модуле sys 197
__stdin__, в модуле sys 197
__stdout__, в модуле sys 197
__str__(), спец. метод 159
__sub__()
спец. метод 161
в модуле operator 204
__xor__()
спец. метод 161
в модуле operator 205
_exit(), в модуле os 186, 277
in, в инструкции for, *см.* for
__builtins__ 350
``, строковое представление 68, 159, 179

A

abort(), в модуле os 276
abs()
в модуле operator 205
встроенная функция 25, 160, 171, 205
abspath(), в модуле os.path 280
accept(), в модуле socket 310
accept2dyear, в модуле time 291
access(), в модуле os 273
acos()
в модуле cmath 247
в модуле math 245
acosh(), в модуле cmath 247
activeCount(), в модуле threading .. 320
add(), в модуле operator 204
AddressList(), в модуле rfc822 370
adler32(), в модуле zlib 334
AF_INET, в модуле socket 308
AF_UNIX, в модуле socket 308
alarm(), в модуле signal 306
all_features, в модуле
xml.sax.handler 395
all_properties, в модуле
xml.sax.handler 395
allocate_lock(), в модуле thread ... 318
altsep, в модуле os 280
altzone(), в модуле time 291
and, оператор 56, 111
and_(), в модуле operator 205
append(), метод объектов list . 41, 47, 48, 146
apply(), встроенная функция 46, 171
aRepr, в модуле repr 215
argv, в модуле sys 192
ArithmeticError, исключение 183
array, в модуле array 145
array(), в модуле array 252
ArrayType, в модуле array 253
as, в инструкции import 118
asctime(), в модуле time 291
asin()
в модуле cmath 247
в модуле math 245
asinh(), в модуле cmath 247
assert, инструкция 61, 120, 183
AssertionError, исключение 120, 183
atan()
в модуле cmath 247
в модуле math 245
atan2(), в модуле math 245
atanh(), в модуле cmath 248
atof()
в модуле locale 239
в модуле string 218
atoi()
в модуле locale 239
в модуле string 218
atol(), в модуле string 218
AttributeError, исключение 107, 183
AttributesImpl(), в модуле
xml.sax.xmlreader 399
AttributesNSImpl(), в модуле
xml.sax.xmlreader 400

B

basename(), в модуле os.path 280
Bastion(), в модуле Bastion 354
BastionClass(), в модуле Bastion ... 354
betavariate(), в модуле random 249
big-endian 193, 235
bind(), в модуле socket 310
bindtextdomain(), в модуле gettext 241
bisect(), в модуле bisect 251
BOM, в модуле codecs 235
BOM32_BE, в модуле codecs 235
BOM32_LE, в модуле codecs 235
BOM64_BE, в модуле codecs 235
BOM64_LE, в модуле codecs 235
BOM_BE, в модуле codecs 235
BOM_LE, в модуле codecs 235
break, инструкция 39, 116
btopen(), в модуле bsddb 331
buffer, встроенный тип 135, 145, 202

buffer(), встроенная функция 145, 171
BufferType, в модуле types 202
builtin_function_or_method,
встроенный тип 150, 203
builtin_module_names, в модуле sys 193
BuiltinFunctionType, в модуле types
203
BuiltinMethodType, в модуле types ..203
byteorder, в модуле sys 192

C

C_BUILTIN, в модуле imp 212
C_EXTENSION, в модуле imp 212
calcsize(), в модуле struct 264
calendar(), в модуле calendar 420
callable(), встроенная функция ..148, 171
cancel(), в модуле sched 295
capitalize()
метод объектов string и unicode 142
в модуле string 218
capwords(), в модуле string 218
Catalog(), в модуле gettext 242
ceil(), в модуле math 246
center()
метод объектов string и unicode 140
в модуле string 220
CGI 355
CHAR_MAX, в модуле locale 238
charmap_decode(), в модуле codecs ..234
charmap_encode(), в модуле codecs ..234
chdir(), в модуле os 273
chmod(), в модуле os 273
choice()
в модуле random 250
в модуле whrandom 251
choose_boundary(), в модуле mimetools
373
chown(), в модуле os 273
chr(), встроенная функция 171
class
инструкция 85, 128
встроенный тип 85, 151, 152, 203
ClassType, в модуле types 203
clear(), метод объектов dictionary ..148
clock(), в модуле time 291
close()
метод объектов file 163
в модуле fileinput 410
в модуле os 270
в модуле socket 310
closed, атрибут объектов file 164
Cmd(), в модуле cmd 417
cmp()
в модуле filecmp 286
встроенная функция 147, 154, 171, 239
cmpfiles(), в модуле filecmp 286
co_argcount, атрибут объектов code ..167
co_code, атрибут объектов code 167
co_consts, атрибут объектов code 167
co_filename, атрибут объектов code ..167
co_firstlineno, атрибут объектов code
167
co_flags, атрибут объектов code 167
co_lnotab, атрибут объектов code 167
co_name, атрибут объектов code 167
co_names, атрибут объектов code 167
co_nlocals, атрибут объектов code 167
co_stacksize, атрибут объектов code .167
co_varnames, атрибут объектов code ..167
code, встроенный тип 166, 203, 263
Codec(), в модуле codecs 234
CodeType, в модуле types 203
coerce(), встроенная функция 171
collect(), в модуле gc 199
commonprefix(), в модуле os.path ...280
compatible_formats, в модулях pickle и
cPickle 261
compile()
в модуле re 225
встроенная функция 117, 167, 172, 173
complex, встроенный тип 24, 131, 202
complex(), встроенная функция 24, 132,
160, 172
ComplexType, в модуле types 202
compress(), в модуле zlib 334
compressobj(), в модуле zlib 334
concat(), в модуле operator 205
Condition(), в модуле threading 320
ConfigParser(), в модуле ConfigParser
412
confstr(), в модуле os 279
confstr_names, в модуле os 279
conjugate(), метод объектов complex 133
connect(), в модуле socket 310
connect_ex(), в модуле socket 310
contains(), в модуле operator 205
ContentHandler()
в модуле xml.sax.handler 394
в модуле xml.sax 393
continue, инструкция 39, 116
ConversionError, в модуле xdrlib ...382

copy()
метод объектов dictionary 148
в модуле shutil 304
copy2(), в модуле shutil 304
copybinary(), в модуле mimetools ...374
copyfile(), в модуле shutil 304
copyfileobj(), в модуле shutil 304
copyliteral(), в модуле mimetools ..373
copymode(), в модуле shutil 304
copyright, в модуле sys 193
copystat(), в модуле shutil 304
copytree(), в модуле shutil 304
cos()
в модуле cmath 248
в модуле math 246
cosh()
в модуле cmath 248
в модуле math 246
count()
метод объектов list 47, 146
метод объектов string и unicode 140
в модуле string 219
countOf(), в модуле operator 206
crc32(), в модуле zlib 334
ctermid(), в модуле os 268
ctime(), в модуле time 291
cunifvariate(), в модуле random 249
curdir, в модуле os 280
currentThread(), в модуле threading 320

D

daylight, в модуле time 292
DEBUG_COLLECTABLE, в модуле gc 200
DEBUG_INSTANCES, в модуле gc 200
DEBUG_LEAK, в модуле gc 200
DEBUG_OBJECTS, в модуле gc 200
DEBUG_SAVEALL, в модуле gc 200
DEBUG_STATS, в модуле gc 200
DEBUG_UNCOLLECTABLE, в модуле gc ...200
decode(), в модуле mimetools 373
decompress(), в модуле zlib 335
decompressobj(), в модуле zlib 335
def, инструкция 40, 125, 149
DEF_MEM_LEVEL, в модуле zlib 334
defpath, в модуле os 280
del, инструкция 52, 55, 115, 145, 147
delattr(), встроенная функция 172
delitem(), в модуле operator 206
delslice(), в модуле operator 206
dgettext(), в модуле gettext 241
dictionary, встроенный тип ... 55, 95, 147, 155, 203
DictionaryType, в модуле types 203
DictType, в модуле types 203
digits, в модуле string 217
dir(), встроенная функция 63, 172
dircmp(), в модуле filecmp 286
dirname(), в модуле os.path 281
disable(), в модуле gc 198
div(), в модуле operator 204
divmod(), встроенная функция 161, 172
dllhandle, в модуле sys 193
DOTALL, в модуле re 226
DST 290
DTD 384, 394, 405, 406
DTDHandler(), в модуле xml.sax.handler 394
dump()
в модуле marshal 263
в модулях pickle и cPickle 261
dumps()
в модуле marshal 263
в модулях pickle и cPickle 261
dup(), в модуле os 271
dup2(), в модуле os 271
DuplicateSectionError, в модуле ConfigParser 413

E

e
в модуле cmath 248
в модуле math 247
E2BIG, в модуле errno 300
EACCES, в модуле errno 300
EAGAIN, в модуле errno 300
EBADF, в модуле errno 300
EBADMSG, в модуле errno 302
EBUSY, в модуле errno 300
ECHILD, в модуле errno 300
EDEADLK, в модуле errno 301
EDOM, в модуле errno 301
EEXIST, в модуле errno 300
EFAULT, в модуле errno 300
EFBIG, в модуле errno 301
EILSEQ, в модуле errno 302
EINPROGRESS, в модуле errno 302
EINTR, в модуле errno 300
EINVAL, в модуле errno 301
EIO, в модуле errno 300
EISDIR, в модуле errno 301
elif, ветвь инструкции if 36, 122
Ellipsis, объект 166

ellipsis, встроенный тип 166, 203
EllipsisType, в модуле types 203
else
ветвь инструкции if 36, 122
ветвь инструкции try 78, 124
ветвь в циклах while и for .. 39, 116, 122, 123
EMFILE, в модуле errno 301
EMLINK, в модуле errno 301
Empty, в модуле Queue 325
empty(), в модуле sched 295
EMSGSIZE, в модуле errno 302
enable(), в модуле gc 198
ENAMETOOLONG, в модуле errno 301
encode()
метод объектов string и unicode 142
в модуле mimetools 373
EncodedFile(), в модуле codecs 233
endswith(), метод объектов string и unicode 141
ENFILE, в модуле errno 301
ENODEV, в модуле errno 300
ENOENT, в модуле errno 300
ENOEXEC, в модуле errno 300
ENOLCK, в модуле errno 302
ENOMEM, в модуле errno 300
ENOSPC, в модуле errno 301
ENOSYS, в модуле errno 302
ENOTDIR, в модуле errno 301
ENOTEMPTY, в модуле errno 302
ENOTTY, в модуле errno 301
enter(), в модуле sched 294
enterabs(), в модуле sched 294
entitydefs, в модуле htmlentitydefs 388
EntityResolver(), в модуле xml.sax.handler 394
enumerate(), в модуле threading 320
environ, в модуле os 267
EnvironmentError, исключение 184
ENXIO, в модуле errno 300
EOFError, исключение 178, 183
EPERM, в модуле errno 300
EPIPE, в модуле errno 301
ERANGE, в модуле errno 301
EROFS, в модуле errno 301
Error
в модуле ConfigParser 412
в модуле locale 236
в модуле multifile 376
в модуле xdrlib 382
error
в модуле bsddb 331
в модуле getopt 296
в модуле os 267
в модуле re 228
в модуле select 313
в модуле socket 308
в модуле struct 264
в модуле thread 318
в модуле xml.parsers.expat 389
в модуле zipfile 337
в модуле zlib 334
в модулях anydbm, dumbdbm, dbhash, dbm и gdbm 328
errorcode, в модуле errno 299
ErrorHandler()
в модуле xml.sax.handler 394
в модуле xml.sax 393
errors, в модуле xml.parsers.expat 389
ErrorString(), в модуле xml.parsers.expat 388
escape()
в модуле cgi 361
в модуле re 228
в модуле xml.sax.saxutils 398
ESPIPE, в модуле errno 301
ESRCH, в модуле errno 300
ETIMEDOUT, в модуле errno 302
ETXTBSY, в модуле errno 301
eval(), встроенная функция .. 117, 119, 130, 167, 172
Event(), в модуле threading 320
exc_info, в модуле sys 124, 193
exc_info(), в модуле sys 169
exc_traceback, в модуле sys 125, 193
exc_type, в модуле sys 125, 193
exc_value, в модуле sys 125, 193
except, ветвь инструкции try ... 21, 77, 93, 124, 182
Exception, исключение 93, 182
EXDEV, в модуле errno 300
exec, инструкция 117, 119, 130, 167
exec_prefix, в модуле sys 193
execfile(), встроенная функция . 117, 130, 173
execl(), в модуле os 276
execle(), в модуле os 276
execlp(), в модуле os 276
executable, в модуле sys 194
execv(), в модуле os 276
execve(), в модуле os 276
execvp(), в модуле os 276
execvpe(), в модуле os 277

exists(), в модуле os.path 281
exit()
  в модуле sys 194
  в модуле thread 318
exit_thread(), в модуле thread 318
exitfunc, в модуле sys 194
exp()
  в модуле cmath 248
  в модуле math 246
expandtabs()
  метод объектов string и unicode 141
  в модуле string 219
expanduser(), в модуле os.path 281
expandvars(), в модуле os.path 281
expovariate(), в модуле random 249
extend(), метод объектов list 146
extract_stack(), в модуле traceback 209
extract_tb(), в модуле traceback ...209

F

f_back, атрибут объектов frame 168
F_BAVAIL, в модуле statvfs 285
F_BFREE, в модуле statvfs 285
F_BLOCKS, в модуле statvfs 285
F_BSIZE, в модуле statvfs 285
f_builtins, атрибут объектов frame ..168
f_code, атрибут объектов frame 168
f_exc_traceback, атрибут объектов frame 169
f_exc_type, атрибут объектов frame ..169
f_exc_value, атрибут объектов frame .169
F_FAVAIL, в модуле statvfs 286
F_FFREE, в модуле statvfs 286
F_FILES, в модуле statvfs 285
F_FLAG, в модуле statvfs 286
F_FRSIZE, в модуле statvfs 285
f_globals, атрибут объектов frame 168
f_lasti, атрибут объектов frame 168
f_lineno, атрибут объектов frame 168
f_locals, атрибут объектов frame 168
F_NAMEMAX, в модуле statvfs 286
F_OK, в модуле os 273
f_restricted, атрибут объектов frame 168
f_trace, атрибут объектов frame 169
fabs(), в модуле math 246
FancyURLopener(), в модуле urllib ..366
fdopen(), в модуле os 269
feature_*, в модуле xml.sax.handler 394
FieldStorage(), в модуле cgi 356
FIFO 48, 325
file, встроенный тип 72, 163, 203, 311
FileInput(), в модуле fileinput 410
filelineno(), в модуле fileinput ...410
filename(), в модуле fileinput 410
fileno()
  метод объектов file 163
  в модуле socket 311
FileType, в модуле types 203
filter(), встроенная функция 49, 173
finally, ветвь инструкции try 80, 124, 186
find()
  метод объектов string и unicode 141
  в модуле gettext 241
  в модуле string 219
find_module(), в модуле imp 211
findall(), в модуле re 227
firstweekday(), в модуле calendar ..419
float, встроенный тип 24, 131, 132, 202
float(), встроенная функция . 25, 132, 160, 173
FloatingPointError, исключение 183
FloatType, в модуле types 202
floor(), в модуле math 246
flush(), метод объектов file 163
fmod(), в модуле math 246
fnmatch(), в модуле fnmatch 303
fnmatchcase(), в модуле fnmatch 303
for
  инструкция 36, 39, 116, 123
  в конструкторе списков 51
fork(), в модуле os 277
forkpty(), в модуле os 277
format(), в модуле locale 239
format_exception(), в модуле traceback 209
format_exception_only(), в модуле traceback 209
format_list(), в модуле traceback ..209
format_stack(), в модуле traceback 210
format_tb(), в модуле traceback 210
format_version, в модулях pickle и cPickle 261
formatdate(), в модуле rfc822 370
fpathconf(), в модуле os 271
frame, встроенный тип 168, 204
FrameType, в модуле types 204
frexp(), в модуле math 246
FRIDAY, в модуле calendar 419
from, в инструкции import 59, 65, 118

fromfd(), в модуле socket 309
fstat(), в модуле os 271
fstatvfs(), в модуле os 271
ftruncate(), в модуле os 271
Full, в модуле Queue 326
func_code, атрибут объектов function 149, 167
func_defaults, атрибут объектов function 149
func_doc, атрибут объектов function . 149
func_globals, атрибут объектов function 149, 350
func_name, атрибут объектов function 149
function, встроенный тип 125, 149, 152, 203
FunctionType, в модуле types 203

G

gamma(), в модуле random 249
garbage, в модуле gc 199
gauss(), в модуле random 249
get(), метод объектов dictionary 148
get_debug(), в модуле gc 199
get_ident(), в модуле thread 319
get_magic(), в модуле imp 211
get_suffixes(), в модуле imp 211
get_threshold(), в модуле gc 199
getatime(), в модуле os.path 281
getattr(), встроенная функция 173
getcwd(), в модуле os 273
getdefaultencoding(), в модуле sys 194
getdefaultlocale(), в модуле locale 238
getegid(), в модуле os 268
geteuid(), в модуле os 268
getfqdn(), в модуле socket 308
getgid(), в модуле os 268
getgroups(), в модуле os 268
gethostbyaddr(), в модуле socket ... 309
gethostbyname(), в модуле socket ... 308
gethostbyname_ex(), в модуле socket 308
gethostname(), в модуле socket 309
getitem(), в модуле operator 206
getlocale(), в модуле locale 238
getlogin(), в модуле os 268
getmtime(), в модуле os.path 281
getopt(), в модуле getopt 296
GetoptError, в модуле getopt 296
getpass(), в модуле getpass 296
getpeername(), в модуле socket 311
getpgrp(), в модуле os 268
getpid(), в модуле os 268
getppid(), в модуле os 268
getprotobyname(), в модуле socket .. 309
getrecursionlimit(), в модуле sys .. 194
getrefcount(), в модуле sys 194
getservbyname(), в модуле socket ... 309
getsignal(), в модуле signal 306
getsize(), в модуле os.path 281
getslice(), в модуле operator 206
getsockname(), в модуле socket 311
getsockopt(), в модуле socket 311
gettempprefix(), в модуле tempfile 299
gettext(), в модуле gettext 241
getuid(), в модуле os 268
getuser(), в модуле getpass 296
glob(), в модуле glob 302
global, инструкция .. 40, 105, 117, 119, 128, 185
globals(), встроенная функция ... 119, 174
GMT 290
gmtime(), в модуле time 292
GNUTranslations(), в модуле gettext 243
GzipFile(), в модуле gzip 336

H

has_key(), метод объектов dictionary 55, 148
hasattr(), встроенная функция 174
hash(), встроенная функция 155, 174
hashopen(), в модуле bsddb 331
hex(), встроенная функция 159, 174
hexdigits, в модуле string 217
hexversion, в модуле sys 194
HotProfile(), в модуле profile 347
HTML 386
HTMLParser(), в модуле htmllib 387
htonl(), в модуле socket 310
htons(), в модуле socket 310
hypot(), в модуле math 246

I

I, в модуле re 226
id(), встроенная функция 174
if
  инструкция 36, 110, 122
  в конструкторе списков 52
IGNORECASE, в модуле re 226
im_class, атрибут объектов instance method 150, 152
im_func, атрибут объектов instance method 150

im_self, атрибут объектов instance method ... 150
imag, атрибут объектов complex ... 133
import, инструкция ... 59, 65, 117, 170, 211
ImportError, исключение 65, 184, 211, 212
in, оператор ... 56, 110, 135, 159, 205
INADDR_*, в модуле socket ... 308
IncrementalParser(), в модуле xml.sax.xmlreader ... 399
index()
- метод объектов list ... 47, 146
- метод объектов string и unicode ... 141
- в модуле string ... 219

IndexError, исключение ... 29, 94, 123, 135, 185
indexOf(), в модуле operator ... 206
inet_aton(), в модуле socket ... 310
inet_ntoa(), в модуле socket ... 310
input()
- в модуле fileinput ... 409
- встроенная функция ... 129, 174, 197

InputSource()
- в модуле xml.sax.xmlreader ... 399
- в модуле xml.sax ... 393

InputType, в модуле cStringIO ... 232
insert(), метод объектов list ... 47, 146
insort(), в модуле bisect ... 251
install(), в модуле gettext ... 242
instance, встроенный тип 86, 151, 152, 203
instance method, встроенный тип . 87, 149
InstanceType, в модуле types ... 203
int, встроенный тип ... 23, 131, 132, 202
int(), встроенная функция 25, 132, 160, 174
intern(), встроенная функция ... 175
Internet ... 355
InterpolationDepthError, в модуле ConfigParser ... 413
InterpolationError, в модуле ConfigParser ... 413
IntType, в модуле types ... 202
inv(), в модуле operator ... 205
invert(), в модуле operator ... 205
IOError, исключение ... 163, 184
IP_*, в модуле socket ... 308
IPPORT_*, в модуле socket ... 308
IPPROTO_*, в модуле socket ... 308
is, оператор ... 56, 109
is not, оператор ... 56, 110
is_zipfile(), в модуле zipfile ... 337
isabs(), в модуле os.path ... 281
isatty()
- метод объектов file ... 163
- в модуле os ... 271

isCallable(), в модуле operator ... 206
isdecimal(), метод объектов unicode 144
isdigit(), метод объектов string и unicode ... 143
isdir(), в модуле os.path ... 281
isenabled(), в модуле gc ... 199
isfile(), в модуле os.path ... 281
isfirstline(), в модуле fileinput .. 410
isinstance(), встроенная функция ... 175
isleap(), в модуле calendar ... 419
islink(), в модуле os.path ... 281
islower(), метод объектов string и unicode ... 143
isMappingType(), в модуле operator 206
ismount(), в модуле os.path ... 282
isNumberType(), в модуле operator .. 207
isnumeric(), метод объектов unicode 144
isreadable(), в модуле pprint ... 214
isrecursive(), в модуле pprint ... 214
isSequenceType(), в модуле operator 207
isspace(), метод объектов string и unicode ... 143
isstdin(), в модуле fileinput ... 410
issubclass(), встроенная функция ... 175
istitle(), метод объектов string и unicode ... 143
isupper(), метод объектов string и unicode ... 143
items(), метод объектов dictionary .. 148

J

join()
- метод объектов string и unicode ... 142
- в модуле os.path ... 282
- в модуле string ... 219

joinfields(), в модуле string ... 219

K

KeyboardInterrupt, исключение ... 21, 77, 184
KeyError, исключение ... 55, 147, 157, 185
keys(), метод объектов dictionary 55, 148
kill(), в модуле os ... 277

L

L, в модуле re ... 226
lambda, оператор ... 45, 111, 149
LambdaType, в модуле types ... 203
LANG, переменная окружения ... 240, 242
LANGUAGE, переменная окружения .. 240, 242

last_traceback, в модуле sys ... 169, 195
last_type, в модуле sys ... 195
last_value, в модуле sys ... 195
LC_ALL
  переменная окружения ... 240, 242
  в модуле locale ... 240
LC_COLLATE, в модуле locale ... 239
LC_CTYPE, в модуле locale ... 239
LC_MESSAGES
  переменная окружения ... 240, 242
  в модуле locale ... 240
LC_MONETARY, в модуле locale ... 240
LC_NUMERIC, в модуле locale ... 240
LC_TIME, в модуле locale ... 239
ldexp(), в модуле math ... 246
leapdays(), в модуле calendar ... 420
len(), встроенная функция ... 30, 33, 38, 135, 147, 157, 175
letters, в модуле string ... 217
LIFO ... 48
lineno(), в модуле fileinput ... 410
linesep, в модуле os ... 280
link(), в модуле os ... 273
list, встроенный тип ... 32, 47, 48, 51, 135, 145, 202
list(), встроенная функция ... 175
listdir(), в модуле os ... 274
listen(), в модуле socket ... 311
ListType, в модуле types ... 202
little-endian ... 193, 235
ljust()
  метод объектов string и unicode ... 140
  в модуле string ... 220
load()
  в модуле marshal ... 263
  в модулях pickle и cPickle ... 261
load_module(), в модуле imp ... 211
loads()
  в модуле marshal ... 263
  в модулях pickle и cPickle ... 261
LOCALE, в модуле re ... 226
localeconv(), в модуле locale ... 237
locals(), встроенная функция ... 119, 175
localtime(), в модуле time ... 292
Locator(), в модуле xml.sax.xmlreader 399
Lock(), в модуле threading ... 320
LockType, в модуле thread ... 318
log()
  в модуле cgi ... 361
  в модуле cmath ... 248
  в модуле math ... 246
log10()
  в модуле cmath ... 248
  в модуле math ... 246
logfile, в модуле cgi ... 361
logfp, в модуле cgi ... 361
lognormvariate(), в модуле random ... 249
long int, встроенный тип ... 131, 132, 202
long(), встроенная функция ... 25, 132, 160, 176
LongType, в модуле types ... 202
lookup(), в модуле codecs ... 233
LookupError, исключение ... 184
lower()
  метод объектов string и unicode ... 143
  в модуле string ... 219
lowercase, в модуле string ... 217
lseek(), в модуле os ... 271
lshift(), в модуле operator ... 205
lstat(), в модуле os ... 274
lstrip()
  метод объектов string и unicode ... 141
  в модуле string ... 219
lvalue ... 113

M

M, в модуле re ... 226
make_parser(), в модуле xml.sax ... 392
makedirs(), в модуле os ... 274
makefile(), в модуле socket ... 311
maketrans(), в модуле string ... 218
map(), встроенная функция ... 49, 176, 207
MAP_PRIVATE, в модуле mmap ... 316
MAP_SHARED, в модуле mmap ... 315
match(), в модуле re ... 227
max(), встроенная функция ... 176
MAX_INTERPOLATION_DEPTH, в модуле ConfigParser ... 412
maxint, в модуле sys ... 195
MemoryError, исключение ... 185
Message()
  в модуле mimetools ... 373
  в модуле rfc822 ... 369
method, встроенный тип ... 203
MethodType, в модуле types ... 203
MIME ... 373, 374
MimeWriter(), в модуле MimeWriter ... 375
min(), встроенная функция ... 176
MiniFieldStorage, в модуле cgi ... 357
MissingSectionHeaderError, в модуле ConfigParser ... 413
mkdir(), в модуле os ... 274

mkfifo(), в модуле os ... 274
mktemp(), в модуле tempfile ... 298
mktime(), в модуле time ... 292
mktime_tz(), в модуле rfc822 ... 370
mmap(), в модуле mmap ... 315
mod(), в модуле operator ... 204
mode, атрибут объектов file ... 164
modf(), в модуле math ... 246
module, встроенный тип ... 151, 203
modules, в модуле sys ... 118, 195
ModuleType, в модуле types ... 203
MONDAY, в модуле calendar ... 419
month(), в модуле calendar ... 420
monthcalendar(), в модуле calendar 420
monthrange(), в модуле calendar ... 420
MSG_*, в модуле socket ... 308
mul(), в модуле operator ... 204
MultiFile(), в модуле multifile ... 376
MULTILINE, в модуле re ... 226
MutableString(), в модуле UserString 256

N

name
  атрибут объектов file ... 165
  в модуле os ... 267
NameError, исключение ... 53, 77, 105, 123, 129, 185
neg(), в модуле operator ... 204
new_module(), в модуле imp ... 212
nextfile(), в модуле fileinput ... 410
nice(), в модуле os ... 277
None
  объект ... 41, 43, 110, 165
  встроенный тип ... 165, 202
NoneType, в модуле types ... 202
NoOptionError, в модуле ConfigParser 413
normalize(), в модуле locale ... 238
normalvariate(), в модуле random ... 249
normcase(), в модуле os.path ... 282
normpath(), в модуле os.path ... 282
NoSectionError, в модуле ConfigParser 412
not, оператор ... 56, 111, 205
not in, оператор ... 56, 110, 135, 159
not_(), в модуле operator ... 205
NotImplementedError, исключение ... 185
NSIG, в модуле signal ... 306
ntohl(), в модуле socket ... 309
ntohs(), в модуле socket ... 309
NullTranslations(), в модуле gettext 242

O

O_APPEND, в модуле os ... 272
O_BINARY, в модуле os ... 273
O_CREAT, в модуле os ... 272
O_DSYNC, в модуле os ... 272
O_EXCL, в модуле os ... 272
O_NDELAY, в модуле os ... 272
O_NOCTTY, в модуле os ... 273
O_NONBLOCK, в модуле os ... 272
O_RDONLY, в модуле os ... 272
O_RDWR, в модуле os ... 272
O_RSYNC, в модуле os ... 272
O_SYNC, в модуле os ... 272
O_TEXT, в модуле os ... 273
O_TRUNC, в модуле os ... 272
O_WRONLY, в модуле os ... 272
oct(), встроенная функция ... 159, 176
octdigits, в модуле string ... 217
open()
  в модуле codecs ... 233
  в модуле gzip ... 336
  в модуле os ... 271
  в модуле shelve ... 262
  в модуле urllib ... 366
  в модулях anydbm, dumbdbm, dbhash, dbm и gdbm ... 328
  встроенная функция ... 72, 163, 176
open_unknown(), в модуле urllib ... 366
openpty(), в модуле os ... 271
or, оператор ... 56, 111
or_(), в модуле operator ... 205
ord(), встроенная функция ... 177
OSError, исключение ... 184, 267
OutputType, в модуле cStringIO ... 232
OverflowError, исключение ... 174, 183, 292

P

P_DETACH, в модуле os ... 277
P_NOWAIT, в модуле os ... 277
P_NOWAITO, в модуле os ... 277
P_OVERLAY, в модуле os ... 277
P_WAIT, в модуле os ... 277
pack(), в модуле struct ... 264
Packer(), в модуле xdrlib ... 379
pardir, в модуле os ... 280
paretovariate(), в модуле random ... 249
parse()

в модуле cgi 359
в модуле xml.sax 392
parse_header(), в модуле cgi 360
parse_multipart(), в модуле cgi 360
parse_qs(), в модуле cgi 360
parse_qsl(), в модуле cgi 360
parsedate(), в модуле rfc822 370
parsedate_tz(), в модуле rfc822 370
ParserCreate(), в модуле xml.parsers.expat 388
parseString(), в модуле xml.sax 393
ParsingError, в модуле ConfigParser 413
pass, инструкция 39, 115
path
в модуле os 267
в модуле sys 119, 195
pathconf(), в модуле os 274
pathconf_names, в модуле os 274
pathsep, в модуле os 280
pause(), в модуле signal 306
Pdb(), в модуле pdb 340
pformat(), в модуле pprint 214
PI 390, 407
pi
в модуле cmath 248
в модуле math 247
PickleError, в модулях pickle и cPickle 261
Pickler(), в модулях pickle и cPickle 260
PicklingError, в модулях pickle и cPickle 262
pipe(), в модуле os 272
PKG_DIRECTORY, в модуле imp 212
platform, в модуле sys 196
plock(), в модуле os 277
pm(), в модуле pdb 342
poll(), в модуле select 314
POLLERR, в модуле select 315
POLLHUP, в модуле select 315
POLLIN, в модуле select 314
POLLNVAL, в модуле select 315
POLLOUT, в модуле select 315
POLLPRI, в модуле select 314
pop(), метод объектов list 48, 146
popen(), в модуле os 270
popen2()
в модуле os 270
в модуле popen2 288
Popen3(), в модуле popen2 288
popen3()
в модуле os 270
в модуле popen2 288
Popen4(), в модуле popen2 289
popen4()
в модуле os 270
в модуле popen2 288
pos(), в модуле operator 204
post_mortem(), в модуле pdb 342
pow()
в модуле math 246
встроенная функция 161, 177
pprint(), в модуле pprint 214
prcal(), в модуле calendar 420
prefix, в модуле sys 196
prepare_input_source(), в модуле xml.sax.saxutils 398
PrettyPrinter(), в модуле pprint ... 213
print, инструкция 27, 35, 68, 115, 159
print_directory(), в модуле cgi 360
print_environ(), в модуле cgi 360
print_environ_usage(), в модуле cgi 360
print_exc(), в модуле traceback 208
print_exception(), в модуле traceback 208
print_form(), в модуле cgi 360
print_last(), в модуле traceback ... 208
print_stack(), в модуле traceback .. 209
print_tb(), в модуле traceback 208
printable, в модуле string 218
prmonth(), в модуле calendar 420
Profile(), в модуле profile 347
property_*, в модуле xml.sax.handler 394
PROT_READ, в модуле mmap 316
PROT_WRITE, в модуле mmap 316
ps1, в модуле sys 196
ps2, в модуле sys 196
punctuation, в модуле string 217
putenv(), в модуле os 268
PY_COMPILED, в модуле imp 212
PY_FROZEN, в модуле imp 212
PY_RESOURCE, в модуле imp 212
PY_SOURCE, в модуле imp 212
PYTHONDEBUG, переменная окружения ... 426
PYTHONHOME, переменная окружения 427
PYTHONINSPECT, переменная окружения 426
PYTHONOPTIMIZE, переменная окружения 61, 427

PYTHONPATH, переменная окружения . 61, 62, 119, 196, 427
PYTHONSTARTUP, переменная окружения 21, 191, 427
PYTHONUNBUFFERED, переменная окружения 427
PYTHONVERBOSE, переменная окружения 427
PYTHONY2K, переменная окружения 290, 291
PyZipFile(), в модуле zipfile 337

Q

Queue(), в модуле Queue 325
quote(), в модуле urllib 364
quote_plus(), в модуле urllib 364

R

R_OK, в модуле os 273
raise, инструкция 92, 120
randint()
  в модуле random 250
  в модуле whrandom 250, 251
random()
  в модуле random 250
  в модуле whrandom 250, 251
randrange()
  в модуле random 250
  в модуле whrandom 250, 251
range(), встроенная функция 37, 177
raw_input(), встроенная функция 178, 197
read()
  метод объектов file 73, 163
  в модуле os 272
readline(), метод объектов file .. 73, 163
readlines(), метод объектов file . 73, 164
readlink(), в модуле os 274
real, атрибут объектов complex 133
recv(), в модуле socket 311
recvfrom(), в модуле socket 311
reduce(), встроенная функция . 51, 178, 207
register()
  в модуле codecs 233
  в модуле select 314
reload(), встроенная функция 179, 195, 211
remove()
  метод объектов list 47, 146
  в модуле os 274
removedirs(), в модуле os 274
rename(), в модуле os 275
renames(), в модуле os 275
repeat(), в модуле operator 205
replace()
  метод объектов string и unicode 141
  в модуле string 220
Repr(), в модуле repr 215
repr()
  в модуле repr 215
  встроенная функция 68, 159, 179
resetlocale(), в модуле locale 238
retrieve(), в модуле urllib 366
return, инструкция 117
reverse(), метод объектов list ... 48, 146
RExec(), в модуле rexec 351
rfind()
  метод объектов string и unicode 141
  в модуле string 219
RHooks(), в модуле rexec 351
rindex()
  метод объектов string и unicode 141
  в модуле string 219
rjust()
  метод объектов string и unicode 140
  в модуле string 220
RLock(), в модуле threading 320
rmdir(), в модуле os 275
rmtree(), в модуле shutil 305
rnopen(), в модуле bsddb 331
round(), встроенная функция 179
rstrip()
  метод объектов string и unicode 141
  в модуле string 219
run()
  в модуле pdb 341
  в модуле profile 347
  в модуле sched 295
runcall(), в модуле pdb 341
runeval(), в модуле pdb 341
RuntimeError, исключение 185

S

S, в модуле re 226
S_IFMT(), в модуле stat 284
S_IMODE(), в модуле stat 284
S_ISBLK(), в модуле stat 283
S_ISCHR(), в модуле stat 283
S_ISDIR(), в модуле stat 283
S_ISFIFO(), в модуле stat 283
S_ISLNK(), в модуле stat 284
S_ISREG(), в модуле stat 283
S_ISSOCK(), в модуле stat 284
saferepr(), в модуле pprint 214
samefile(), в модуле os.path 282
sameopenfile(), в модуле os.path 282
samestat(), в модуле os.path 282

SATURDAY, в модуле calendar 419
SAXException, в модуле xml.sax 393
SAXNotRecognizedException, в модуле xml.sax 393
SAXNotSupportedException, в модуле xml.sax 393
SAXParseException, в модуле xml.sax 393
SAXReaderNotAvailable, в модуле xml.sax 393
scheduler(), в модуле sched 294
search(), в модуле re 227
seed()
в модуле random 250
в модуле whrandom 250, 251
seek(), метод объектов file 164, 177
select(), в модуле select 313
Semaphore(), в модуле threading 320
send(), в модуле socket 312
sendto(), в модуле socket 312
sep, в модуле os 280
sequenceIncludes(), в модуле operator 205
set_debug(), в модуле gc 199
set_threshold(), в модуле gc 199
set_trace(), в модуле pdb 342
setattr(), встроенная функция 180
setblocking(), в модуле socket 312
setcheckinterval(), в модуле sys ...196
setdefault(), метод объектов dictionary 148
setdefaultencoding(), в модуле sys 196
setegid(), в модуле os 268
seteuid(), в модуле os 268
setfirstweekday(), в модуле calendar 419
setgid(), в модуле os 268
setitem(), в модуле operator 206
setlocale(), в модуле locale 236
setpgid(), в модуле os 269
setpgrp(), в модуле os 269
setprofile(), в модуле sys 196
setrecursionlimit(), в модуле sys ..197
setregid(), в модуле os 269
setreuid(), в модуле os 269
setsid(), в модуле os 269
setslice(), в модуле operator 206
setsockopt(), в модуле socket 312
settrace(), в модуле sys 197
setuid(), в модуле os 269
SGML 383
SGMLParser(), в модуле sgmllib 383
shlex(), в модуле shlex 415
shutdown(), в модуле socket 312
SIG*, в модуле signal 306
SIG_DFL, в модуле signal 306
SIG_IGN, в модуле signal 306
signal(), в модуле signal 184, 306
sin()
в модуле cmath 248
в модуле math 246
sinh()
в модуле cmath 248
в модуле math 247
sleep(), в модуле time 292
slice, встроенный тип 166, 203
slice(), встроенная функция 166, 180
SliceType, в модуле types 203
SO_*, в модуле socket 308
SOCK_DGRAM, в модуле socket 308
SOCK_RAW, в модуле socket 308
SOCK_RDM, в модуле socket 308
SOCK_SEQPACKET, в модуле socket 308
SOCK_STREAM, в модуле socket 308
socket(), в модуле socket 309
SocketType, в модуле socket 310
softspace, атрибут объектов file 115, 116, 165
SOL_*, в модуле socket 308
SOMAXCONN, в модуле socket 308
sort(), метод объектов list 48, 55, 147
spawnv(), в модуле os 277
spawnve(), в модуле os 277
split()
метод объектов string и unicode 142
в модуле os.path 282
в модуле re 227
в модуле string 219
splitdrive(), в модуле os.path 283
splitext(), в модуле os.path 283
splitfields(), в модуле string 219
splitlines(), метод объектов string и unicode 142
sqrt()
в модуле cmath 248
в модуле math 247
ST_ATIME, в модуле stat 284
ST_CTIME, в модуле stat 285
ST_DEV, в модуле stat 284
ST_GID, в модуле stat 284
ST_INO, в модуле stat 284
ST_MODE, в модуле stat 284

ST_MTIME, в модуле stat ... 284
ST_NLINK, в модуле stat ... 284
ST_SIZE, в модуле stat ... 284
ST_UID, в модуле stat ... 284
StandardError, исключение ... 183
start, атрибут объектов slice ... 166
start_new_thread(), в модуле thread 318
startfile(), в модуле os ... 278
startswith(), метод объектов string и unicode ... 141
stat(), в модуле os ... 275
Stats(), в модуле pstats ... 348
statvfs(), в модуле os ... 275
stderr, в модуле sys ... 163, 197
stdin, в модуле sys ... 163, 197
stdout, в модуле sys ... 163, 197
step, атрибут объектов slice ... 166
stop, атрибут объектов slice ... 166
str()
  в модуле locale ... 239
  встроенная функция ... 68, 71, 159, 180
strcoll(), в модуле locale ... 239
StreamReader(), в модуле codecs ... 234
StreamReaderWriter(), в модуле codecs 234
StreamRecoder(), в модуле codecs ... 235
StreamWriter(), в модуле codecs ... 234
strerror(), в модуле os ... 267
strftime(), в модуле time ... 292
string, встроенный тип ... 26, 135, 136, 145, 202
StringIO(), в модулях pickle и cPickle 232
StringType, в модуле types ... 202
strip()
  метод объектов string и unicode ... 141
  в модуле string ... 219
strptime(), в модуле time ... 293
strxfrm(), в модуле locale ... 239
sub()
  в модуле operator ... 204
  в модуле re ... 227
subn(), в модуле re ... 228
SUNDAY, в модуле calendar ... 419
swapcase()
  метод объектов string и unicode ... 143
  в модуле string ... 219
symlink(), в модуле os ... 275
SyntaxError, исключение ... 28, 185
sysconf(), в модуле os ... 279
sysconf_names, в модуле os ... 279
system(), в модуле os ... 278
SystemError, исключение ... 186
SystemExit, исключение . 186, 194, 318, 319

T

T, в модуле re ... 226
TabError, исключение ... 426
tan()
  в модуле cmath ... 248
  в модуле math ... 247
tanh()
  в модуле cmath ... 248
  в модуле math ... 247
tb_frame, атрибут объектов traceback 168, 169
tb_lasti, атрибут объектов traceback 169
tb_lineno, атрибут объектов traceback 169
tb_lineno(), в модуле traceback ... 210
tb_next, атрибут объектов traceback . 169
tcgetpgrp(), в модуле os ... 272
tcsetpgrp(), в модуле os ... 272
tell(), метод объектов file ... 164
tempdir, в модуле tempfile ... 299
TEMPLATE, в модуле re ... 226
template, в модуле tempfile ... 299
tempnam(), в модуле os ... 275
TemporaryFile(), в модуле tempfile 298
test(), в модуле cgi ... 360
textdomain(), в модуле gettext ... 241
Thread(), в модуле threading ... 321
THURSDAY, в модуле calendar ... 419
time(), в модуле time ... 294
timegm(), в модуле calendar ... 421
times(), в модуле os ... 278
timezone, в модуле time ... 294
title(), метод объектов string и unicode ... 143
TMP_MAX, в модуле os ... 275
tmpfile(), в модуле os ... 270
tmpnam(), в модуле os ... 275
traceback, встроенный тип .. 169, 193, 204, 208
tracebacklimit, в модуле sys ... 197
TracebackType, в модуле types ... 204
translate()
  метод объектов string ... 142
  метод объектов unicode ... 144
  в модуле string ... 219
translation(), в модуле gettext ... 242
truncate(), метод объектов file ... 164
truth(), в модуле operator ... 205

try, инструкция 21, 77, 124, 182
ttyname(), в модуле os 272
TUESDAY, в модуле calendar 419
tuple, встроенный тип 53, 135, 144, 202
tuple(), встроенная функция 180
TupleType, в модуле types 202
type, встроенный тип 165, 202
type(), встроенная функция 165, 180
TypeError, исключение 44, 77, 186
TypeType, в модуле types 202
tzname, в модуле time 294

U

U, в модуле re 227
umask(), в модуле os 269
uname(), в модуле os 269
UnboundLocalError, исключение 185
UnboundMethodType, в модуле types ..203
unichr(), встроенная функция 180
UNICODE, в модуле re 227
unicode, встроенный тип . 30, 135, 136, 143, 145, 202
unicode(), встроенная функция 31, 180
UnicodeError, исключение ...109, 142, 186
UnicodeType, в модуле types 202
uniform()
 в модуле random 250
 в модуле whrandom 250, 251
unlink(), в модуле os 274
unpack(), в модуле struct 264
Unpacker(), в модуле xdrlib 379
Unpickler(), в модулях pickle и cPickle 260
UnpicklingError, в модулях pickle и cPickle 262
unquote(), в модуле urllib 364
unquote_plus(), в модуле urllib 365
unregister(), в модуле select 314
update(), метод объектов dictionary 148
upper()
 метод объектов string и unicode 143
 в модуле string 219
uppercase, в модуле string 218
URI 388, 406
URL 363, 367
urlcleanup(), в модуле urllib 364
urlencode(), в модуле urllib 365
urljoin(), в модуле urlparse 368
urlopen(), в модуле urllib 364
URLopener(), в модуле urllib 365
urlparse(), в модуле urlparse 367
urlretrieve(), в модуле urllib 364
urlunparse(), в модуле urlparse 368
UserDict(), в модуле UserDict 257
UserList(), в модуле UserList 256
UserString(), в модуле UserString ..255
UTC 290
utime(), в модуле os 276

V

ValueError, исключение 186
values(), метод объектов dictionary 148
vars(), встроенная функция 72, 181
VERBOSE, в модуле re 226
version
 в модуле sys 198
 в модуле urllib 366
version_info, в модуле sys 198
vonmisesvariate(), в модуле random 249

W

W_OK, в модуле os 273
wait(), в модуле os 278
waitpid(), в модуле os 278
walk(), в модуле os.path 283
WEDNESDAY, в модуле calendar 419
weekday(), в модуле calendar 420
weibullvariate(), в модуле random ..250
WEXITSTATUS(), в модуле os 279
whichdb(), в модуле whichdb 330
while, инструкция 34, 39, 110, 116, 122
whitespace 143, 218
whitespace, в модуле string 218
whrandom(), в модуле whrandom 250
WIFEXITED(), в модуле os 279
WIFSIGNALED(), в модуле os 279
WIFSTOPPED(), в модуле os 278
WindowsError, исключение 184
winver, в модуле sys 198
WNOHANG, в модуле os 278
write()
 метод объектов file 74, 116, 164
 в модуле os 272
writelines(), метод объектов file ... 164
WSTOPSIG(), в модуле os 279
WTERMSIG(), в модуле os 279

X

X, в модуле re 226
X_OK, в модуле os 273
XDR 378
XML 388, 392, 404
XML_ERROR_*, в модуле xml.parsers.expat.errors 389

XMLFilterBase(), в модуле
 xml.sax.saxutils 398
XMLGenerator(), в модуле
 xml.sax.saxutils 398
XMLParser(), в модуле xmllib 404
XMLParserType, в модуле
 xml.parsers.expat 389
XMLReader(), в модуле
 xml.sax.xmlreader 399
xor(), в модуле operator 205
xrange, встроенный тип 135, 144, 202
xrange(), встроенная функция 37, 181
XRangeType, в модуле types 202

Z

Z_BEST_COMPRESSION, в модуле zlib ..333
Z_BEST_SPEED, в модуле zlib 333
Z_DEFAULT_COMPRESSION, в модуле zlib 333
Z_DEFAULT_STRATEGY, в модуле zlib ..334
Z_DEFLATED, в модуле zlib 334
Z_FILTERED, в модуле zlib 334
Z_FINISH, в модуле zlib 335
Z_FULL_FLUSH, в модуле zlib 335
Z_HUFFMAN_ONLY, в модуле zlib 334
Z_NO_COMPRESSION, в модуле zlib 333
Z_SYNC_FLUSH, в модуле zlib 335
ZeroDivisionError, исключение ..77, 183
zfill(), в модуле string 218
zip(), встроенная функция 50, 181
ZIP_DEFLATED, в модуле zipfile 337
ZIP_STORED, в модуле zipfile 337
ZipFile(), в модуле zipfile 337
ZipInfo(), в модуле zipfile 337
ZLIB_VERSION, в модуле zlib 333

А

аргументы
 значения по умолчанию 42
 именованные 43, 44, 46, 107, 108
 позиционные 43, 44, 46, 107
арифметические операторы 23, 108, 133, 134, 160
атрибуты 83, 86
 статические 90
 частные 88

Б

битовые операторы 108, 160
буквы
 прописные 143, 218
 строчные 143, 217

В

ввода, стандартный поток . 163, 178, 197, 288
вещественные числа, *см.* float
временный файл 298
встроенная функция
 __import__() 119, 170
 abs() 25, 160, 171, 205
 apply() 46, 171
 buffer() 145, 171
 callable() 148, 171
 chr() 171
 cmp() 147, 154, 171, 239
 coerce() 171
 compile() 117, 167, 172, 173
 complex() 24, 132, 160, 172
 delattr() 172
 dir() 63, 172
 divmod() 161, 172
 eval() 117, 119, 130, 167, 172
 execfile() 117, 130, 173
 filter() 49, 173
 float() 25, 132, 160, 173
 getattr() 173
 globals() 119, 174
 hasattr() 174
 hash() 155, 174
 hex() 159, 174
 id() 174
 input() 129, 174, 197
 int() 25, 132, 160, 174
 intern() 175
 isinstance() 175
 issubclass() 175
 len() 30, 33, 38, 135, 147, 157, 175
 list() 175
 locals() 119, 175
 long() 25, 132, 160, 176
 map() 49, 176, 207
 max() 176
 min() 176
 oct() 159, 176
 open() 72, 163, 176
 ord() 177
 pow() 161, 177
 range() 37, 177
 raw_input() 178, 197
 reduce() 51, 178, 207
 reload() 179, 195, 211
 repr() 68, 159, 179
 round() 179

setattr() 180
slice() 166, 180
str() 68, 71, 159, 180
tuple() 180
type() 165, 180
unichr() 180
unicode() 31, 180
vars() 72, 181
xrange() 37, 181
zip() 50, 181
встроенный тип
buffer 135, 145, 202
builtin_function_or_method . 150, 203
class 85, 151, 152, 203
code 166, 203, 263
complex 24, 131, 202
dictionary 55, 95, 147, 155, 203
ellipsis 166, 203
file 72, 163, 203, 311
float 24, 131, 132, 202
frame 168, 204
function 125, 149, 152, 203
instance 86, 151, 152, 203
instance method 87, 149
int 23, 131, 132, 202
list 32, 47, 48, 51, 135, 145, 202
long int 131, 132, 202
method 203
module 151, 203
None 165, 202
slice 166, 203
string 26, 135, 136, 145, 202
traceback 169, 193, 204, 208
tuple 53, 135, 144, 202
type 165, 202
unicode 30, 135, 136, 143, 145, 202
xrange 135, 144, 202
вывода, стандартный поток 115, 163, 178, 197, 288
вызова оператор 46, 107, 148, 149, 156
выражение 104, 113
регулярное 220

Д

деструктор 153
длинные целые числа, *см.* long int
документации строка 40, 45, 61, 149–152, 168
доступа к атрибуту оператор . 83, 85, 86, 106, 151, 155
доступа к элементу по индексу/ключу
оператор 28, 107, 135, 145, 147, 157, 206

З

зарезервированные слова 105

И

идентификатор 105
объекта 174
изменяемые последовательности 145
именованные аргументы . 43, 44, 46, 107, 108
имя 105
индекс 28
инструкции
простые 112
составные 121
инструкция
assert 61, 120, 183
break 39, 116
class 85, 128
continue 39, 116
def 40, 125, 149
del 52, 55, 115, 145, 147
exec 117, 119, 130, 167
for 36, 39, 116, 123
global 40, 105, 117, 119, 128, 185
if 36, 110, 122
import 59, 65, 117, 170, 211
pass 39, 115
print 27, 35, 68, 115, 159
raise 92, 120
return 117
try 21, 77, 124, 182
while 34, 39, 110, 116, 122
выражение 113
присваивания 24, 113
исключение 76, 120
ArithmeticError 183
AssertionError 120, 183
AttributeError 107, 183
EnvironmentError 184
EOFError 178, 183
Exception 93, 182
FloatingPointError 183
ImportError 65, 184, 211, 212
IndexError 29, 94, 123, 135, 185
IOError 163, 184
KeyboardInterrupt 21, 77, 184
KeyError 55, 147, 157, 185
LookupError 184
MemoryError 185
NameError 53, 77, 105, 123, 129, 185
NotImplementedError 185
OSError 184, 267
OverflowError 174, 183, 292

  RuntimeError ... 185
  StandardError ... 183
  SyntaxError ... 28, 185
  SystemError ... 186
  SystemExit ... 186, 194, 318, 319
  TabError ... 426
  TypeError ... 44, 77, 186
  UnboundLocalError ... 185
  UnicodeError ... 109, 142, 186
  ValueError ... 186
  WindowsError ... 184
  ZeroDivisionError ... 77, 183
истина ... 110
истинность ... 110, 205

К

класс, *см.* class
ключевые слова ... 105
комплексные числа, *см.* complex
конструктор ... 153
контрольная сумма ... 334
кортеж, *см.* tuple

Л

лексикографическое сравнение ... 57
литералы
  строковые ... 136
  числовые ... 131
литеральные выражения ... 106
логические операторы ... 56, 110
ложь ... 110

М

массив ... 252
метод
  встроенный, *см.* builtin_function_or_method
  определенный пользователем, *см.* instance method
  экземпляра, *см.* instance method
множественное присваивание ... 34
модуль, *см.* module

О

области видимости ... 84
объединения последовательностей оператор 27, 135, 205
объект
  Ellipsis ... 166
  None ... 41, 43, 110, 165
оператор
  () ... 46, 107, 148, 149, 156
  * ... 27, 108, 133, 135, 161, 204, 205
  ** ... 108, 134, 161
  +, бинарный 27, 108, 133, 135, 161, 204, 205
  +, унарный ... 108, 134, 160, 204
  −, бинарный ... 108, 133, 161, 204
  −, унарный ... 108, 134, 160, 204
  . ... 83, 85, 86, 106, 151, 155
  / ... 108, 134, 161, 204
  < ... 109
  <= ... 109
  <> ... 109
  << ... 108, 134, 161, 205
  == ... 109
  > ... 109
  >= ... 109
  >> ... 108, 134, 161
  [] ... 28, 107, 135, 145–147, 157, 166, 206
  % ... 68, 108, 134, 137, 159, 161, 204
  & ... 108, 134, 161, 205
  ^ ... 108, 134, 161, 205
  | ... 108, 134, 161, 205
  ~ ... 108, 134, 160, 205
  and ... 56, 111
  in ... 56, 110, 135, 159, 205
  is ... 56, 109
  is not ... 56, 110
  lambda ... 45, 111, 149
  not ... 56, 111, 205
  not in ... 56, 110, 135, 159
  or ... 56, 111
  вызова ... 46, 107, 148, 149, 156
  доступа к атрибуту 83, 85, 86, 106, 151, 155
  доступа к элементу по индексу/ключу ..28, 107, 135, 145, 147, 157, 206
  объединения последовательностей ..27, 135, 205
  размножения последовательностей .27, 135, 205
  среза ... 28, 107, 135, 146, 206
    расширенная запись ... 107, 166
  форматирования ... 68, 137, 159
операторы
  арифметические ... 23, 108, 133, 134, 160
  битовые ... 108, 160
  логические ... 56, 110
  сравнения ... 34, 108, 109
  условные ... 56
определение
  функции ... 125
  класса ... 128
отладочное утверждение ... 120
отображения ... 147
очередь ... 48, 325

ошибок, стандартный поток 163, 197, 288

П

пакет 64
переменная окружения
  LANG 240, 242
  LANGUAGE 240, 242
  LC_ALL 240, 242
  LC_MESSAGES 240, 242
  PYTHONDEBUG 426
  PYTHONHOME 427
  PYTHONINSPECT 426
  PYTHONOPTIMIZE 61, 427
  PYTHONPATH 61, 62, 119, 196, 427
  PYTHONSTARTUP 21, 191, 427
  PYTHONUNBUFFERED 427
  PYTHONVERBOSE 427
  PYTHONY2K 290, 291
позиционные аргументы 43, 44, 46, 107
порядок следования байтов 192
последовательности 135
  изменяемые 145
присваивание 24, 113
  множественное 34
прописные буквы 143, 218
пространства имен 83
пространство имен 128
простые инструкции 112

Р

размножения последовательностей оператор
  27, 135, 205
расширенная запись среза 107, 166
регулярное выражение 220

С

семафор 320
символы пропуска, *см.* whitespace
словарь, *см.* dictionary
составные инструкции 121
специальные методы 153
список, *см.* list
сравнения операторы 34, 108, 109
среза
  оператор
    расширенная запись 107, 166
среза оператор 28, 107, 135, 146, 206
стандартный поток
  ошибок 163, 197, 288
  ввода 163, 178, 197, 288
  вывода 115, 163, 178, 197, 288
статические атрибуты 90
стек 48
строка, *см.* string
  документации 40, 45, 61, 149–152, 168
строка Unicode, *см.* unicode
строковое представление 68, 159, 179
строковые литералы 136
строчные буквы 143, 217

У

условные операторы 56

Ф

файл, временный 298
форматирования оператор 68, 137, 159
функция
  встроенная, *см.*
    builtin_function_or_method
  определенная пользователем, *см.* function

Ц

целочисленное деление 134
целые числа, *см.* int

Ч

частные атрибуты 88
числа
  вещественные, *см.* float
  длинные целые, *см.* long int
  комплексные, *см.* complex
  с плавающей точкой, *см.* float
  целые, *см.* int
числовые литералы 131

Э

экземпляр класса, *см.* instance
эпоха 289