Даг Хеллман

2-е издание

Стандартная библиотека Python 3

Справочник с примерами

Стандартная библиотека Python 3

Справочник с примерами

2-е издание

Даг Хеллман

Диалектика

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Х36
УДК 681.3.07

Компьютерное издательство "Диалектика"

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Хеллман, Даг.

Х36 Стандартная библиотека Python 3: справочник с примерами, 2-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 1376 с. : ил. — Парал. тит. англ.

ISBN 978-5-6040043-8-8 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Даг Хеллман

Стандартная библиотека Python 3: справочник с примерами

2-е издание

Подписано в печать 18.12.2018. Формат 70×100/16
Гарнитура Times
Усл. печ. л. 110,94. Уч.-изд. л. 54,9
Доп. тираж 400 экз. Заказ № 16314

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-6040043-8-8 (рус.)
ISBN 978-0-13-429105-5 (англ.)

Оглавление

Содержание

Об авторе

Даг Хеллман в настоящее время занимается разработкой программных средств для облачных вычислений в компании Red Hat, является членом технического комитета международного сообщества OpenStack и принимает активное участие в разработке многих направлений этого проекта. Он программирует на Python со времен выхода версии 1.4, уделяя основное внимание проектам для различных платформ в таких областях, как картография, банковские операции и дата-центры. Даг – номинированный член организации Python Software Foundation, пресс-секретарем которой был с 2010 по 2012 г. Уже через год после того, как он начал публиковать обзорные статьи в *Python Magazine*, Даг стал его главным редактором и занимал эту должность с 2008 по 2009 г. В период с 2007 по 2011 год Даг опубликовал в своем блоге серию популярных статей *Python Module of the Week* и выпустил первое издание книги *The Python Standard Library by Example*, посвященное Python 2. В настоящее время живет в Афинах, штат Джорджия.

Об изображении на обложке

Музей Орсе, Париж, Франция

Музей изобразительных искусств Орсе располагается на левом берегу Сены в бывшем здании первого в мире электрифицированного железнодорожного вокзала, поражающего своей архитектурой в стиле боз-ар. Сам вокзал был возведен на месте дворца Орсе, уничтоженного пожаром в 1871 году во времена Парижской коммуны. Открытие вокзала состоялось 28 мая 1900 года во время Всемирной выставки в Париже. После присвоения ему статуса исторического памятника в 1978 году сооружение было преобразовано в музей. Работами руководила архитектурная группа АСТ в составе Рено Бардона, Пьера Колбока и Жана-Поля Филиппона. Итальянский архитектор Гаэ Ауленти превратила огромное помещение вокзала в единое музейное пространство с гармоничным стилем оформления каменных стен и пола. Фонд нового музея, открытого в 1986 году, составили три крупные художественные коллекции, относящиеся к периоду 1848–1914 гг. В настоящее время музей ежегодно посещают свыше трех миллионов человек, которые имеют возможность любоваться картинами таких великих художников, как Сезанн, Курбе, Дега, Гоген, Мане, Моне и Ренуар.

Предисловие

Мое знакомство с Python состоялось примерно в 1997 году благодаря Дику Уоллу, с которым мы в то время разрабатывали программное обеспечение для геоинформационной системы в компании ERDAS. Я хорошо помню, как радовался тому, что нашел новый язык, легко применяемый на практике, и одновременно испытывал огорчение от того, что компания не разрешила использовать его в практической работе. Где бы я ни работал с тех пор, я интенсивно использовал Python, испытывая от этого огромное удовольствие, за что должен поблагодарить Дика.

Основная команда разработчиков Python создала мощную экосистему, включающую сам язык, инструменты и библиотеки, которая продолжает набирать популярность и находит все новые области применения. Не будь этой экосистемы, мы постоянно тратили бы свое время на изобретение очередного колеса.

Первоначально материал книги увидел свет в виде публикаций в моем блоге. Без необычайно положительной оценки читателями блога эти статьи не были бы обновлены для того, чтобы соответствовать версии Python 3, и эта книга просто не появилась бы. Каждая из статей в блоге была отрецензирована и прокомментирована членами сообщества Python с внесением соответствующих исправлений, доработок и рекомендаций, от которых книга существенно выиграла.

Даг Хелльман

Введение

В состав каждого дистрибутива Python входит стандартная библиотека. Она содержит сотни модулей, которые предоставляют инструменты, предназначенные для взаимодействия с операционной системой, интерпретатором и Интернетом. Все они тщательно протестированы и готовы к немедленному использованию в приложениях. В этой книге, созданной на основе ставшей популярной серии статей *Python Module of the Week (PyMOTW)*, которая публиковалась в моем блоге, представлены избранные примеры, демонстрирующие возможности наиболее часто используемых модулей, в полной мере соответствующих одному из девизов Python: "Батарейки входят в комплект".

Для кого предназначена эта книга

Книга рассчитана на программистов средней квалификации, разрабатывающих программы на языке Python. В связи с этим, несмотря на то что каждый из приведенных примеров сопровождается обсуждением, объяснение отдельных строк кода приводится лишь в редких случаях. Во всех разделах основное внимание уделено демонстрации возможностей модулей на примере полностью автономных фрагментов исходного кода, обеспечивающих получение конечного результата. Для иллюстрации применения каждого средства используется как можно более компактный код, чтобы внимание читателя не отвлекалось на второстепенные детали, а фокусировалось на демонстрируемых возможностях модуля или функции.

Опытные программисты, знакомые с другими языками, могут использовать книгу для изучения языка Python, но при написании текста не ставилась задача сделать книгу введением в этот язык программирования. Наибольшую пользу изучение примеров принесет тем читателям, которые уже имеют опыт написания программ на Python.

Некоторые разделы, например те, в которых речь идет о сетевом программировании с помощью сокетов или о механизме HMAC, требуют определенных знаний в соответствующей предметной области. Примеры сопровождаются необходимыми пояснениями, однако диапазон тем, с которыми связаны модули стандартной библиотеки, настолько широк, что дать исчерпывающие объяснения по каждой теме в рамках одной книги совершенно нереально. Обсуждение каждого модуля завершается приведением ссылок на рекомендованные источники дополнительной информации, включая онлайн-ресурсы, документы RFC и книги.

Учет различий между версиями Python 3 и Python 2

В настоящее время в сообществе Python происходит переход от версии Python 2 к версии Python 3. Как это всегда бывает при смене старшего номера версии программного продукта, версии Python 2 и Python 3 несовместимы во многих отношениях, и речь идет не только о языке. Довольно много стандартных библиотечных модулей получили в Python 3 новые имена либо были так или иначе реорганизованы.

Несмотря на обновление экосистемы библиотек и инструментов Python с целью ее адаптации для работы в версии Python 3, сообщество разработчиков Python осознавало, что для устранения последствий несовместимости версий потребуется длительный переходный период. И хотя многие проекты все еще зависят от Python 2, в настоящее время эта версия получает лишь обновления безопасности, а к 2020 году ее официальная поддержка будет полностью прекращена. Новые средства включаются только в выпуски Python 3.

Написание программ, способных выполняться в обеих версиях Python, может оказаться трудной задачей, однако вполне осуществимой. Для этого необходимо тщательно исследовать различия между версиями, в которых должна выполняться программа, и использовать имена импортируемых модулей или аргументов вызываемых классов и функций, соответствующие конкретной версии. Существует ряд сторонних инструментов, не входящих в состав стандартной библиотеки, которые способны упростить этот процесс. Чтобы избежать излишнего усложнения примеров для учета этих различий и ограничиться лишь средствами стандартной библиотеки, примеры были подготовлены для выполнения в версии Python 3. Все примеры протестированы с использованием версии Python 3.5 (текущий выпуск линейки 3.*x* на момент написания книги) и могут не работать в версии Python 2, если не внести в них соответствующие изменения.

Ради сохранения максимальной ясности и лаконичности в описаниях примеров различия между версиями Python 2 и Python 3 в каждом конкретном случае не обсуждаются. Соответствующие замечания, касающиеся наиболее важных различий между версиями, вынесены в приложение А, которое пригодится вам при портировании программ из Python 2 в Python 3.

Структура книги

Эта книга дополняет официальное справочное руководство по языку Python (доступное по адресу `http://docs.python.org`), предоставляя читателю примеры полностью функциональных программ, демонстрирующие работу средств, представленных в руководстве. Описания модулей сгруппированы по главам, каждая из которых охватывает определенную тему. Маловероятно, что кто-то будет читать книгу от корки до корки, тем не менее она организована таким образом, чтобы количество ссылок на модули, которые рассматриваются в последующих разделах, было сведено к минимуму, хотя полностью избежать этого было просто невозможно.

Файлы примеров и дополнительные материалы

Исходные версии статей из авторского блога доступны по адресу `https://pymotw.com/3/`. Файлы примеров можно скачать на сайте автора:

```
https://doughellmann.com/blog/the-python-3-standard-library-by-example/
```

Они также доступны на сайте издательства "Диалектика" по следующему адресу:

```
http://www.williamspublishing.com/Books/978-5-6040043-8-8.html
```

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

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

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

Наши электронные адреса:

E-mail: info@dialektika.com

WWW: www.dialektika.com

Глава 1

Текст

Наиболее очевидным из всех инструментов, которые могут быть использованы программистами на языке Python для обработки текста, является класс `str`, однако стандартная библиотека Python предлагает множество других инструментов, значительно упрощающих манипулирование текстом.

Класс `string.Template` обеспечивает простой способ параметризации строк, недоступный для объектов `str`. И хотя возможности создаваемых с его помощью шаблонов более скромные по сравнению с теми, которые предлагают шаблоны многих веб-фреймворков или модулей расширения, доступных в каталоге пакетов Python Package Index (PyPI), класс `string.Template` предоставит вам неплохое промежуточное решение для создания шаблонов, позволяющих включать динамические значения в статический текст.

Модуль `textwrap` (раздел 1.2) содержит инструменты, предназначенные для форматирования текстовых абзацев путем ограничения ширины вывода, добавления отступов и вставки разрывов строк для последовательно применяемого автоматического перехода текста на новую строку.

Стандартная библиотека включает два модуля, расширяющих возможности сравнения текстовых значений по сравнению с возможностями аналогичных встроенных операций, поддерживаемых строковыми объектами. Модуль `re` (раздел 1.3) предоставляет полноценную библиотеку регулярных выражений, реализованную на языке C для повышения быстродействия. Регулярные выражения хорошо приспособлены для поиска подстрок в больших наборах данных, сравнения строк с шаблонами более сложной структуры, чем фиксированные строки, и выполнения операций средней сложности по синтаксическому анализу (парсингу) текста.

В противоположность этому модуль `difflib` (раздел 1.4) позволяет обнаруживать фактические различия между последовательностями текстовых символов в терминах добавленных, удаленных или измененных элементов. Результаты, возвращаемые функциями сравнения модуля `difflib`, могут быть использованы для организации обратной связи с пользователем на основе информации о том, в каких местах два документа отличаются один от другого, как изменялся документ с течением времени и т.п.

1.1. `string`: текстовые константы и шаблоны

Модуль `string` берет свое начало в ранних версиях Python. Многие из когда-то реализованных в нем функций перешли в методы объектов `str`, но в этом модуле все еще хранится несколько полезных констант и классов, предназначенных для работы с объектами `str`. Именно на них будет сосредоточено внимание в этом разделе.

1.1.1. Функции

Функция `capwords()` переводит в верхний регистр первую букву каждого слова в строке.

Листинг 1.1. `string_capwords.py`

```
import string

s = 'The quick brown fox jumped over the lazy dog.'
print(s)
print(string.capwords(s))
```

Результат совпадает с тем, который мы могли бы получить, создав список слов исходной строки с помощью метода `split()`, переведя первую букву каждого слова в результирующем списке в верхний регистр и вновь объединив все слова в одну строку с помощью метода `join()`.

```
$ python3 string_capwords.py

The quick brown fox jumped over the lazy dog.
The Quick Brown Fox Jumped Over The Lazy Dog.
```

1.1.2. Шаблоны

Строковые шаблоны были предложены в документе **PEP 292**[1] в качестве альтернативы встроенному синтаксису замены строк. В соответствии с синтаксисом, поддерживаемым классом `string.Template`, переменные идентифицируются префиксом в виде символа `$`, помещаемого перед именем переменной (например, `$var`). Если необходимо отделить переменную от окружающего текста, то ее можно заключить в фигурные скобки (например, `${var}`).

Ниже приведен пример, позволяющий сравнить три подхода к форматированию одной и той же строки: с помощью простого шаблона, оператора `%` и нового синтаксиса форматирующей строки с использованием метода `str.format()`.

Листинг 1.2. `string_template.py`

```
import string

values = {'var': 'foo'}

t = string.Template("""
Variable        : $var
Escape          : $$
Variable in text: ${var}iable
""")

print('TEMPLATE:', t.substitute(values))

s = """
Variable        : %(var)s
Escape          : %%
```

[1] www.python.org/dev/peps/pep-0292

```
Variable in text: %(var)siable
"""

print('INTERPOLATION:', s % values)

s = """
Variable        : {var}
Escape          : {{}}
Variable in text: {var}iable
"""

print('FORMAT:', s.format(**values))
```

В первых двух случаях триггерный символ (`$` или `%`) экранируется посредством повторения самого символа. При форматировании с помощью метода `format()` тем же способом должна экранироваться каждая из фигурных скобок (`{` и `}`).

```
$ python3 string_template.py

TEMPLATE:
Variable        : foo
Escape          : $
Variable in text: fooiable

INTERPOLATION:
Variable        : foo
Escape          : %
Variable in text: fooiable

FORMAT:
Variable        : foo
Escape          : {}
Variable in text: fooiable
```

Ключевым отличием шаблонов от строковых подстановок (интерполяции) и форматирования с помощью метода `format()` является то, что они не учитывают тип аргументов. Значения преобразуются в строки, а строки вставляются в результат. Опции форматирования полностью отсутствуют. Например, не существует способа управлять количеством цифр, используемых для представления значений с плавающей запятой.

Однако шаблоны обладают тем неоспоримым преимуществом, что использование их метода `safe_substitute()` позволяет избежать возникновения исключений в тех случаях, когда не все значения, в которых нуждаются шаблоны, представлены аргументами.

Листинг 1.3. `string_template_missing.py`

```
import string

values = {'var': 'foo'}

t = string.Template("$var is here but $missing is not provided")
```

```
try:
    print('substitute() :', t.substitute(values))
except KeyError as err:
    print('ERROR:', str(err))

print('safe_substitute():', t.safe_substitute(values))
```

Отсутствие в словаре значения для переменной `missing` приводит к тому, что метод `substitute()` возбуждает исключение `KeyError`. В то же время метод `safe_substitute()` вместо возбуждения исключения перехватывает его и оставляет в тексте само выражение переменной.

```
$ python3 string_template_missing.py

ERROR: 'missing'
safe_substitute(): foo is here but $missing is not provided
```

1.1.3. Более сложные шаблоны

Синтаксис класса `string.Template`, принятый по умолчанию, можно изменить, адаптируя используемое им регулярное выражение (РВ) для нахождения имен переменных в теле шаблона. Для этого достаточно изменить атрибуты `delimiter` (разделитель) и `idpattern` (шаблон РВ).

Листинг 1.4. `string_template_advanced.py`

```
import string

class MyTemplate(string.Template):
    delimiter = '%'
    idpattern = '[a-z]+_[a-z]+'

template_text = '''
  Delimiter : %%
  Replaced  : %with_underscore
  Ignored   : %notunderscored
'''

d = {
    'with_underscore': 'replaced',
    'notunderscored': 'not replaced',
}

t = MyTemplate(template_text)
print('Modified ID pattern:')
print(t.safe_substitute(d))
```

В этом примере правила подстановки изменены: в качестве разделителя установлен символ `%` вместо символа `$`, а обязательным условием замены переменной ее значением является наличие символа подчеркивания посреди ее имени. Шаблон строки `%notunderscored` не заменяется соответствующим значением, поскольку в нем не содержится символ подчеркивания.

```
$ python3 string_template_advanced.py

Modified ID pattern:
Delimiter : %
Replaced  : replaced
Ignored   : %notunderscored
```

Если требуются еще более сложные изменения, можете переопределить атрибут pattern и задать другое регулярное выражение. Новый шаблон PB должен содержать четыре именованные группы, захватывающие соответственно шаблоны экранированного разделителя, именованной переменной, именованной переменной в фигурных скобках и выражения с некорректным использованием разделителя.

Листинг 1.5. string_template_defaultpattern.py

```
import string

t = string.Template('$var')
print(t.pattern.pattern)
```

Значением атрибута t.pattern является скомпилированное регулярное выражение, но исходная строка доступна через его атрибут pattern.

```
\$(?:
(?P<escaped>\$) |                # Два разделителя
(?P<named>[_a-z][_a-z0-9]*) |    # Идентификатор
{(?P<braced>[_a-z][_a-z0-9]*)} | # Идентификатор в фигурных
                                 # скобках
(?P<invalid>)                    # Некорректное выражение
                                 # с разделителем
)
```

В следующем примере определяется новый шаблон регулярного выражения (pattern), и на его основе создается новый тип строкового шаблона, в котором для переменных используется синтаксис {{var}}.

Листинг 1.6. string_template_newsyntax.py

```
import re
import string

class MyTemplate(string.Template):
    delimiter = '{{'
    pattern = r'''
    \{\{(?:
    (?P<escaped>\{\{)|
    (?P<named>[_a-z][_a-z0-9]*)\}\}|
    (?P<braced>[_a-z][_a-z0-9]*)\}\}|
    (?P<invalid>)
    )
    '''
```

```
t = MyTemplate('''
{{{{
{{var}}
''')

print('MATCHES:', t.pattern.findall(t.template))
print('SUBSTITUTED:', t.safe_substitute(var='replacement'))
```

Как следует из кода примера, шаблоны именованных (named) и заключенных в фигурные скобки (braced) переменных должны предоставляться по отдельности, даже если они совпадают. Выполнив этот код, вы получите следующие результаты.

```
$ python3 string_template_newsyntax.py

MATCHES: [('{{', '', '', ''), ('', 'var', '', '')]
SUBSTITUTED:
{{
replacement
```

1.1.4. Класс `Formatter`

Класс `Formatter` реализует тот же самый язык спецификаций, что и метод `format()` класса `str`. Язык спецификаций охватывает приведение типов, выравнивание текста, ссылки на поля и атрибуты, именованные и позиционные аргументы шаблонов, а также зависящие от типа опции форматирования. В большинстве случаев метод `format()` предлагает более удобный интерфейс для доступа к этим средствам, однако класс `Formatter` позволяет создавать подклассы, которые можно адаптировать для внесения в язык новых элементов.

1.1.5. Константы

В модуле `string` определен ряд констант, имеющих отношение к таблице ASCII и символьным наборам.

Листинг 1.7. `string_constants.py`

```
import inspect
import string

def is_str(value):
    return isinstance(value, str)

for name, value in inspect.getmembers(string, is_str):
    if name.startswith('_'):
        continue
    print('%s=%r\n' % (name, value))
```

Константы модуля `string` полезны при работе с ASCII-данными, но ввиду все возрастающей популярности различных кодировок стандарта Unicode сфера применимости этих констант ограничена.

```
$ python3 string_constants.py

ascii_letters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
XYZ'

ascii_lowercase='abcdefghijklmnopqrstuvwxyz'

ascii_uppercase='ABCDEFGHIJKLMNOPQRSTUVWXYZ'

digits='0123456789'

hexdigits='0123456789abcdefABCDEF'

octdigits='01234567'

printable='0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
RSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'

punctuation='!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

whitespace=' \t\n\r\x0b\x0c'
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный классу `string`[2].
- *String Methods*[3]. Методы объектов `str`, которые заменяют строковые функции, признанные устаревшими.
- **PEP 292**[4]. *Simpler String Substitutions*.
- *Format String Syntax*[5]. Формальное определение языка спецификаций формата, используемого в классе `Formatter` и методе `str.format()`.

1.2. `textwrap`: форматирование текстовых абзацев

Модуль `textwrap` может использоваться для форматирования текста в ситуациях, когда требуется красиво оформленный вывод. Он предоставляет функциональность, аналогичную средствам автоматического выравнивания и распределения текста в пределах абзаца, предлагаемым многими текстовыми редакторами и издательскими системами.

1.2.1. Данные, используемые в примерах

В приведенных в этом разделе примерах используется модуль `textwrap_example.py`, в котором содержится строка `sample_text`.

[2] www.python.org/dev/peps/pep-0292

[3] https://docs.python.org/3/library/stdtypes.html#string-methods

[4] www.python.org/dev/peps/pep-0292

[5] https://docs.python.org/3.5/library/string.html#format-string-syntax

Листинг 1.8. textwrap_example.py

```
sample_text = '''
    The textwrap module can be used to format text for output in
    situations where pretty-printing is desired. It offers
    programmatic functionality similar to the paragraph wrapping
    or filling features found in many text editors.
    '''
```

1.2.2. Заполнение абзацев с помощью функции fill

Функция fill() получает текст в качестве входных данных и возвращает текст, отформатированный с учетом ширины заданной области.

Листинг 1.9. textwrap_fill.py

```
import textwrap
from textwrap_example import sample_text

print(textwrap.fill(sample_text, width=50))
```

Результат все еще оставляет желать лучшего. Теперь текст выровнен по левому краю, но первая строка осталась выделенной отступом, а пробелы, располагавшиеся до этого в начале каждой последующей строки, перешли в тело абзаца.

```
$ python3 textwrap_fill.py

    The textwrap module can be used to format
text for output in     situations where pretty-
printing is desired. It offers     programmatic
functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

1.2.3. Удаление существующих отступов

В предыдущем примере символы табуляции и дополнительные пробелы смешались с текстом, поэтому такое форматирование нельзя считать приемлемым. Удаление общего пробельного префикса из всех строк с помощью функции dedent() приводит к лучшему результату и дает возможность использовать строки документирования и многострочный текст непосредственно из кода на Python, одновременно удаляя форматирование самого кода. Строка в примере содержит искусственно введенное выделение отступом, чтобы продемонстрировать эту особенность.

Листинг 1.10. textwrap_dedent.py

```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text)
print('Dedented:')
print(dedented_text)
```

Результат начинает улучшаться.

```
$ python3 textwrap_dedent.py

Dedented:

The textwrap module can be used to format text for output in
situations where pretty-printing is desired. It offers
programmatic functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

Поскольку отмена отступа (*dedent*) противоположна выделению отступом (*indent*), результат представляет собой прямоугольный блок текста, в котором общие для всех строк начальные пробелы удалены. Если исходные строки имеют разные отступы от левого края, то некоторые из пробелов не будут удалены.

В этих условиях входной текст вида

```
␣Line␣one.
␣␣␣Line␣two.
␣Line␣three.
```

превратится в следующий:

```
Line␣one.
␣␣Line␣two.
Line␣three.
```

1.2.4. Совместный эффект функций `dedent` и `fill`

Текст с предварительно удаленными начальными отступами можно пропустить через функцию `fill()`, несколько изменив ширину области вывода.

Листинг 1.11. `textwrap_fill_width.py`

```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text).strip()
for width in [45, 60]:
    print('{} Columns:\n'.format(width))
    print(textwrap.fill(dedented_text, width=width))
    print()
```

Этот код выведет текст с использованием двух указанных значений ширины.

```
$ python3 textwrap_fill_width.py

45 Columns:

The textwrap module can be used to format
text for output in situations where pretty-
printing is desired. It offers programmatic
functionality similar to the paragraph
```

```
wrapping or filling features found in many
text editors.

60 Columns:

The textwrap module can be used to format text for output in
situations where pretty-printing is desired. It offers
programmatic functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

1.2.5. Декорирование блоков текста с помощью функции `indent`

Функцию `indent()` можно использовать для добавления общего префикса в начале каждой выведенной строки. В следующем примере уже знакомый вам текст форматируется так, как если бы это был текст из сообщения электронной почты, включаемый в ответное сообщение с использованием символа > в качестве префикса в каждой строке.

Листинг 1.12. `textwrap_indent.py`

```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text)
wrapped = textwrap.fill(dedented_text, width=50)
wrapped += '\n\nSecond paragraph after a blank line.'
final = textwrap.indent(wrapped, '> ')

print('Quoted block:\n')
print(final)
```

Текстовый блок разбивается на физические строки по символам перевода строки, в начало каждой строки, содержащей текст, добавляется префикс, после чего все строки объединяются в новую строку, которая возвращается в качестве результата.

```
$ python3 textwrap_indent.py

Quoted block:
> The textwrap module can be used to format text
> for output in situations where pretty-printing is
> desired. It offers programmatic functionality
> similar to the paragraph wrapping or filling
> features found in many text editors.

> Second paragraph after a blank line.
```

Дополнением строк префиксами можно управлять, передав функции `indent()` вызываемый объект `predicate` в качестве аргумента. Этот объект будет вызван поочередно для каждой строки, и префикс получат те строки, для которых он вернет истинное значение.

Листинг 1.13. `textwrap_indent_predicate.py`

```
import textwrap
from textwrap_example import sample_text

def should_indent(line):
    print('Indent {!r}?'.format(line))
    return len(line.strip()) % 2 == 0

dedented_text = textwrap.dedent(sample_text)
wrapped = textwrap.fill(dedented_text, width=50)
final = textwrap.indent(wrapped, 'EVEN ',
                        predicate=should_indent)

print('\nQuoted block:\n')
print(final)
```

В этом примере префикс EVEN добавляется в строки, содержащие четное количество символов.

```
$ python3 textwrap_indent_predicate.py

Indent ' The textwrap module can be used to format text\n'?
Indent 'for output in situations where pretty-printing is\n'?
Indent 'desired. It offers programmatic functionality\n'?
Indent 'similar to the paragraph wrapping or filling\n'?
Indent 'features found in many text editors.'?

Quoted block:

EVEN The textwrap module can be used to format text
for output in situations where pretty-printing is
EVEN desired. It offers programmatic functionality
EVEN similar to the paragraph wrapping or filling
EVEN features found in many text editors.
```

1.2.6. Висячие отступы

Точно так же, как существует возможность устанавливать ширину вывода, можно управлять и величиной отступа первой строки независимо от последующих строк.

Листинг 1.14. `textwrap_hanging_indent.py`

```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text).strip()
print(textwrap.fill(dedented_text,
                    initial_indent='',
                    subsequent_indent=' ' * 4,
                    width=50,
                    ))
```

Это делает возможным создание висячего отступа, характеризующегося тем, что первая строка смещена на меньшее расстояние от левого края, чем остальные строки.

```
$ python3 textwrap_hanging_indent.py

The textwrap module can be used to format text for
    output in situations where pretty-printing is
    desired. It offers programmatic functionality
    similar to the paragraph wrapping or filling
    features found in many text editors.
```

Значения отступов могут включать также непробельные символы. Например, висячий отступ можно снабдить префиксом в виде символа * для создания маркера.

1.2.7. Усечение длинного текста

При подготовке кратких резюме или анонсов на основе существующего теста вам может пригодиться функция `shorten()`, которая стандартизирует встречающиеся в тексте пробельные символы — символы табуляции, символы перевода строки и длинные последовательности пробелов, — заменяя их одиночными пробелами, а затем усекает текст до заданных размеров, не допуская образования неполных слов.

Листинг 1.15. `textwrap_shorten.py`

```
import textwrap
from textwrap_example import sample_text

dedented_text = textwrap.dedent(sample_text)
original = textwrap.fill(dedented_text, width=50)

print('Original:\n')
print(original)

shortened = textwrap.shorten(original, 100)
shortened_wrapped = textwrap.fill(shortened, width=50)

print('\nShortened:\n')
print(shortened_wrapped)
```

Если в той части исходного текста, которая удаляется в результате его усечения, содержатся не только пробельные символы, вместо нее можно использовать заполнитель. Предоставляемое по умолчанию значение заполнителя `[...]` можно изменить, передав его функции `shorten()` в виде аргумента `placeholder`.

```
$ python3 textwrap_shorten.py

Original:

The textwrap module can be used to format text
for output in situations where pretty-printing is
```

```
desired. It offers programmatic functionality
similar to the paragraph wrapping or filling
features found in many text editors.

Shortened:

The textwrap module can be used to format text for
output in situations where pretty-printing [...]
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `textwrap`[6].

1.3. `re`: регулярные выражения

Регулярные выражения (РВ) представляют собой образцы текста, или шаблоны, предназначенные для сопоставления с текстовыми подстроками и описываемые с помощью специального формального синтаксиса. Шаблоны РВ интерпретируются как набор инструкций, которые выполняются при предоставлении им входной строки и создают объект совпадения или измененную версию исходной строки. В профессиональной литературе вместо термина "regular expressions" часто используют его сокращенные формы "regex" или "regexp". Регулярные выражения могут включать литералы, повторения, составные шаблоны, чередования элементов (альтернативы) и другие элементы, подчиняющиеся сложному набору правил сопоставления с целевым текстом. Многие задачи, связанные с разбором текста, проще решить с помощью регулярных выражений, чем создавать специально для этого синтаксический анализатор или парсер.

Как правило, необходимость в использовании регулярных выражений возникает в приложениях, связанных с обработкой больших объемов текстовой информации. Например, они часто используются в качестве шаблонов поиска в программах обработки текстов, используемых разработчиками, включая текстовые редакторы *vi*, *emacs* и современные интегрированные средства разработки (IDE). Кроме того, они являются неотъемлемой частью таких утилит командной оболочки Unix, как *sed*, *grep* и *awk*. Многие языки программирования (Perl, Ruby, Awk и Tcl) поддерживают регулярные выражения на уровне синтаксиса языка. В других языках (например, C, C++ и Python) поддержка регулярных выражений обеспечивается библиотеками расширений.

Существуют многочисленные реализации регулярных выражений на основе открытого исходного кода, в каждой из которых используется общий базовый синтаксис, но с различными расширениями или изменениями, обеспечивающими дополнительные возможности. Синтаксис, используемый в модуле `re` в Python, основан на синтаксисе регулярных выражений, используемом в Perl, но с некоторыми усовершенствованиями, специфичными для Python.

Примечание

Несмотря на то что формальное определение термина "регулярное выражение" ограничено выражениями, описывающими регулярные языки, некоторые из расширений, под-

[6] `https://docs.python.org/3.5/library/textwrap.html`

держиваемых модулем re, выходят за рамки описания регулярных языков. В данной книге термин "регулярное выражение" используется в более общем смысле и подразумевает любое выражение, которое может быть вычислено средствами модуля re в Python.

1.3.1. Поиск образцов текста

Чаще всего регулярные выражения используются для поиска в тексте подстрок, удовлетворяющих определенным требованиям. Функция search() получает в качестве аргументов шаблон и текст, в котором должен осуществляться поиск, и возвращает объект Match, если найдено совпадение с шаблоном. Если найти соответствие шаблону не удается, то функция search() возвращает значение None.

Каждый объект Match хранит информацию о природе найденного соответствия (совпадения), в том числе исходную входную строку, использованное регулярное выражение и позицию в исходной строке, в которой встретилось вхождение данного шаблона.

Листинг 1.16. re_simple_match.py

```
import re

pattern = 'this'
text = 'Does this text match the pattern?'

match = re.search(pattern, text)

s = match.start()
e = match.end()

print('Found "{}"\nin "{}"\nfrom {} to {} ("{}")'.format(
    match.re.pattern, match.string, s, e, text[s:e]))
```

Методы start() и end() предоставляют индексы в строке, в пределах которых встретился шаблон.

```
$ python3 re_simple_match.py

Found "this"
in "Does this text match the pattern?"
from 5 to 9 ("this")
```

1.3.2. Компиляция выражений

Несмотря на то что модуль re включает функции уровня модуля, позволяющие работать с регулярными выражениями как с текстовыми строками, гораздо более эффективным способом, часто используемым программами, является компиляция регулярных выражений. Функция compile() преобразует строку регулярного выражения в объект RegexObject.

Листинг 1.17. re_simple_compiled.py

```
import re

# Предварительная компиляция шаблонов
regexes = [
    re.compile(p)
    for p in ['this', 'that']
]
text = 'Does this text match the pattern?'

print('Text: {!r}\n'.format(text))

for regex in regexes:
    print('Seeking "{}" ->'.format(regex.pattern),
        end=' ')

    if regex.search(text):
        print('match!')
    else:
        print('no match')
```

Функции уровня модуля поддерживают кеш-память для хранения скомпилированных выражений, но ее объем ограничен, в то время как непосредственное использование скомпилированных выражений позволяет избежать накладных расходов, связанных с выполнением операций поиска в кеше. Еще одним преимуществом использования предварительно скомпилированных выражений является то, что вся работа по их компиляции выполняется во время загрузки модуля, а не тогда, когда программе приходится реагировать на действия пользователя.

```
$ python3 re_simple_compiled.py

Text: 'Does this text match the pattern?'

Seeking "this" -> match!
Seeking "that" -> no match
```

1.3.3. Многократные совпадения

В приведенных до сих пор примерах использовалась функция `search()` для поиска одиночных вхождений литеральных строк. Функция `findall()` возвращает все неперекрывающиеся подстроки, совпадающие с шаблоном.

Листинг 1.18. re_findall.py

```
import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'

for match in re.findall(pattern, text):
    print('Found {!r}'.format(match))
```

Входная строка содержит два экземпляра ab.

```
$ python3 re_findall.py

Found 'ab'
Found 'ab'
```

В отличие от функции findall(), возвращающей строки, функция finditer() возвращает итератор, создающий экземпляры Match.

Листинг 1.19. re_finditer.py

```
import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'
for match in re.finditer(pattern, text):
    s = match.start()
    e = match.end()
    print('Found {!r} at {:d}:{:d}'.format(
        text[s:e], s, e))
```

Этот код находит те же два вхождения подстроки ab, и экземпляр Match предоставляет информацию о том, в каких именно позициях исходной строки они были обнаружены.

```
$ python3 re_finditer.py

Found 'ab' at 0:2
Found 'ab' at 5:7
```

1.3.4. Синтаксис шаблонов регулярных выражений

Регулярные выражения поддерживают более сложные шаблоны, чем простые литеральные текстовые строки. Шаблоны могут повторяться, привязываться к различным логическим позициям в исходной строке и даже выражаться в компактных формах, не требующих наличия в шаблоне каждого литерального символа искомой подстроки. Все эти возможности основаны на совместном использовании литеральных текстовых значений и метасимволов, являющихся частью синтаксиса регулярных выражений в модуле re.

Листинг 1.20. re_test_patterns.py

```
import re

def test_patterns(text, patterns):
    """Получив исходный текст и список шаблонов в качестве
    аргументов, выполняет поиск всех вхождений каждого шаблона
    в тексте и направляет результаты в стандартный поток вывода
    stdout.
    """
    # Поиск всех вхождений шаблона в тексте и вывод результатов
    for pattern, desc in patterns:
```

```
        print("'{}' ({})\n".format(pattern, desc))
        print(" '{}'".format(text))
        for match in re.finditer(pattern, text):
            s = match.start()
            e = match.end()
            substr = text[s:e]
            n_backslashes = text[:s].count('\\')
            prefix = '.' * (s + n_backslashes)
            print(" {}'{}'".format(prefix, substr))
        print()
    return

if __name__ == '__main__':
    test_patterns('abbaaabbbbaaaaa',
                  [('ab', "'a' followed by 'b'"),
                   ])
```

В последующих примерах функция `test_patterns()` используется для исследования того, каким образом изменение шаблона влияет на характер обнаруживаемых им совпадений в одном и том же исходном тексте. В выводимых результатах отображается входной текст и диапазон подстроки из каждой части входного текста, которая совпадает с шаблоном.

```
$ python3 re_test_patterns.py

'ab' ('a' followed by 'b')

  'abbaaabbbbaaaaa'
  'ab'
  .....'ab'
```

1.3.4.1. Повторение

Существует пять способов указания повторения шаблона. Шаблон, за которым следует метасимвол `*`, повторяется нуль или более раз (фраза "повторяется нуль раз" означает, что шаблон может вообще отсутствовать в совпадающей подстроке). Замена метасимвола `*` метасимволом `+` приводит к тому, что шаблон должен встретиться по крайней мере один раз. Использование метасимвола `?` означает, что шаблон должен встретиться нуль или один раз. Если требуется указать определенное количество повторений шаблона, используйте фигурные скобки: `{m}`, где *m* — количество повторений шаблона. Наконец, если требуется указать допустимый диапазон количества повторений, используйте запись вида `{m,n}`, где *m* — минимальное требуемое количество повторений, а *n* — максимально допустимое количество повторений. При опущенном *n* запись `{m,}` означает, что значение должно встретиться по крайней мере *m* раз подряд, тогда как максимальное количество повторений ничем не ограничено.

Листинг 1.21. re_repetition.py

```
from re_test_patterns import test_patterns

test_patterns(
```

```
    'abbaabbba',
    [('ab*', 'a followed by zero or more b'),
     ('ab+', 'a followed by one or more b'),
     ('ab?', 'a followed by zero or one b'),
     ('ab{3}', 'a followed by three b'),
     ('ab{2,3}', 'a followed by two to three b')],
)
```

В этом примере для шаблонов ab* и ab? найдено большее количество совпадений, чем для шаблона ab+.

```
$ python3 re_repetition.py

'ab*' (a followed by zero or more b)

'abbaabbba'
'abb'
...'a'
....'abbb'
........'a'

'ab+' (a followed by one or more b)

  'abbaabbba'
  'abb'
  ....'abbb'

'ab?' (a followed by zero or one b)

  'abbaabbba'
  'ab'
  ...'a'
  ....'ab'
  ........'a'

'ab{3}' (a followed by three b)

  'abbaabbba'
  ....'abbb'

'ab{2,3}' (a followed by two to three b)

  'abbaabbba'
  'abb'
  ....'abbb'
```

Обычно, обрабатывая инструкции повторения, модуль re пытается захватить как можно бо́льшую часть строки в процессе сопоставления ее с шаблоном. Этот так называемый *жадный* поиск может привести к меньшему количеству отдельных совпадений, т.е. совпадения будут включать бо́льшую часть текста, чем предполагалось. Для отмены жадного поведения следует поместить после инструкции повторения символ ?.

Листинг 1.22. re_repetition_non_greedy.py

```
from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [('ab*?', 'a followed by zero or more b'),
     ('ab+?', 'a followed by one or more b'),
     ('ab??', 'a followed by zero or one b'),
     ('ab{3}?', 'a followed by three b'),
     ('ab{2,3}?', 'a followed by two to three b')],
)
```

В случае шаблонов, разрешающих повторение буквы b нуль раз, что эквивалентно ее отсутствию, отключение жадного поведения означает, что найденные совпадения вообще не будут включать букву b.

```
$ python3 re_repetition_non_greedy.py

'ab*?' (a followed by zero or more b)

  'abbaabbba'
  'a'
  ...'a'
  ....'a'
  ........'a'

'ab+?' (a followed by one or more b)

  'abbaabbba'
  'ab'
  ....'ab'

'ab??' (a followed by zero or one b)

  'abbaabbba'
  'a'
  ...'a'
  ....'a'
  ........'a'

'ab{3}?' (a followed by three b)

  'abbaabbba'
  ....'abbb'

'ab{2,3}?' (a followed by two to three b)

  'abbaabbba'
  'abb'
  ....'abb'
```

1.3.4.2. Наборы символов

Набор символов — это группа символов, любой из которых может считаться совпадением в данной позиции шаблона. Например, набор `[ab]` совпадет с a или b.

Листинг 1.23. `re_charset.py`

```
from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [('[ab]', 'either a or b'),
     ('a[ab]+', 'a followed by 1 or more a or b'),
     ('a[ab]+?', 'a followed by 1 or more a or b, not greedy')],
)
```

Жадная форма выражения `(a[ab]+)` пытается захватить всю строку, поскольку первой буквой в ней является a, а все последующие символы — это либо буква a, либо b.

```
$ python3 re_charset.py

'[ab]' (either a or b)

  'abbaabbba'
  'a'
  .'b'
  ..'b'
  ...'a'
  ....'a'
  .....'b'
  ......'b'
  .......'b'
  ........'a'

'a[ab]+' (a followed by 1 or more a or b)

  'abbaabbba'
  'abbaabbba'

'a[ab]+?' (a followed by 1 or more a or b, not greedy)

  'abbaabbba'
  'ab'
  ...'aa'
```

Набор символов можно также использовать для исключения некоторых символов. Символ "крышка", или циркумфлекс (^), означает поиск символов, не входящих в набор символов, следующих за циркумфлексом.

Листинг 1.24. `re_charset_exclude.py`

```
from re_test_patterns import test_patterns

test_patterns(
```

```
    'This is some text -- with punctuation.',
    [('[^-. ]+', 'sequences without -, ., or space')],
)
```

Этот шаблон находит все подстроки, которые не содержат символов -, . или символа пробела.

```
$ python3 re_charset_exclude.py

'[^-. ]+' (sequences without -, ., or space)

  'This is some text -- with punctuation.'
  'This'
  .....'is'
  ........'some'
  ............'text'
  ....................'with'
  ........................'punctuation'
```

По мере разрастания символьного набора ввод каждого символа, который должен (или не должен) встречаться в совпадении, превращается в трудоемкую задачу. Для определения набора символов, числовые коды которых следуют подряд один за другим в пределах некоторого диапазона, можно использовать более компактную запись, указав начало и конец диапазона.

Листинг 1.25. re_charset_ranges.py

```
from re_test_patterns import test_patterns

test_patterns(
    'This is some text -- with punctuation.',
    [('[a-z]+', 'sequences of lowercase letters'),
     ('[A-Z]+', 'sequences of uppercase letters'),
     ('[a-zA-Z]+', 'sequences of lower- or uppercase letters'),
     ('[A-Z][a-z]+', 'one uppercase followed by lowercase')],
)
```

Здесь диапазон a-z включает все буквы нижнего регистра из таблицы кодов ASCII, тогда как диапазон A-Z включает аналогичные буквы верхнего регистра. Также допускается объединение диапазонов в единый символьный набор.

```
$ python3 re_charset_ranges.py

'[a-z]+' (sequences of lowercase letters)

  'This is some text -- with punctuation.'
  .'his'
  .....'is'
  ........'some'
  ............'text'
  ....................'with'
  ........................'punctuation'
```

```
'[A-Z]+' (sequences of uppercase letters)

  'This is some text -- with punctuation.'
  'T'

'[a-zA-Z]+' (sequences of lower- or uppercase letters)

  'This is some text -- with punctuation.'
  'This'
  .....'is'
  ........'some'
  ............'text'
  ....................'with'
.......................'punctuation'

'[A-Z][a-z]+' (one uppercase followed by lowercase)

  'This is some text -- with punctuation.'
  'This'
```

Специальным случаем набора символов является метасимвол "точка" (`.`), указывающий на то, что данный шаблон будет совпадать с любым одиночным символом, находящимся в этой позиции.

Листинг 1.26. `re_charset_dot.py`

```
from re_test_patterns import test_patterns

test_patterns(
    'abbaabbba',
    [('a.', 'a followed by any one character'),
     ('b.', 'b followed by any one character'),
     ('a.*b', 'a followed by anything, ending in b'),
     ('a.*?b', 'a followed by anything, ending in b')],
)
```

Кроме тех случаев, когда выполняется нежадный поиск, сочетание точки с повторением может приводить к получению длинных совпадений.

```
$ python3 re_charset_dot.py

'a.' (a followed by any one character)

  'abbaabbba'
  'ab'
  ...'aa'

'b.' (b followed by any one character)

  'abbaabbba'
  .'bb'
  .....'bb'
  .......'ba'
```

```
'a.*b' (a followed by anything, ending in b)

  'abbaabbba'
  'abbaabbb'

'a.*?b' (a followed by anything, ending in b)

  'abbaabbba'
  'ab'
  ...'aab'
```

1.3.4.3. Специальные символы

Еще более компактное представление обеспечивают специальные символы в виде экранированных последовательностей (Escape-кодов), позволяющие указывать предопределенные символьные наборы. Коды специальных символов, распознаваемых модулем `re`, приведены в табл. 1.1.

Таблица 1.1. Специальные символы регулярных выражений

Код	Значение
`\d`	Цифра
`\D`	Любой нецифровой символ
`\s`	Пробельный символ (табуляция, пробел, перевод строки и т.п.)
`\S`	Любой символ, не являющийся пробельным
`\w`	Алфавитно-цифровой символ
`\W`	Любой символ, не являющийся алфавитно-цифровым

Примечание

Специальные символы обозначаются префиксом в виде обратной косой черты (\), помещаемым перед символом. К сожалению, в обычных строках Python обратная косая черта сама нуждается в экранировании. Использование "сырых" строк, создаваемых посредством помещения префикса `r` перед литеральным значением, устраняет эту проблему и способствует улучшению удобочитаемости кода.

Листинг 1.27. `re_escape_codes.py`

```
from re_test_patterns import test_patterns

test_patterns(
    'A prime #1 example!',
    [(r'\d+', 'sequence of digits'),
     (r'\D+', 'sequence of non-digits'),
     (r'\s+', 'sequence of whitespace'),
     (r'\S+', 'sequence of non-whitespace'),
     (r'\w+', 'alphanumeric characters'),
     (r'\W+', 'non-alphanumeric')],
)
```

В этих образцах регулярных выражений специальные символы сочетаются с повторениями для нахождения подобных им символов во входной строке.

```
$ python3 re_escape_codes.py

'\d+' (sequence of digits)

  'A prime #1 example!'
  .........'1'

'\D+' (sequence of non-digits)
  'A prime #1 example!'
  'A prime #'
  ..........' example!'

'\s+' (sequence of whitespace)

  'A prime #1 example!'
  .' '
  .......' '
  ..........' '

'\S+' (sequence of non-whitespace)

  'A prime #1 example!'
  'A'
  ..'prime'
  ........'#1'
  ...........'example!'

'\w+' (alphanumeric characters)

  'A prime #1 example!'
  'A'
  ..'prime'
  .........'1'
  ...........'example'

'\W+' (non-alphanumeric)

  'A prime #1 example!'
  .' '
  .......' #'
  ..........' '
  ..................'!'
```

Для поиска символов, являющихся составной частью регулярного выражения, их следует экранировать в шаблоне поиска.

Листинг 1.28. `re_escape_escapes.py`

```
from re_test_patterns import test_patterns

test_patterns(
```

```
    r'\d+ \D+ \s+',
    [(r'\\.\+', 'escape code')],
)
```

В этом шаблоне экранируются символы обратной косой черты и знака "плюс", поскольку оба они являются метасимволами и имеют специальный смысл в регулярных выражениях.

```
$ python3 re_escape_escapes.py

'\\.\+' (escape code)

  '\d+ \D+ \s+'
  '\d+'
  .....'\D+'
  ..........'\s+'
```

1.3.4.4. Якорные привязки

Помимо описания содержимого шаблонов, сопоставляемых с текстом, можно указывать относительные позиции в исходном тексте, в которых должен встретиться шаблон, что достигается за счет использования привязок. Коды привязок, так называемых *якорей*, приведены в табл. 1.2.

Листинг 1.29. `re_anchoring.py`

```
from re_test_patterns import test_patterns

test_patterns(
    'This is some text -- with punctuation.',
    [(r'^\w+', 'word at start of string'),
     (r'\A\w+', 'word at start of string'),
     (r'\w+\S*$', 'word near end of string'),
     (r'\w+\S*\Z', 'word near end of string'),
     (r'\w*t\w*', 'word containing t'),
     (r'\bt\w+', 't at start of word'),
     (r'\w+t\b', 't at end of word'),
     (r'\Bt\B', 't, not start or end of word')],
)
```

Приведенные в этом примере шаблоны поиска слов в начале и в конце строки различаются своей структурой, поскольку за словом, расположенным в конце строки, следует знак препинания, завершающий предложение. Шаблон `\w+$` не дал бы совпадения, поскольку точка (`.`) не относится к алфавитно-цифровым символам.

```
$ python3 re_anchoring.py

'^\w+' (word at start of string)

  'This is some text -- with punctuation.'
  'This'
```

```
'\A\w+' (word at start of string)

  'This is some text -- with punctuation.'
  'This'

'\w+\S*$' (word near end of string)

  'This is some text -- with punctuation.'
  ........................'punctuation.'

'\w+\S*\Z' (word near end of string)

  'This is some text -- with punctuation.'
  ........................'punctuation.'

'\w*t\w*' (word containing t)

  'This is some text -- with punctuation.'
  ............'text'
  ..................'with'
  ........................'punctuation'

'\bt\w+' (t at start of word)

  'This is some text -- with punctuation.'
  ............'text'

'\w+t\b' (t at end of word)

  'This is some text -- with punctuation.'
  ............'text'

'\Bt\B' (t, not start or end of word)

  'This is some text -- with punctuation.'
  ......................'t'
  ............................'t'
  ..............................'t'
```

Таблица 1.2. Коды якорных привязок в регулярных выражениях

Код	Значение
^	Начало логической или физической строки
$	Конец логической или физической строки
\A	Начало логической строки
\Z	Конец логической строки
\b	Пустая строка в начале или в конце слова
\B	Пустая строка, расположенная не в начале или в конце слова

1.3.5. Ограничение зоны поиска

В ситуациях, когда заранее известно, что будет осуществляться только поиск подстроки полной исходной строки, на условия сопоставления шаблона с текстом можно наложить дополнительные ограничения, указав модулю `re` границы диапазона поиска. Например, если шаблон встречается в начале входной строки, то использование функции `match()` вместо функции `search()` автоматически привяжет поиск к началу строки без явного включения соответствующего якоря в шаблон поиска.

Листинг 1.30. `re_match.py`

```
import re

text = 'This is some text -- with punctuation.'
pattern = 'is'

print('Text :', text)
print('Pattern:', pattern)

m = re.match(pattern, text)
print('Match :', m)
s = re.search(pattern, text)
print('Search :', s)
```

Поскольку литеральный текст `is` не встречается в начале входного текста, он не будет найден функцией `match()`. Однако эта последовательность символов встречается в тексте дважды и потому будет найдена функцией `search()`.

```
$ python3 re_match.py

Text : This is some text -- with punctuation.
Pattern: is
Match : None
Search : <_sre.SRE_Match object; span=(2, 4), match='is'>
```

Функция `fullmatch()` требует, чтобы вся входная строка совпала с шаблоном.

Листинг 1.31. `re_fullmatch.py`

```
import re

text = 'This is some text -- with punctuation.'
pattern = 'is'

print('Text :', text)
print('Pattern :', pattern)

m = re.search(pattern, text)
print('Search :', m)

s = re.fullmatch(pattern, text)
print('Full match :', s)
```

В данном случае функция search() подтверждает, что шаблон действительно встречается во входной строке, но вся строка им не исчерпывается, и поэтому функция fullmatch() не обнаруживает совпадения.

```
$ python3 re_fullmatch.py

Text       : This is some text -- with punctuation.
Pattern    : is
Search     : <_sre.SRE_Match object; span=(2, 4), match='is'>
Full match : None
```

Метод search() скомпилированного регулярного выражения поддерживает необязательные позиционные параметры start и end, которые позволяют ограничить поиск подстрокой полной строки.

Листинг 1.32. re_search_substring.py

```
import re

text = 'This is some text -- with punctuation.'
pattern = re.compile(r'\b\w*is\w*\b')

print('Text:', text)
print()

pos = 0
while True:

    match = pattern.search(text, pos)
    if not match:
        break
    s = match.start()
    e = match.end()
    print(' {:>2d} : {:>2d} = "{}"'.format(
        s, e - 1, text[s:e]))
    # Переместиться вперед по строке text для продолжения поиска
    pos = e
```

Этот пример реализует менее эффективную форму iterall(). Каждый раз, когда обнаруживается совпадение, его конечная позиция используется в качестве начальной для продолжения поиска.

```
$ python3 re_search_substring.py

Text: This is some text -- with punctuation.

    0 : 3 = "This"
    5 : 6 = "is"
```

1.3.6. Отделение совпадений с помощью групп

Поиск совпадений с шаблоном служит основой для более мощных регулярных выражений. Добавление групп в шаблон изолирует отдельные составляющие совпавшего текста, открывая дорогу для создания парсеров (программ для разбора текста). Группы определяются посредством заключения шаблонов в круглые скобки.

Листинг 1.33. re_groups.py

```
from re_test_patterns import test_patterns

test_patterns(
    'abbaaabbbbaaaaa',
    [('a(ab)', 'a followed by literal ab'),
     ('a(a*b*)', 'a followed by 0-n a and 0-n b'),
     ('a(ab)*', 'a followed by 0-n ab'),
     ('a(ab)+', 'a followed by 1-n ab')],
)
```

Любое завершенное регулярное выражение может быть преобразовано в группу и вложено в более крупное выражение. Для повторения группового шаблона как целого к нему можно применить любой из спецификаторов числа повторений.

```
$ python3 re_groups.py

'a(ab)' (a followed by literal ab)

  'abbaaabbbbaaaaa'
  ....'aab'

'a(a*b*)' (a followed by 0-n a and 0-n b)

  'abbaaabbbbaaaaa'
  'abb'
  ...'aaabbbb'
  ..........'aaaaa'

'a(ab)*' (a followed by 0-n ab)

  'abbaaabbbbaaaaa'
  'a'
  ...'a'
  ....'aab'
  ..........'a'
  ...........'a'
  ............'a'
  .............'a'
  ..............'a'

'a(ab)+' (a followed by 1-n ab)

  'abbaaabbbbaaaaa'
  ....'aab'
```

Чтобы получить доступ ко всем подстрокам, совпавшим с отдельными группами шаблона, используйте метод `groups()` объекта `Match`.

Листинг 1.34. `re_groups_match.py`

```
import re

text = 'This is some text -- with punctuation.'

print(text)
print()

patterns = [
    (r'^(\w+)', 'word at start of string'),
    (r'(\w+)\S*$', 'word at end, with optional punctuation'),
    (r'(\bt\w+)\W+(\w+)', 'word starting with t, another word'),
    (r'(\w+t)\b', 'word ending with t'),
]

for pattern, desc in patterns:
    regex = re.compile(pattern)
    match = regex.search(text)
    print("'{}' ({})\n".format(pattern, desc))
    print('  ', match.groups())
    print()
```

Вызов `Match.groups()` возвращает последовательность строк в том порядке, в котором группы встречаются в выражении, сопоставляемом со строкой.

```
$ python3 re_groups_match.py

This is some text -- with punctuation.

'^(\w+)' (word at start of string)

   ('This',)

'(\w+)\S*$' (word at end, with optional punctuation)

   ('punctuation',)

'(\bt\w+)\W+(\w+)' (word starting with t, another word)

   ('text', 'with')

'(\w+t)\b' (word ending with t)

   ('text',)
```

Чтобы получить доступ к подстроке, совпавшей с какой-то одной группой, используйте метод `group()`. Это может пригодиться, если группирование используется для нахождения отдельных частей строки, но некоторые из частей, совпадающих с группами, не нужны в конечных результатах.

Листинг 1.35. re_groups_individual.py

```
import re

text = 'This is some text -- with punctuation.'

print('Input text :', text)

# Слово, начинающееся с 't', за которым следует другое слово
regex = re.compile(r'(\bt\w+)\W+(\w+)')
print('Pattern :', regex.pattern)

match = regex.search(text)
print('Entire match :', match.group(0))
print('Word starting with "t":', match.group(1))
print('Word after "t" word :', match.group(2))
```

Группа 0 представляет строку, совпавшую со всем выражением, тогда как подгруппы нумеруются начиная с 1 в порядке появления их открывающих скобок в выражении.

```
$ python3 re_groups_individual.py

Input text : This is some text -- with punctuation.
Pattern : (\bt\w+)\W+(\w+)
Entire match : text -- with
Word starting with "t": text
Word after "t" word : with
```

Python расширяет базовый синтаксис группирования, разрешая использовать *именованные группы*. Обращение к группам по именам упрощает внесение изменений в шаблон по прошествии некоторого времени, избавляя от необходимости изменять также код, использующий результаты поиска совпадений. Чтобы присвоить имя группе, используйте синтаксис `(?P<имя>шаблон)`.

Листинг 1.36. re_groups_named.py

```
import re

text = 'This is some text -- with punctuation.'

print(text)

print()

patterns = [
    r'^(?P<first_word>\w+)',
    r'(?P<last_word>\w+)\S*$',
    r'(?P<t_word>\bt\w+)\W+(?P<other_word>\w+)',
    r'(?P<ends_with_t>\w+t)\b',
]

for pattern in patterns:
    regex = re.compile(pattern)
```

```
    match = regex.search(text)
    print("'{}'".format(pattern))
    print(' ', match.groups())
    print(' ', match.groupdict())
    print()
```

Метод `groupdict()` используется для получения словаря, который сопоставляет имена групп с подстроками совпадений, хранящимися в объекте `match`. В упорядоченную последовательность, возвращаемую методом `groups()`, включаются также именованные шаблоны.

```
$ python3 re_groups_named.py

This is some text -- with punctuation.

'^(?P<first_word>\w+)'
   ('This',)
   {'first_word': 'This'}

'(?P<last_word>\w+)\S*$'
   ('punctuation',)
   {'last_word': 'punctuation'}

'(?P<t_word>\bt\w+)\W+(?P<other_word>\w+)'
   ('text', 'with')
   {'t_word': 'text', 'other_word': 'with'}

'(?P<ends_with_t>\w+t)\b'
   ('text',)
   {'ends_with_t': 'text'}
```

Обновленная версия функции `test_patterns()`, отображающая нумерованные и именованные группы, совпавшие с шаблоном, упростит понимание результатов, получаемых в последующих примерах.

Листинг 1.37. `re_test_patterns_groups.py`

```
import re

def test_patterns(text, patterns):
"""Получив исходный текст и список шаблонов в качестве
аргументов, выполняет поиск всех вхождений каждого шаблона
в тексте и направляет результаты в стандартный поток вывода
stdout.
"""
# Поиск всех вхождений шаблона в тексте и вывод результатов
for pattern, desc in patterns:
    print('{!r} ({})\n'.format(pattern, desc))
    print(' {!r}'.format(text))
    for match in re.finditer(pattern, text):
        s = match.start()
        e = match.end()
        prefix = ' ' * (s)
        print(
```

```
            ' {}{!r}{} '.format(prefix,
                                text[s:e],
                                ' ' * (len(text) - e)),
            end=' ',
        )
        print(match.groups())
        if match.groupdict():
            print('{}{}'.format(
                ' ' * (len(text) - s),
                match.groupdict()),
            )
    print()
return
```

Поскольку группа сама по себе является законченным регулярным выражением, возможно вложение одних групп в другие для создания еще более сложных выражений.

Листинг 1.38. re_groups_nested.py

```
from re_test_patterns_groups import test_patterns

test_patterns(
    'abbaabbba',
    [(r'a((a*)(b*))', 'a followed by 0-n a and 0-n b')],
)
```

В данном случае группа (a*) совпадает с пустой строкой, и таким образом значение, возвращаемое методом groups(), включает эту пустую строку в качестве совпадающего значения.

```
$ python3 re_groups_nested.py

'a((a*)(b*))' (a followed by 0-n a and 0-n b)

  'abbaabbba'
  'abb'        ('bb', '', 'bb')
    'aabbb'    ('abbb', 'a', 'bbb')
       'a'     ('', '', '')
```

Группы можно также использовать для определения альтернативных шаблонов. Символ канала (|) указывает на возможность выбора одного из нескольких альтернативных вариантов шаблона при поиске совпадений. Однако работа с каналами требует внимательности. В следующем примере первое выражение совпадает с буквой a, за которой идет последовательность, состоящая только из букв a или только из букв b. Второй шаблон совпадает с буквой a, за которой идет последовательность, включающая буквы a и b в любых сочетаниях. Шаблоны очень похожи, но результирующие совпадения разительно отличаются.

Листинг 1.39. re_groups_alternative.py

```
from re_test_patterns_groups import test_patterns

test_patterns(
```

```
    'abbaabbba',
    [(r'a((a+)|(b+))', 'a then seq. of a or seq. of b'),
     (r'a((a|b)+)', 'a then seq. of [ab]')],
)
```

Если для какой-либо альтернативной группы не существует соответствия, но оно существует для всего шаблона, то возвращаемое методом `groups()` значение включает значение `None` в той точке последовательности, в которой должна была бы появиться данная альтернативная группа.

```
$ python3 re_groups_alternative.py

'a((a+)|(b+))' (a then seq. of a or seq. of b)

  'abbaabbba'
  'abb'        ('bb', None, 'bb')
    'aa'       ('a', 'a', None)

'a((a|b)+)' (a then seq. of [ab])

  'abbaabbba'
  'abbaabbba'  ('bbaabbba', 'a')
```

Определение группы, содержащей подшаблон, может быть полезным также в тех случаях, когда строка, совпадающая с подшаблоном, не является частью того содержимого, которое должно извлекаться из полного текста. Группы такого рода называются *незахватывающими*. Незахватывающие группы можно использовать для описания повторяющихся шаблонов или альтернатив без обособления совпадающего фрагмента строки в возвращаемом значении. Незахватывающая группа описывается с помощью синтаксиса `(?:`*шаблон*`)`.

Листинг 1.40. re_groups_noncapturing.py

```
from re_test_patterns_groups import test_patterns

test_patterns(
    'abbaabbba',
    [(r'a((a+)|(b+))', 'capturing form'),
     (r'a((?:a+)|(?:b+))', 'noncapturing')],
)
```

Сравните в следующем примере группы, которые возвращаются для захватывающей и незахватывающей форм шаблона, дающих одни и те же результирующие совпадения.

```
$ python3 re_groups_noncapturing.py

'a((a+)|(b+))' (capturing form)

  'abbaabbba'
  'abb'        ('bb', None, 'bb')
    'aa'       ('a', 'a', None)

'a((?:a+)|(?:b+))' (noncapturing)
```

```
  'abbaabbba'
  'abb'         ('bb',)
    'aa'        ('a',)
```

1.3.7. Опции поиска

Правила сопоставления шаблона с текстом, используемые движком регулярных выражений, можно изменить с помощью флагов опций. Эти флаги можно объединять с помощью побитовой операции ИЛИ и передавать функциям `compile()`, `search()`, `match()`, а также другим функциям, получающим шаблон для выполнения поиска.

1.3.7.1. Поиск, нечувствительный к регистру

При установленном флаге `IGNORECASE` литеральные символы и наборы символов, указанные в шаблоне, сопоставляются с соответствующими символами в тексте, игнорируя их регистр.

Листинг 1.41. `re_flags_ignorecase.py`

```
import re

text = 'This is some text -- with punctuation.'
pattern = r'\bT\w+'
with_case = re.compile(pattern)
without_case = re.compile(pattern, re.IGNORECASE)

print('Text:\n  {!r}'.format(text))
print('Pattern:\n {}'.format(pattern))
print('Case-sensitive:')
for match in with_case.findall(text):
    print('  {!r}'.format(match))
print('Case-insensitive:')
for match in without_case.findall(text):
    print('  {!r}'.format(match))
```

Данный шаблон включает литерал `T`, и если флаг `IGNORECASE` не установлен, то единственным совпадением будет слово `This`. Если игнорировать регистр, то совпадет также слово `text`.

```
$ python3 re_flags_ignorecase.py

Text:
  'This is some text -- with punctuation.'
Pattern:
  \bT\w+
Case-sensitive:
  'This'
Case-insensitive:
  'This'
  'text'
```

1.3.7.2. Многострочные входные строки

На характер выполнения поиска в случае многострочных входных строк влияют два флага: MULTILINE и DOTALL. Флаг MULTILINE управляет обработкой якорных привязок для текста, содержащего символы перевода строки. Если включен многострочный режим, то якорные привязки ^ и $ означают начало и конец не только всей логической строки, но и каждой из физических строк.

Листинг 1.42. re_flags_multiline.py

```
import re

text = 'This is some text -- with punctuation.\nA second line.'
pattern = r'(^\w+)|(\w+\S*$)'
single_line = re.compile(pattern)
multiline = re.compile(pattern, re.MULTILINE)

print('Text:\n  {!r}'.format(text))
print('Pattern:\n  {}'.format(pattern))
print('Single Line :')
for match in single_line.findall(text):
    print('  {!r}'.format(match))
print('Multline    :')
for match in multiline.findall(text):
    print('  {!r}'.format(match))
```

В этом примере шаблон совпадает с первым или последним словом входной строки. Он совпадает с подстрокой line. в конце всей строки даже в отсутствие символа перевода строки.

```
$ python3 re_flags_multiline.py

Text:
  'This is some text -- with punctuation.\nA second line.'
Pattern:
  (^\w+)|(\w+\S*$)
Single Line :
  ('This', '')
  ('', 'line.')
Multline :
  ('This', '')
  ('', 'punctuation.')
  ('A', '')
  ('', 'line.')
```

Еще одним флагом, влияющим на обработку многострочного текста, является флаг DOTALL. Обычно точке (.) соответствует любой одиночный символ во входном тексте, кроме символа перевода строки. Данный флаг разрешает совпадение точки и с этим символом.

Листинг 1.43. re_flags_dotall.py

```
import re

text = 'This is some text -- with punctuation.\nA second line.'
```

```
pattern = r'.+'
no_newlines = re.compile(pattern)
dotall = re.compile(pattern, re.DOTALL)

print('Text:\n  {!r}'.format(text))
print('Pattern:\n  {}'.format(pattern))
print('No newlines :')
for match in no_newlines.findall(text):
    print('  {!r}'.format(match))
print('Dotall      :')
for match in dotall.findall(text):
    print('  {!r}'.format(match))
```

Без установки этого флага шаблону соответствовала бы каждая физическая строка входного текста. Добавление флага приводит к тому, что захватывается вся строка.

```
$ python3 re_flags_dotall.py

Text:
  'This is some text -- with punctuation.\nA second line.'
Pattern:
  .+
No newlines :
  'This is some text -- with punctuation.'
  'A second line.'
Dotall :
  'This is some text -- with punctuation.\nA second line.'
```

1.3.7.3. Unicode

В Python 3 объекты `str` используют полный набор символов Unicode, и при обработке механизмом регулярных выражений предполагается, что именно этот набор задействован в шаблоне и исходном тексте. Описанные ранее Escape-последовательности по умолчанию также определяются в терминах Unicode. Эти предположения, в частности, означают, что шаблон `\w+` совпадет как со словом "French", так и со словом "Français". Чтобы ограничить Escape-последовательности символьным набором ASCII, как это принято по умолчанию в Python 2, при компиляции шаблона или вызове таких функций уровня модуля, как `search()` и `match()`, следует использовать флаг `ASCII`.

Листинг 1.44. `re_flags_ascii.py`

```
import re

text = u'Français łzoty Österreich'
pattern = r'\w+'
ascii_pattern = re.compile(pattern, re.ASCII)
unicode_pattern = re.compile(pattern)

print('Text    :', text)
```

```
print('Pattern :', pattern)
print('ASCII   :', list(ascii_pattern.findall(text)))
print('Unicode :', list(unicode_pattern.findall(text)))
```

Другие Escape-последовательности (\W, \b, \B, \d, \D, \s и \S) также обрабатываются по-другому в случае ASCII-текста. Вместо того чтобы каждый раз обращаться к базе данных Unicode для выяснения свойств каждого символа, модуль re использует ASCII-определение набора символов, идентифицируемого Escape-последовательностью.

```
$ python3 re_flags_ascii.py

Text    : Français łzoty Österreich
Pattern : \w+
ASCII   : ['Fran', 'ais', 'z', 'oty', 'sterreich']
Unicode : ['Français', 'łzoty', 'Österreich']
```

1.3.7.4. Синтаксис описательных регулярных выражений

Компактный формат синтаксиса регулярных выражений может стать препятствием по мере их усложнения. С ростом количества групп в выражении вам будет требоваться все больше времени для того, чтобы понять роль каждого элемента и разобраться в том, каким образом взаимодействуют различные части выражения. Использование именованных групп немного сглаживает подобные проблемы, но лучшим решением является использование *описательного (многословного) режима* (verbose mode) регулярных выражений, который разрешает включение в шаблон комментариев и дополнительных пробелов, позволяющих подробно описать используемое выражение. Продемонстрируем это на примере шаблона для проверки корректности адресов электронной почты. Его первая версия распознает адреса, заканчивающиеся одним из трех доменов верхнего уровня: .com, .org и .edu.

Листинг 1.45. re_email_compact.py

```
import re

address = re.compile('[\w\d.+-]+@([\w\d.]+\.)+(com|org|edu)')

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com',
    u'valid-address@mail.example.com',
    u'not-valid@example.foo',
]

for candidate in candidates:
    match = address.search(candidate)
    print('{:<30} {}'.format(
        candidate, 'Matches' if match else 'No match')
)
```

Это выражение уже является довольно сложным. Оно содержит выражения классов, групп и повторений.

```
$ python3 re_email_compact.py

first.last@example.com Matches
first.last+category@gmail.com Matches
valid-address@mail.example.com Matches
not-valid@example.foo No match
```

Преобразование этого выражения в описательный формат упростит последующее расширение шаблона.

Листинг 1.46. re_email_verbose.py

```
import re

address = re.compile(
    '''
    [\w\d.+-]+ # Имя пользователя
    @
    ([\w\d.]+\.)+ # Префикс имени домена
    (com|org|edu) # TODO: поддержка других доменов верхнего уровня
    ''',
    re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com',
    u'valid-address@mail.example.com',
    u'not-valid@example.foo',
]

for candidate in candidates:
    match = address.search(candidate)
    print('{:<30}  {}'.format(
        candidate, 'Matches' if match else 'No match'),
)
```

Это выражение совпадает с теми же входными данными, но использование описательного формата сделало его более понятным. Комментарии помогают выделить отдельные части выражения, поэтому его нетрудно расширить для сопоставления с дополнительными элементами входной информации.

```
$ python3 re_email_verbose.py

first.last@example.com         Matches
first.last+category@gmail.com  Matches
valid-address@mail.example.com Matches
not-valid@example.foo          No match
```

Расширенная версия разбирает входную строку. которая включает имя пользователя и электронный адрес в том виде, в каком они встречаются в заголов-

ке электронной почты. Сначала идет имя пользователя, а за ним следует адрес электронной почты, заключенный в угловые скобки (< и >).

Листинг 1.47. re_email_with_name.py

```
import re

address = re.compile(
    '''

    # Имя состоит из букв и может включать символы точки "."
    # в сокращенных вариантах обращения и инициалах
    ((?P<name>
       ([\w.,]+\s+)*[\w.,]+)
       \s*
       # Адреса электронной почты заключаются в угловые скобки
       # < >, но только если найдено имя, поэтому открывающая
       # угловая скобка включена в эту группу
       <
    )? # Полное имя является необязательным элементом

    # Собственно электронный адрес: username@domain.tld
    (?P<email>
      [\w\d.+-]+ # Имя пользователя
      @
      ([\w\d.]+\.)+ # Префикс имени домена
      (com|org|edu) # Ограничение списка доменов верхнего уровня
    )

    >? # Необязательная закрывающая угловая скобка
    ''',
    re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'first.last+category@gmail.com',
    u'valid-address@mail.example.com',
    u'not-valid@example.foo',
    u'First Last <first.last@example.com>',
    u'No Brackets first.last@example.com',
    u'First Last',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
    u'<first.last@example.com>',
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print(' Name :', match.groupdict()['name'])
        print(' Email:', match.groupdict()['email'])
```

```
    else:
    print(' No match')
```

Как и в других языках программирования, возможность включения комментариев в регулярные выражения облегчает их сопровождение. Окончательная версия включает примечания, предназначенные для тех, кому впоследствии придется сопровождать программу, и пробельные символы, отделяющие группы одну от другой и выделяющие их уровни вложения.

```
$ python3 re_email_with_name.py

Candidate: first.last@example.com
  Name : None
  Email: first.last@example.com
Candidate: first.last+category@gmail.com
  Name : None
  Email: first.last+category@gmail.com
Candidate: valid-address@mail.example.com
  Name : None
  Email: valid-address@mail.example.com
Candidate: not-valid@example.foo
No match
Candidate: First Last <first.last@example.com>
  Name : First Last
  Email: first.last@example.com
Candidate: No Brackets first.last@example.com
  Name : None
  Email: first.last@example.com
Candidate: First Last
  No match
Candidate: First Middle Last <first.last@example.com>
  Name : First Middle Last
  Email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
  Name : First M. Last
  Email: first.last@example.com
Candidate: <first.last@example.com>
  Name : None
  Email: first.last@example.com
```

1.3.7.5. Включение флагов в шаблоны

В тех ситуациях, когда флаги не могут быть добавлены во время компиляции выражения (например, когда шаблон передается в качестве аргумента библиотечной функции, которая будет компилировать его позднее), их можно включить в саму строку выражения. Например, чтобы включить режим поиска, нечувствительного к регистру, в начало выражения следует добавить группу `(?i)`.

Листинг 1.48. `re_flags_embedded.py`

```
import re

text = 'This is some text -- with punctuation.'
```

```
pattern = r'(?i)\bT\w+'
regex = re.compile(pattern)

print('Text      :', text)
print('Pattern   :', pattern)
print('Matches   :', regex.findall(text))
```

Поскольку флаги управляют вычислением или разбором всего выражения, они всегда должны указываться в начале выражения.

```
$ python3 re_flags_embedded.py

Text      : This is some text -- with punctuation.
Pattern   : (?i)\bT\w+
Matches   : ['This', 'text']
```

Сокращенные обозначения всех флагов приведены в табл. 1.3.

Флаги можно объединять путем помещения их в одну группу. Например, группа `(?im)` включает нечувствительный к регистру поиск для многострочных строк.

Таблица 1.3. Сокращенные обозначения флагов регулярных выражений

Флаг	Сокращение
ASCII	a
IGNORECASE	i
MULTILINE	m
DOTALL	s
VERBOSE	x

1.3.8. Просмотр вперед или назад

Во многих случаях полезно иметь возможность находить соответствия одной части шаблона только при условии, что одновременно удается найти соответствие другой его части. Например, в выражении для разбора адресов электронной почты угловые скобки были обозначены как необязательные. В реальных ситуациях угловые скобки должны встречаться парами, и совпадение должно быть возможным только при наличии или отсутствии одновременно обеих скобок. В модифицированной версии выражения для поиска совпадений с парой угловых скобок используется *положительный просмотр вперед* (другое название — *положительная опережающая проверка*). Для этого вида просмотра используется синтаксис `(?=`*шаблон*`)`.

Листинг 1.49. `re_look_ahead.py`

```
import re

address = re.compile(
    '''
    # Имя состоит из букв и может включать символы точки "."
    # в сокращенных вариантах обращения и инициалах
    ((?P<name>
       ([\w.,]+\s+)*[\w.,]+
```

```
      )
      \s+
    ) # Имя уже не является необязательным

    # ПРОСМОТР ВПЕРЕД
    # Адреса электронной почты заключены в угловые скобки, но
    # только в том случае, если имеются обе скобки
    # или ни одной
    (?= (<.*>$)       # Остаток заключен в угловые скобки
        |
        ([^<].*[^>]$) # Остаток *не* заключен в угловые скобки
      )

    <? # Необязательная открывающая угловая скобка

    # Собственно электронный адрес: username@domain.tld
    (?P<email>
      [\w\d.+-]+ # Имя пользователя
      @
      ([\w\d.]+\.)+ # Префикс имени домена
      (com|org|edu) # Ограничение списка доменов верхнего уровня
    )

    >? # Необязательная закрывающая угловая скобка
    ''',
    re.VERBOSE)

candidates = [
    u'First Last <first.last@example.com>',
    u'No Brackets first.last@example.com',
    u'Open Bracket <first.last@example.com',
    u'Close Bracket first.last@example.com>',
]
for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print(' Name :', match.groupdict()['name'])
        print(' Email:', match.groupdict()['email'])
    else:
        print(' No match')
```

В эту версию выражения внесено несколько существенных изменений. Во-первых, имя уже не является необязательным. Это не только означает, что один лишь адрес без имени не может образовывать совпадение, но и предотвращает нахождение некорректно отформатированных комбинаций “имя — адрес”. Правило положительного просмотра вперед, вставленное после группы `name`, позволяет удостовериться в том, что либо оставшаяся часть строки заключена в угловые скобки, либо отсутствуют непарные угловые скобки (т.е. угловые скобки должны быть либо в виде пары, либо отсутствовать). Правило просмотра вперед оформлено в виде группы, но совпадение с этой группой не приводит к перемещению текущей позиции вперед во входном тексте, поэтому остальная часть шаблона начинает обработку текста с той же позиции.

```
$ python3 re_look_ahead.py

Candidate: First Last <first.last@example.com>
  Name : First Last
  Email: first.last@example.com
Candidate: No Brackets first.last@example.com
  Name : No Brackets
  Email: first.last@example.com
Candidate: Open Bracket <first.last@example.com
  No match
Candidate: Close Bracket first.last@example.com>
  No match
```

Просмотр вперед с отрицанием (другое название — *отрицательная опережающая проверка*), для которого используется синтаксис `(?!шаблон)`, не должен совпадать с текстом за текущей позицией. Например, шаблон, распознающий электронные адреса, можно видоизменить таким образом, чтобы он игнорировал адреса, начинающиеся с `noreply`, которые обычно используются системами автоматической рассылки электронных сообщений.

Листинг 1.50. `re_negative_look_ahead.py`

```
import re

address = re.compile(
    '''
    ^

    # Адрес: username@domain.tld

    # Игнорировать адреса noreply
    (?!noreply@.*$)
    [\w\d.+-]+ # Имя пользователя
    @
    ([\w\d.]+\.)+ # Префикс имени домена
    (com|org|edu) # Ограничение списка доменов верхнего уровня

    $
    ''',
    re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'noreply@example.com',
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print(' Match:', candidate[match.start():match.end()])
    else:
        print(' No match')
```

Адреса, начинающиеся с noreply, не совпадают с шаблоном, поскольку не выполняется условие опережающей проверки.

```
$ python3 re_negative_look_ahead.py

Candidate: first.last@example.com
  Match: first.last@example.com
Candidate: noreply@example.com
  No match
```

Вместо выполнения опережающей проверки наличия строки noreply в части адреса, содержащей имя пользователя, можно было бы переписать шаблон таким образом, чтобы после сопоставления с именем пользователя выполнялся *просмотр назад с отрицанием* (другое название — *отрицательная ретроспективная проверка*), для которого используется синтаксис (?<!*шаблон*).

Листинг 1.51. re_negative_look_behind.py

```
import re

address = re.compile(
    '''
    ^

    # Адрес: username@domain.tld

    [\w\d.+-]+ # Имя пользователя

    # Игнорировать адреса noreply

    (?<!noreply)

    @
    ([\w\d.]+\.)+ # Префикс имени домена
    (com|org|edu) # Ограничение списка доменов верхнего уровня

    $
    ''',
    re.VERBOSE)

candidates = [
    u'first.last@example.com',
    u'noreply@example.com',
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print(' Match:', candidate[match.start():match.end()])
    else:
        print(' No match')
```

Просмотр назад работает несколько иначе, чем просмотр вперед, в том смысле, что выражение должно использовать шаблон фиксированной длины. Повторения допускаются, коль скоро задано их фиксированное количество (никаких символов-заполнителей или диапазонов).

```
$ python3 re_negative_look_behind.py

Candidate: first.last@example.com
  Match: first.last@example.com
Candidate: noreply@example.com
  No match
```

Положительный просмотр назад можно использовать для поиска текста, следующего за шаблоном, используя синтаксис `(?<=шаблон)`. В следующем примере выражение находит идентификаторы пользователей Твиттера.

Листинг 1.52. re_look_behind.py

```
import re

twitter = re.compile(
    '''
    # Идентификатор пользователя в Твиттере: @username
    (?<=@)
    ([\w\d_]+) # Имя пользователя
    ''',
    re.VERBOSE)

text = '''This text includes two Twitter handles.
One for @ThePSF, and one for the author, @doughellmann.
'''

print(text)
for match in twitter.findall(text):
    print('Handle:', match)
```

Шаблон соответствует последовательностям символов, которые могут быть идентификаторами пользователей в Твиттере, если им предшествует символ `@`.

```
$ python3 re_look_behind.py

This text includes two Twitter handles.
One for @ThePSF, and one for the author, @doughellmann.

Handle: ThePSF
Handle: doughellmann
```

1.3.9. Обратные ссылки

Найденные совпадения с шаблоном можно использовать в последующих частях выражения. Так, пример с адресами электронной почты можно обновить таким образом, чтобы выполнялся поиск только тех адресов, которые состоят из

имени и фамилии пользователя, используя обратные ссылки на соответствующие группы. Проще всего это можно сделать, ссылаясь на ранее найденное совпадение по номеру с помощью синтаксиса *номер*.

Листинг 1.53. re_refer_to_group.py

```
import re

address = re.compile(
    r'''

    # ФИО в обычной форме
    (\w+)               # Имя
    \s+
    (([\w.]+)\s+)?      # Необязательное отчество или инициалы
    (\w+)               # Фамилия

    \s+

    <

    # Адрес: имя.фамилия@domain.tld
    (?P<email>
      \1                # Имя
      \.
      \4                # Фамилия
      @
      ([\w\d.]+\.)+     # Префикс имени домена
      (com|org|edu)     # Ограничение доменов верхнего уровня
    )

    >
    ''',
    re.VERBOSE | re.IGNORECASE)

candidates = [
    u'First Last <first.last@example.com>',
    u'Different Name <first.last@example.com>',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print(' Match name :', match.group(1), match.group(4))
        print(' Match email:', match.group(5))
    else:
        print(' No match')
```

Несмотря на простоту синтаксиса, создание нумерованных обратных ссылок имеет некоторые недостатки. Если посмотреть на это с практической точки зре-

ния, то изменение выражения может изменить нумерацию групп, и в этом случае вам придется каждый раз обновлять ссылки. Еще одним недостатком является то, что стандартный синтаксис \n допускает использование не более 99 ссылок, поскольку любое число, включающее три цифры, будет интерпретироваться как восьмеричный код символа, а не как обратная ссылка. Разумеется, если в выражении содержится более 99 групп, то это говорит о наличии более серьезных проблем с сопровождением кода, чем невозможность сослаться на каждую из таких групп.

```
$ python3 re_refer_to_group.py

Candidate: First Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: Different Name <first.last@example.com>
  No match
Candidate: First Middle Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
```

Средства Python, предназначенные для разбора регулярных выражений, включают расширение, использующее ссылки вида (?P=*имя*) для обращения к значениям именованных групп, для которых ранее были найдены совпадения.

Листинг 1.54. re_refer_to_named_group.py

```
import re

address = re.compile(
    '''

    # ФИО в обычной форме
    (?P<first_name>\w+)
    \s+
    (([\w.]+)\s+)? # Необязательное отчество или инициалы
    (?P<last_name>\w+)

    \s+

    <

    # Адрес: имя.фамилия@domain.tld
    (?P<email>
      (?P=first_name)
      \.
      (?P=last_name)
      @
      ([\w\d.]+\.)+ # Префикс домена верхнего уровня
      (com|org|edu) # Ограничение доменов верхнего уровня
    )
```

```
    >
    ''',
    re.VERBOSE | re.IGNORECASE)

candidates = [
    u'First Last <first.last@example.com>',
    u'Different Name <first.last@example.com>',
    u'First Middle Last <first.last@example.com>',
    u'First M. Last <first.last@example.com>',
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print(' Match name :', match.groupdict()['first_name'],
              end=' ')
        print(match.groupdict()['last_name'])
        print(' Match email:', match.groupdict()['email'])
    else:
        print(' No match')
```

Выражение для проверки адресов компилируется с установленным флагом IGNORECASE, поскольку имена собственные обычно начинаются с прописной буквы, тогда как в адресах электронной почты используется нижний регистр.

```
$ python3 re_refer_to_named_group.py

Candidate: First Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: Different Name <first.last@example.com>
  No match
Candidate: First Middle Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
Candidate: First M. Last <first.last@example.com>
  Match name : First Last
  Match email: first.last@example.com
```

Другой механизм использования обратных ссылок в выражениях выбирает шаблон в зависимости от того, было ли найдено совпадение для предыдущей группы. Шаблон для поиска адресов электронной почты можно скорректировать таким образом, чтобы угловые скобки требовались лишь при наличии имени и не требовались, если проверяется сам электронный адрес. Проверка того, было ли найдено совпадение для группы, осуществляется с помощью синтаксиса `(?(id)yes-expression|no-expression)`, где `id` — имя или номер группы, `yes-expression` — шаблон, используемый, если группа имеет значение, а `no-expression` — шаблон, используемый в противном случае.

Листинг 1.55. `re_id.py`

```
import re

address = re.compile(
    '''
    ^

    # Имя состоит из букв и может включать символы точки "."
    # в сокращенных вариантах обращения и инициалах
    (?P<name>
       ([\w.]+\s+)*[\w.]+
    )?
    \s*

    # Адреса электронной почты должны быть заключены в угловые
    # скобки, но только в том случае, если найдено имя
    (?(name)
      # Остаток заключен в угловые скобки, поскольку
      # имя присутствует
      (?P<brackets>(?=(<.*>$)))
      |
      # Остаток не заключен в угловые скобки, поскольку
      # имя отсутствует

      (?=([^<].*[^>]$))
    )

    # Находить угловую скобку только в том случае, если
    # опережающая проверка обнаружила обе скобки
    (?(brackets)<|\s*)

    # Собственно адрес: username@domain.tld
    (?P<email>
      [\w\d.+-]+ # Имя пользователя
      @
      ([\w\d.]+\.)+ # Префикс имени домена
      (com|org|edu) # Ограничение списка доменов верхнего уровня
     )
    # Находить угловую скобку только в том случае, если
    # опережающая проверка обнаружила обе скобки
    (?(brackets)>|\s*)

    $
    ''',
    re.VERBOSE)

candidates = [
    u'First Last <first.last@example.com>',
    u'No Brackets first.last@example.com',
    u'Open Bracket <first.last@example.com',
    u'Close Bracket first.last@example.com>',
    u'no.brackets@example.com',
```

```
]

for candidate in candidates:
    print('Candidate:', candidate)
    match = address.search(candidate)
    if match:
        print(' Match name :', match.groupdict()['name'])
        print(' Match email:', match.groupdict()['email'])
    else:
    print(' No match')
```

В этой версии выражения для разбора адреса электронной почты используются две проверки. Если для группы `name` найдено совпадение, то проверка требует наличия обеих скобок и устанавливает группу `brackets`. Если совпадения для `name` не найдено, то проверка требует, чтобы остальная часть текста не была заключена в угловые скобки. Далее, если группа `brackets` установлена, фактический шаблон, выполняющий поиск совпадений, захватывает скобки во входном тексте, используя литеральные шаблоны; в противном случае захватывается пустое пространство.

```
$ python3 re_id.py
Candidate: First Last <first.last@example.com>
Match name : First Last
Match email: first.last@example.com
Candidate: No Brackets first.last@example.com
No match
Candidate: Open Bracket <first.last@example.com
No match
Candidate: Close Bracket first.last@example.com>
No match
Candidate: no.brackets@example.com
Match name : None
Match email: no.brackets@example.com
```

1.3.10. Изменение строк с помощью шаблонов

Помимо операций поиска в тексте, модуль `re` поддерживает изменение текста за счет использования регулярных выражений в качестве поискового механизма и обратных ссылок на группы в качестве замещающего текста. Для замены всех вхождений шаблона другой строкой используйте функцию `sub()`.

Листинг 1.56. `re_sub.py`

```
import re

bold = re.compile(r'\*{2}(.*?)\*{2}')

text = 'Make this **bold**.  This **too**.'

print('Text:', text)
print('Bold:', bold.sub(r'<b>\1</b>', text))
```

Для ссылки на текст, совпавший с шаблоном, можно использовать синтаксис нумерованных обратных ссылок (*номер*).

```
$ python3 re_sub.py

Text: Make this **bold**.  This **too**.
Bold: Make this <b>bold</b>. This <b>too</b>.
```

Именованные группы обозначаются в замещающем тексте с помощью синтаксиса \g<*имя*>.

Листинг 1.57. re_sub_named_groups.py

```
import re

bold = re.compile(r'\*{2}(?P<bold_text>.*?)\*{2}')

text = 'Make this **bold**.  This **too**.'

print('Text:', text)
print('Bold:', bold.sub(r'<b>\g<bold_text></b>', text))
```

Синтаксис \g<*имя*> работает также с нумерованными ссылками, и его использование позволяет устранить неоднозначность, создаваемую одновременным наличием номеров групп и окружающими литеральными цифрами.

```
$ python3 re_sub_named_groups.py

Text: Make this **bold**.  This **too**.
Bold: Make this <b>bold</b>.  This <b>too</b>.
```

Количество выполняемых замен можно ограничить, передав функции `sub()` значение `count`.

Листинг 1.58. re_sub_count.py

```
import re

bold = re.compile(r'\*{2}(.*?)\*{2}')

text = 'Make this **bold**.  This **too**.'

print('Text:', text)
print('Bold:', bold.sub(r'<b>\1</b>', text, count=1))
```

Будет выполнена всего одна замена, поскольку аргумент `count` равен `1`.

```
$ python3 re_sub_count.py

Text: Make this **bold**.  This **too**.
Bold: Make this <b>bold</b>.  This **too**.
```

Функция `subn()` аналогична функции `sub()`, за исключением того, что она возвращает как измененную строку, так и количество выполненных замен.

Листинг 1.59. re_subn.py

```
import re

bold = re.compile(r'\*{2}(.*?)\*{2}')

text = 'Make this **bold**.  This **too**.'

print('Text:', text)
print('Bold:', bold.subn(r'<b>\1</b>', text))
```

В этом примере шаблон поиска совпадает с текстом дважды.

```
$ python3 re_subn.py

Text: Make this **bold**.  This **too**.
Bold: ('Make this <b>bold</b>.  This <b>too</b>.', 2)
```

1.3.11. Разбиение текста с помощью шаблонов

Метод `str.split()` — один из тех, которые чаще всего используются для разбиения анализируемых строк. Однако в качестве разделителей он поддерживает лишь литеральные значения, в связи с чем иногда для этих целей приходится использовать регулярные выражения. Например, во многих языках разметки простого текста разделителем абзацев служит последовательность двух или более символов перевода строки (`\n`). Именно слова "двух или более символов" в этом определении делают невозможным применение метода `str.split()` в подобных случаях. Стратегия идентификации абзацев с помощью функции `findall()` предполагает использование шаблона вида `(.+?)\n{2,}`.

Листинг 1.60. re_paragraphs_findall.py

```
import re

text = '''Paragraph one
on two lines.

Paragraph two.


Paragraph three.'''

for num, para in enumerate(re.findall(r'(.+?)\n{2,}',
                                      text,
                                      flags=re.DOTALL)
                           ):
    print(num, repr(para))
    print()
```

Этот шаблон не работает для абзацев, находящихся в конце входного текста, что подтверждается отсутствием текста "Paragraph three." в составе выведенной информации.

```
$ python3 re_paragraphs_findall.py

0 'Paragraph one\non two lines.'

1 'Paragraph two.'
```

Расширение шаблона для учета того факта, что концом абзаца должен считаться не только удвоенный символ перевода строки, но и конец всего входного текста, позволяет решить эту проблему, но усложняет шаблон. Замена функции `re.findall()` функцией `re.split()` позволяет справиться с граничными условиями и сохранить простоту шаблона.

Листинг 1.61. `re_split.py`

```
import re

text = '''Paragraph one
on two lines.

Paragraph two.


Paragraph three.'''

print('With findall:')
for num, para in enumerate(re.findall(r'(.+?)(\n{2,}|$)',
                                      text,
                                      flags=re.DOTALL)):
    print(num, repr(para))
    print()

print()
print('With split:')
for num, para in enumerate(re.split(r'\n{2,}', text)):
    print(num, repr(para))
print()
```

Аргумент шаблона в функции `split()` более точно выражает спецификацию разметки. Два и более символа перевода строки обозначают границу между абзацами во входной строке.

```
$ python3 re_split.py

With findall:
0 ('Paragraph one\non two lines.', '\n\n')

1 ('Paragraph two.', '\n\n\n')

2 ('Paragraph three.', '')


With split:
0 'Paragraph one\non two lines.'
```

```
1 'Paragraph two.'

2 'Paragraph three.'
```

Заключение выражения в скобки для определения группы заставляет функцию `split()` работать аналогично методу `str.partition()`, поэтому наряду с другими частями строки он возвращает также значения разделителя.

Листинг 1.62. `re_split_groups.py`

```
import re

text = '''Paragraph one
on two lines.

Paragraph two.

Paragraph three.'''

print('With split:')
for num, para in enumerate(re.split(r'(\n{2,})', text)):
    print(num, repr(para))
    print()
```

Теперь в вывод включаются не только все абзацы, но и разделяющие их символы перевода строки.

```
$ python3 re_split_groups.py

With split:
0 'Paragraph one\non two lines.'

1 '\n\n'

2 'Paragraph two.'
3 '\n\n\n'
4 'Paragraph three.'
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `re`[7].
- *Regular Expression HOWTO* (Andrew Kuchling). Введение в регулярные выражения для разработчиков Python.
- *Kodos* (Phil Schwartz)[8]. Интерактивная программа для тестирования регулярных выражений.
- *pythex* (Gabriel Rodriguez)[9]. Веб-инструмент для тестирования регулярных выражений, аналогичный Rubular.

[7] `https://docs.python.org/3.5/library/re.html`

[8] `http://kodos.sourceforge.net`

[9] `pythex.org`

- Википедия: *Регулярные выражения*[10]. Общее введение в теорию и практику регулярных выражений.
- `locale` (раздел 15.2). Использование модуля `locale` для настройки языковых параметров при работе с текстом в кодировках Unicode.
- `Unicodedata`. Программный доступ к базе данных свойств символов Unicode.

1.4. `difflib`: сравнение последовательностей

Модуль `difflib` содержит инструменты, предназначенные для вычисления и обработки различий между последовательностями. Он особенно полезен для сравнения текстов и включает функции, генерирующие отчеты с использованием нескольких популярных форматов.

В приведенных в этом разделе примерах используется текст, который хранится в модуле `difflib_data.py`.

Листинг 1.63. `difflib_data.py`

```
text1 = """Lorem ipsum dolor sit amet, consectetuer adipiscing
elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
pulvinar porttitor tellus. Aliquam venenatis. Donec facilisis
pharetra tortor.  In nec mauris eget magna consequat
convalis. Nam sed sem vitae odio pellentesque interdum. Sed
consequat viverra nisl. Suspendisse arcu metus, blandit quis,
rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate
tristique enim. Donec quis lectus a justo imperdiet tempus."""

text1_lines = text1.splitlines()

text2 = """Lorem ipsum dolor sit amet, consectetuer adipiscing
elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
pulvinar, porttitor tellus. Aliquam venenatis. Donec facilisis
pharetra tortor. In nec mauris eget magna consequat
convalis. Nam cras vitae mi vitae odio pellentesque interdum. Sed
consequat viverra nisl. Suspendisse arcu metus, blandit quis,
rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Duis vulputate tristique enim. Donec quis lectus a
justo imperdiet tempus. Suspendisse eu lectus. In nunc."""

text2_lines = text2.splitlines()
```

1.4.1. Сравнение версий текста

Класс `Differ` работает с последовательностями текстовых строк, генерируя так называемые *дельты* — инструкции, включающие различия между отдельными

[10] https://ru.wikipedia.org/wiki/Регулярные_выражения

строками в удобочитаемом виде. Информация, выводимая классом `Differ` по умолчанию, аналогична той, которая выводится командой `diff` в Unix. Она включает исходные входные значения обоих списков, в том числе общие значения, и разметку, обозначающую внесенные изменения.

- Строки с префиксом `-`, которые встречаются в первой последовательности, но отсутствуют во второй.
- Строки с префиксом `+`, которые встречаются во второй последовательности, но отсутствуют в первой.
- В случае строк с частичными различиями между версиями используется дополнительная строка с префиксом `?`, выделяющая изменение в новой версии.
- Если строка не изменилась, то она выводится с префиксом в виде дополнительного пробела в левом столбце, что обеспечивает их выравнивание относительно других строк, претерпевших изменения.

Разбиение текста на последовательности отдельных строк перед передачей их методу `compare()` улучшает читаемость выводимых результатов по сравнению с передачей длинных строк.

Листинг 1.64. `difflib_differ.py`

```
import difflib
from difflib_data import *

d = difflib.Differ()
diff = d.compare(text1_lines, text2_lines)
print('\n'.join(diff))
```

Начальные части обоих текстовых фрагментов, используемых в примере, совпадают, поэтому первая строка выводится без аннотаций.

```
Lorem ipsum dolor sit amet, consectetuer adipiscing
elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
```

Третья строка была изменена за счет добавления запятой в измененном тексте. Выводятся обе версии строки с предоставлением дополнительной информации о строке 5 в виде позиции, в которой был изменен текст, с указанием того, что была добавлена запятая.

```
- pulvinar porttitor tellus. Aliquam venenatis. Donec facilisis
+ pulvinar, porttitor tellus. Aliquam venenatis. Donec facilisis
?         +
```

В нескольких следующих строках отмечено удаление лишних пробелов.

```
- pharetra tortor.  In nec mauris eget magna consequat
?                 -

+ ,pharetra tortor. In nec mauris eget magna consequat
```

Далее следует более сложное изменение — замена нескольких слов во фразе.

```
- convalis. Nam sed sem vitae odio pellentesque interdum. Sed
?                  - --

+ convalis. Nam cras vitae mi vitae odio pellentesque interdum. Sed
?                ++++ +++++   +
```

Последнее предложение в абзаце претерпело существенные изменения, поэтому различие представлено удалением старой версии и добавлением новой.

```
  consequat viverra nisl. Suspendisse arcu metus, blandit quis,
  rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
  molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
  tristique vel, mauris. Curabitur vel lorem id nisl porta
- adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate
- tristique enim. Donec quis lectus a justo imperdiet tempus.
+ adipiscing. Duis vulputate tristique enim. Donec quis lectus a
+ justo imperdiet tempus. Suspendisse eu lectus. In nunc.
```

Результат, получаемый с помощью функции `ndiff()`, практически совпадает с описанным выше. Обработка учитывает специфику работы с текстовыми данными и удаляет "шум" из входных данных.

1.4.1.1. Другие выходные форматы

В то время как класс `Differ` отображает все входные строки, функция `unified_diff()` выводит лишь измененные строки и немного контекста.

Листинг 1.65. `difflib_unified.py`

```
import difflib
from difflib_data import *

diff = difflib.unified_diff(
    text1_lines,
    text2_lines,
    lineterm='',
)
print('\n'.join(list(diff)))
```

Аргумент `lineterm` используется для того, чтобы сообщить функции `unified_diff()` о том, что не следует добавлять символы перевода строки к управляющим строкам, которые она возвращает, поскольку входные строки их не включают. Символы перевода строки добавляются ко всем строкам при их выводе на экран. Характер выводимых результатов должен быть знаком пользователям многих систем управления версиями.

```
$ python3 difflib_unified.py
---
+++
@@ -1,11 +1,11 @@
 Lorem ipsum dolor sit amet, consectetuer adipiscing
 elit. Integer eu lacus accumsan arcu fermentum euismod. Donec
```

```
-pulvinar porttitor tellus. Aliquam venenatis. Donec facilisis
+pulvinar, porttitor tellus. Aliquam venenatis. Donec facilisis
 pharetra tortor.  In nec mauris eget magna consequat
-convalis. Nam sed sem vitae odio pellentesque interdum. Sed
+convalis. Nam cras vitae mi vitae odio pellentesque interdum. Sed
 consequat viverra nisl. Suspendisse arcu metus, blandit quis,
 rhoncus ac, pharetra eget, velit. Mauris urna. Morbi nonummy
 molestie orci. Praesent nisi elit, fringilla ac, suscipit non,
 tristique vel, mauris. Curabitur vel lorem id nisl porta
-adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate
-tristique enim. Donec quis lectus a justo imperdiet tempus.
+adipiscing. Duis vulputate tristique enim. Donec quis lectus a
+justo imperdiet tempus. Suspendisse eu lectus. In nunc.
```

Функция `context_diff()` выводит результаты в аналогичной удобочитаемой форме.

1.4.2. Ненужные данные

Все функции, обеспечивающие получение разностных последовательностей, поддерживают аргументы, с помощью которых можно указать, какие строки и символы в строках следует игнорировать. Эти параметры можно использовать для отказа от отображения различий в разметке или пробельных промежутках между двумя версиями файла.

Листинг 1.66. `difflib_junk.py`

```
# Этот пример основан на исходном коде difflib.py.

from difflib import SequenceMatcher


def show_results(match):
    print(' a = {}'.format(match.a))
    print(' b = {}'.format(match.b))
    print(' size = {}'.format(match.size))
    i, j, k = match
    print(' A[a:a+size] = {!r}'.format(A[i:i + k]))
    print(' B[b:b+size] = {!r}'.format(B[j:j + k]))

A = " abcd"
B = "abcd abcd"

print('A = {!r}'.format(A))
print('B = {!r}'.format(B))

print('\nWithout junk detection:')
s1 = SequenceMatcher(None, A, B)
match1 = s1.find_longest_match(0, len(A), 0, len(B))
show_results(match1)

print('\nTreat spaces as junk:')
s2 = SequenceMatcher(lambda x: x == " ", A, B)
```

```
match2 = s2.find_longest_match(0, len(A), 0, len(B))
show_results(match2)
```

По умолчанию класс `Differ` не прибегает к явному игнорированию некоторых строк и символов, полагаясь в этом отношении на способность класса `SequenceMatcher` обнаруживать шум. Функция `ndiff()` по умолчанию игнорирует пробелы и символы табуляции.

```
$ python3 difflib_junk.py

A = ' abcd'
B = 'abcd abcd'

Without junk detection:
  a    = 0
  b    = 4
  size = 5
  A[a:a+size] = ' abcd'
  B[b:b+size] = ' abcd'

Treat spaces as junk:
  a    = 1
  b    = 0
  size = 4
A[a:a+size] = 'abcd'
B[b:b+size] = 'abcd'
```

1.4.3. Сравнение произвольных типов

Класс `SequenceMatcher` обеспечивает сравнение двух последовательностей любого типа, если их значения являются хешируемыми. Он использует алгоритм обнаружения непрерывных совпадающих блоков наибольшей длины и удаляет "бесполезные" значения, которые не вносят вклад в реальные данные. Метод `get_opcodes()` возвращает список инструкций, применение которых к первой последовательности обеспечивает ее совпадение со второй последовательностью. Каждая из этих инструкций кодируется в виде кортежа из пяти элементов, включающего строку инструкции (код операции – "opcode") и по паре начального и конечного индексов для каждой из последовательностей (обозначаемых `i1`, `i2`, `j1` и `j2`), как показано в табл. 1.4.

Таблица 1.4. Инструкции метода `difflib.get_opcodes()`

Код операции	Определение
`'replace'`	Заменяет `a[i1:i2]` на `b[j1:j2]`
`'delete'`	Полностью удаляет `a[i1:i2]`
`'insert'`	Вставляет `b[j1:j2]` в `a[i1:i1]`
`'equal'`	Подпоследовательности уже равны

Листинг 1.67. `difflib_seq.py`

```
import difflib

s1 = [1, 2, 3, 5, 6, 4]
s2 = [2, 3, 5, 4, 6, 1]

print('Initial data:')
print('s1 =', s1)
print('s2 =', s2)
print('s1 == s2:', s1 == s2)
print()

matcher = difflib.SequenceMatcher(None, s1, s2)
for tag, i1, i2, j1, j2 in reversed(matcher.get_opcodes()):

    if tag == 'delete':
        print('Remove {} from positions [{}:{}]'.format(
            s1[i1:i2], i1, i2))
        print(' before =', s1)
        del s1[i1:i2]

    elif tag == 'equal':
        print('s1[{}:{}] and s2[{}:{}] are the same'.format(
                i1, i2, j1, j2))

    elif tag == 'insert':
        print('Insert {} from s2[{}:{}] into s1 at {}'.format(
            s2[j1:j2], j1, j2, i1))
        print(' before =', s1)
        s1[i1:i2] = s2[j1:j2]

    elif tag == 'replace':
        print(('Replace {} from s1[{}:{}] '
               'with {} from s2[{}:{}]').format(
                   s1[i1:i2], i1, i2, s2[j1:j2], j1, j2))
        print(' before =', s1)
        s1[i1:i2] = s2[j1:j2]

    print(' after =', s1, '\n')

print('s1 == s2:', s1 == s2)
```

В этом примере сравнение двух списков целочисленных значений с помощью метода `get_opcodes()` используется для получения инструкций, преобразующих исходный список в обновленную версию. Изменения применяются в обратном порядке, что обеспечивает актуальность их значений даже после удаления или добавления элементов.

```
$ python3 difflib_seq.py

Initial data:
s1 = [1, 2, 3, 5, 6, 4]
```

```
s2 = [2, 3, 5, 4, 6, 1]
s1 == s2: False

Replace [4] from s1[5:6] with [1] from s2[5:6]
  before = [1, 2, 3, 5, 6, 4]
   after = [1, 2, 3, 5, 6, 1]

s1[4:5] and s2[4:5] are the same
   after = [1, 2, 3, 5, 6, 1]

Insert [4] from s2[3:4] into s1 at 4
  before = [1, 2, 3, 5, 6, 1]
   after = [1, 2, 3, 5, 4, 6, 1]

s1[1:4] and s2[0:3] are the same
   after = [1, 2, 3, 5, 4, 6, 1]

Remove [1] from positions [0:1]
  before = [1, 2, 3, 5, 4, 6, 1]
   after = [2, 3, 5, 4, 6, 1]

s1 == s2: True
```

Класс `SequenceMatcher` работает как с пользовательскими классами, так и со встроенными типами, при условии, что они являются хешируемыми.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `difflib`[11].
- *Pattern Matching: The Gestalt Approach*[12] (John W. Ratcliff, D. E. Metzener). Обсуждение аналогичного алгоритма, опубликованного в *Dr. Dobb's Journal* (июль, 1988 год).

[11] https://docs.python.org/3.5/library/difflib.html

[12] www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970

Глава 2

Структуры данных

Python включает несколько стандартных структур данных, таких как `list`, `tuple`, `dict` и `set`, как часть своих встроенных типов. Для многих приложений другие структуры не потребуются, но даже если и потребуются, то стандартная библиотека содержит их мощные, прошедшие тщательное тестирование версии, готовые к использованию.

Модуль `enum` (раздел 2.1) предоставляет реализацию *перечисления* как типа данных с возможностями итерирования и сравнения значений. Он позволяет создавать символические имена для значений вместо того, чтобы использовать литеральные строки или целочисленные значения.

Модуль `collections` (раздел 2.2) включает реализации нескольких структур данных, которые расширяют структуры, уже существующие в других модулях. Например, тип `Deque` — это двухсторонняя очередь, допускающая добавление и удаление элементов на любом из ее концов. Тип `defaultdict` — словарь, возвращающий значение по умолчанию для отсутствующего ключа, тогда как `OrderedDict` запоминает последовательность, в которой в него добавлялись элементы. Тип `namedtuple` расширяет обычный тип `tuple`, разрешая присваивать элементам имена в дополнение к числовым индексам.

В случае больших объемов данных тип `array` (раздел 2.3) может обеспечить более эффективное использование памяти, чем `list`. Поскольку его элементами должны быть данные одного типа, он может использовать более компактное представление памяти, чем универсальный тип `list`. В то же время многие из методов `list` пригодны для манипулирования экземплярами `array`, и поэтому в приложениях списки часто могут быть заменены массивами за счет внесения лишь небольших изменений в приложение.

Один из важнейших аспектов манипулирования данными — сортировка элементов. Тип `list` в Python включает метод `sort()`, но иногда гораздо эффективнее поддерживать список в отсортированном состоянии, не прибегая к пересортировке каждый раз, когда изменяется его содержимое. Функции модуля `heapq` (раздел 2.4) изменяют содержимое списка, сохраняя упорядоченное состояние элементов за счет незначительных издержек.

Еще одну возможность создания сортированных списков или массивов предлагает модуль `bisect` (раздел 2.5). Он использует бинарный (двоичный) поиск для нахождения точки вставки новых элементов, что представляет собой альтернативу повторной сортировке часто изменяемых списков.

Несмотря на то что методы `insert()` и `pop()` встроенного типа `list` позволяют имитировать очереди, этот способ не является потокобезопасным. Подлинное упорядочение обмена сообщениями между потоками обеспечивает использование модуля `queue` (раздел 2.6). Модуль `multiprocessing` (раздел 10.4) включает версию класса `Queue`, которая обеспечивает межпроцессное взаимодей-

ствие, упрощая преобразование многопоточной программы в программу, работающую с процессами.

Модуль `struct` (раздел 2.7) используется для декодирования данных других приложений (возможно, поступающих из бинарного файла или потока данных) в собственные типы Python с целью упрощения манипулирования данными.

В этой главе обсуждаются два модуля, имеющих отношение к управлению памятью. Для работы с такими структурами данных, как графы или деревья, характеризующиеся сложной организацией внутренних связей между элементами, используйте модуль `weakref` (раздел 2.8), который обеспечивает поддержку ссылок, одновременно разрешая сборщику мусора удалять из памяти более не используемые объекты. Для дублирования структур данных и их содержимого используйте функции модуля `copy` (раздел 2.9), в том числе функцию `deepcopy()`, обеспечивающую рекурсивное копирование вложенных структур.

Обработка отладочных структур данных может занимать много времени, особенно если она требует просмотра больших объемов информации, выведенной на печать. Используйте модуль `pprint` (раздел 2.10) для создания удобочитаемых форм представления информации, выводимой на консоль или записываемой в файл журнала, чтобы упростить процесс отладки.

Наконец, если существующие типы не удовлетворяют вашим потребностям, создайте подкласс одного из встроенных типов Python и адаптируйте его под себя или создайте новый контейнерный тип, используя один из абстрактных базовых классов, определенных в модуле `collections` (раздел 2.2), в качестве отправной точки.

2.1. `enum`: перечисление

Модуль `enum` определяет перечислимый тип данных с возможностями итерирования и сравнения элементов. Он позволяет создавать символические имена для значений, чтобы не нужно было использовать строковые литералы или целочисленные значения.

2.1.1. Создание перечислений

Новое перечисление определяется с использованием синтаксиса `class` посредством создания подкласса класса `Enum` и добавления атрибутов класса, описывающих значения.

Листинг 2.1. `enum_create.py`

```
import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
```

```
    fix_released = 1


print('\nMember name: {}'.format(BugStatus.wont_fix.name))
print('Member value: {}'.format(BugStatus.wont_fix.value))
```

В ходе синтаксического анализа класса элементы Enum преобразуются в экземпляры. Каждый экземпляр имеет свойство name, соответствующее имени элемента, и свойство value, которое соответствует значению, присвоенному имени в определении класса.

```
$ python3 enum_create.py


Member name: wont_fix
Member value: 4
```

2.1.2. Итерирование по элементам

Итерирование по классу перечисления позволяет получить отдельные элементы перечисления.

Листинг 2.2. enum_iterate.py

```
import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1


for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

Элементы предоставляются в том порядке, в каком они объявлены в определении класса. Их имена и значения никоим образом не используются для сортировки.

```
$ python3 enum_iterate.py

new             = 7
incomplete      = 6
invalid         = 5
wont_fix        = 4
in_progress     = 3
fix_committed   = 2
fix_released    = 1
```

2.1.3. Сравнение элементов перечислений

Поскольку элементы перечислений не упорядочены, для них возможны лишь операции проверки равенства (Equality) и тождественности (Identity).

Листинг 2.3. enum_comparison.py

```
import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1


actual_state = BugStatus.wont_fix
desired_state = BugStatus.fix_released

print('Equality:',
      actual_state == desired_state,
      actual_state == BugStatus.wont_fix)
print('Identity:',
      actual_state is desired_state,
      actual_state is BugStatus.wont_fix)
print('Ordered by value:')
try:
    print('\n'.join('  ' + s.name for s in sorted(BugStatus)))
except TypeError as err:
    print('  Cannot sort: {}'.format(err))
```

При попытке выполнения операций сравнения "больше чем" или "меньше чем" возбуждается исключение TypeError.

```
$ python3 enum_comparison.py

Equality: False True
Identity: False True
Ordered by value:
  Cannot sort: unorderable types: BugStatus() < BugStatus()
```

Для перечислений, элементы которых должны вести себя как числа, — например, в отношении операций сравнения, — используйте класс IntEnum.

Листинг 2.4. enum_intenum.py

```
import enum


class BugStatus(enum.IntEnum):
```

```
    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1

print('Ordered by value:')
print('\n'.join('  ' + s.name for s in sorted(BugStatus)))


$ python3 enum_intenum.py
Ordered by value:
  fix_released
  fix_committed
  in_progress
  wont_fix
  invalid
  incomplete
  new
```

2.1.4. Уникальность значений перечисления

Элементы перечисления, которым соответствует одно и то же значение, являются всего лишь алиасными (альтернативными) ссылками на один и тот же объект. Наличие альтернативных ссылок не приводит к появлению повторяющихся значений в итераторе для Enum.

Листинг 2.5. enum_aliases.py

```
import enum


class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1

    by_design = 4
    closed = 1


for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

```
print('\nSame: by_design is wont_fix: ',
      BugStatus.by_design is BugStatus.wont_fix)
print('Same: closed is fix_released: ',
      BugStatus.closed is BugStatus.fix_released)
```

Поскольку by_design и closed являются псевдонимами других элементов, они не появляются в выводе при итерировании по перечислению. Каноническим именем элемента является то, которое было первым связано со значением.

```
$ python3 enum_aliases.py

new = 7
incomplete      = 6
invalid         = 5
wont_fix        = 4
in_progress     = 3
fix_committed   = 2
fix_released    = 1

Same: by_design is wont_fix: True
Same: closed is fix_released: True
```

Чтобы потребовать уникальность значений перечисления, следует декорировать тип Enum декоратором @unique.

Листинг 2.6. enum_unique_enforce.py

```
import enum


@enum.unique
class BugStatus(enum.Enum):

    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1

    # При наличии декоратора unique возбуждается исключение
    by_design = 4
    closed = 1
```

При наличии элементов с одинаковыми значениями интерпретатор возбуждает исключение ValueError.

```
$ python3 enum_unique_enforce.py

Traceback (most recent call last):
  File "enum_unique_enforce.py", line 11, in <module>
    class BugStatus(enum.Enum):
  File ".../lib/python3.5/enum.py", line 573, in unique
```

```
    (enumeration, alias_details))
ValueError: duplicate values found in <enum 'BugStatus'>:
by_design -> wont_fix, closed -> fix_released
```

2.1.5. Создание перечислений программным способом

В некоторых случаях удобнее создавать перечисления программным способом, а не задавать их в коде определения класса. В подобных ситуациях класс Enum поддерживает передачу значений конструктору класса.

Листинг 2.7. enum_programmatic_create.py

```
import enum


BugStatus = enum.Enum(
    value='BugStatus',
    names=('fix_released fix_committed in_progress '
           'wont_fix invalid incomplete new'),
)

print('Member: {}'.format(BugStatus.new))

print('\nAll members:')
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

Аргумент value — это имя перечисления, которое используется для создания представления элементов. Аргумент names содержит имена элементов перечисления. Если он передается в виде одиночной строки, то она разбивается с использованием пробелов и запятых в качестве разделителей, а результирующие лексемы используются в качестве имен элементов, которым автоматически присваиваются значения, начиная с 1.

```
$ python3 enum_programmatic_create.py

Member: BugStatus.new

All members:
fix_released    = 1
fix_committed   = 2
in_progress     = 3
wont_fix        = 4
invalid         = 5
incomplete      = 6
new             = 7
```

С целью получения большего контроля над значениями, связанными с элементами, вместо строки с именами можно передавать двухкомпонентные кортежи или словарь, отображающий имена на значения.

Листинг 2.8. enum_programmatic_mapping.py

```
import enum


BugStatus = enum.Enum(
    value='BugStatus',
    names=[
        ('new', 7),
        ('incomplete', 6),
        ('invalid', 5),
        ('wont_fix', 4),
        ('in_progress', 3),
        ('fix_committed', 2),
        ('fix_released', 1),
    ],
)

print('All members:')
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

В этом примере вместо строки с именами элементов передаются двухкомпонентные кортежи. Это позволяет реконструировать перечисление BugStatus с сохранением того же порядка элементов, что и в версии, определенной в файле enum_create.py.

```
$ python3 enum_programmatic_mapping.py

All members:
new             = 7
incomplete      = 6
invalid         = 5
wont_fix        = 4
in_progress     = 3
fix_committed   = 2
fix_released    = 1
```

2.1.6. Значения элементов, не являющиеся целыми числами

Значения элементов перечисления не ограничиваются целыми числами. В действительности с каждым элементом перечисления может быть ассоциирован любой объект. Если значениями являются кортежи, их элементы передаются методу __init__() в виде отдельных аргументов.

Листинг 2.9. enum_tuple_values.py

```
import enum


class BugStatus(enum.Enum):

    new = (7, ['incomplete',
```

```
                'invalid',
                'wont_fix',
                'in_progress'])B
    incomplete = (6, ['new', 'wont_fix'])
    invalid = (5, ['new'])
    wont_fix = (4, ['new'])
    in_progress = (3, ['new', 'fix_committed'])
    fix_committed = (2, ['in_progress', 'fix_released'])
    fix_released = (1, ['new'])

    def __init__(self, num, transitions):
        self.num = num
        self.transitions = transitions

    def can_transition(self, new_state):
        return new_state.name in self.transitions


print('Name:', BugStatus.in_progress)
print('Value:', BugStatus.in_progress.value)
print('Custom attribute:', BugStatus.in_progress.transitions)
print('Using attribute:',
      BugStatus.in_progress.can_transition(BugStatus.new))
```

В этом примере значение каждого элемента представляет собой кортеж, содержащий числовой идентификатор (например, типа тех, которые могут сохраняться в базе данных) и список допустимых переходов из текущего состояния в другие.

```
$ python3 enum_tuple_values.py

Name: BugStatus.in_progress
Value: (3, ['new', 'fix_committed'])
Custom attribute: ['new', 'fix_committed']
Using attribute: True
```

В более сложных случаях кортежи могут становиться слишком громоздкими. Поскольку значениями элементов могут быть объекты любого типа, то при большом количестве отдельных атрибутов, которые приходится отслеживать для каждого из значений перечисления, более удобными могут оказаться словари. Сложные значения передаются непосредственно методу `__init__()` в качестве единственного аргумента (не считая `self`).

Листинг 2.10. `enum_complex_values.py`

```
import enum


class BugStatus(enum.Enum):

    new = {
        'num': 7,
        'transitions': [
```

```
            'incomplete',
            'invalid',
            'wont_fix',
            'in_progress',
        ],
    }
    incomplete = {
        'num': 6,
        'transitions': ['new', 'wont_fix'],
    }
    invalid = {
        'num': 5,
        'transitions': ['new'],
    }
    wont_fix = {
        'num': 4,
        'transitions': ['new'],
    }
    in_progress = {
        'num': 3,
        'transitions': ['new', 'fix_committed'],
    }
    fix_committed = {
        'num': 2,
        'transitions': ['in_progress', 'fix_released'],
    }
    fix_released = {
        'num': 1,
        'transitions': ['new'],
    }

    def __init__(self, vals):
        self.num = vals['num']
        self.transitions = vals['transitions']

    def can_transition(self, new_state):
        return new_state.name in self.transitions


print('Name:', BugStatus.in_progress)
print('Value:', BugStatus.in_progress.value)
print('Custom attribute:', BugStatus.in_progress.transitions)
print('Using attribute:',
      BugStatus.in_progress.can_transition(BugStatus.new))
```

В этом примере представлены те же данные, что и в предыдущем, но с использованием словарей, а не кортежей.

```
$ python3 enum_complex_values.py

Name: BugStatus.in_progress
Value: {'transitions': ['new', 'fix_committed'], 'num': 3}
```

```
Custom attribute: ['new', 'fix_committed']
Using attribute: True
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `enum`[1].
- **PEP 435**[2]. *Adding an Enum type to the Python standard library.*
- `flufl.enum` — *A Python enumeration package*[3] (Barry Warsaw). Исходная реализация перечислений.

2.2. `collections`: контейнерные типы данных

Модуль `collections` включает контейнерные типы данных, которые не охватываются встроенными типами `list`, `dict` и `tuple`.

2.2.1. `ChainMap`: поиск в нескольких словарях

Класс `ChainMap` управляет последовательностью словарей, используя их в порядке появления для поиска значений, ассоциированных с ключами. Этот класс хорошо подходит на роль "контекстного" контейнера, поскольку с ним можно обращаться как со стеком, в котором изменения вступают в силу при добавлении элемента и отбрасываются, когда элемент удаляется из стека.

2.2.1.1. Доступ к значениям

Класс `ChainMap` поддерживает тот же API для доступа к существующим значениям, что и обычный словарь.

Листинг 2.11. `collections_chainmap_read.py`

```
import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)

print('Individual Values')
print('a = {}'.format(m['a']))
print('b = {}'.format(m['b']))
print('c = {}'.format(m['c']))
print()

print('Keys = {}'.format(list(m.keys())))
print('Values = {}'.format(list(m.values())))
print()

print('Items:')
```

[1] https://docs.python.org/3.5/library/enum.html

[2] www.python.org/dev/peps/pep-0435

[3] http://pythonhosted.org/flufl.enum/

```
for k, v in m.items():
    print('{} = {}'.format(k, v))
print()

print('"d" in m: {}'.format(('d' in m)))
```

Дочерние отображения используются для поиска в том порядке, в каком они были переданы конструктору, поэтому значение, выведенное для ключа 'c', берется из словаря a.

```
$ python3 collections_chainmap_read.py

Individual Values
a = A
b = B
c = C

Keys = ['c', 'b', 'a']
Values = ['C', 'B', 'A']

Items:
c = C
b = B
a = A
"d" in m: False
```

2.2.1.2. Переупорядочение

Список отображений, используемых классом ChainMap для поиска значений, хранится в его атрибуте maps. Этот список — изменяемый и потому допускает непосредственное добавление новых отображений и изменение порядка следования элементов для управления процессом поиска и изменения поведения.

Листинг 2.12. collections_chainmap_reorder.py

```
import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)

print(m.maps)
print('c = {}\n'.format(m['c']))

# Reverse the list.
m.maps = list(reversed(m.maps))

print(m.maps)
print('c = {}'.format(m['c']))
```

Обращение списка отображений приводит к изменению значения, ассоциированного с ключом 'c'.

```
$ python3 collections_chainmap_reorder.py

[{'c': 'C', 'a': 'A'}, {'c': 'D', 'b': 'B'}]
c = C

[{'c': 'D', 'b': 'B'}, {'c': 'C', 'a': 'A'}]
c = D
```

2.2.1.3. Обновление значений

Класс ChainMap не кеширует значения, хранящиеся в дочерних отображениях. Таким образом, любое изменение их содержимого отражается на значениях, получаемых при обращении к ChainMap.

Листинг 2.13. collections_chainmap_update_behind.py

```
import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)
print('Before: {}'.format(m['c']))
a['c'] = 'E'
print('After : {}'.format(m['c']))
```

Операции изменения значений, ассоциированных с существующими ключами, и добавления новых элементов работают одинаковым образом.

```
$ python3 collections_chainmap_update_behind.py

Before: C
After : E
```

Для установки значений можно использовать непосредственно класс ChainMap, однако соответствующее изменение затронет лишь первое из отображений в цепочке.

Листинг 2.14. collections_chainmap_update_directly.py

```
import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)
print('Before:', m)
m['c'] = 'E'
print('After :', m)
print('a:', a)
```

При сохранении нового значения с использованием объекта m обновляется отображение a.

```
$ python3 collections_chainmap_update_directly.py

Before: ChainMap({'c': 'C', 'a': 'A'}, {'c': 'D', 'b': 'B'})
After : ChainMap({'c': 'E', 'a': 'A'}, {'c': 'D', 'b': 'B'})
a: {'c': 'E', 'a': 'A'}
```

Класс ChainMap предоставляет вспомогательный метод для создания нового экземпляра с одним дополнительным отображением, помещаемым в начало списка maps без изменения уже существующих структур данных.

Листинг 2.15. collections_chainmap_new_child.py

```
import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m1 = collections.ChainMap(a, b)
m2 = m1.new_child()

print('m1 before:', m1)
print('m2 before:', m2)

m2['c'] = 'E'

print('m1 after:', m1)
print('m2 after:', m2)
```

Именно благодаря такому характеру поведения, свойственному стекам, экземпляры ChainMap удобно использовать в качестве шаблонов или контекстов приложений. В частности, не составляет труда добавить или обновить значения на одной итерации и отбросить их на следующей.

```
$ python3 collections_chainmap_new_child.py

m1 before: ChainMap({'c': 'C', 'a': 'A'}, {'c': 'D', 'b': 'B'})
m2 before: ChainMap({}, {'c': 'C', 'a': 'A'}, {'c': 'D', 'b':
'B'})
m1 after: ChainMap({'c': 'C', 'a': 'A'}, {'c': 'D', 'b': 'B'})
m2 after: ChainMap({'c': 'E'}, {'c': 'C', 'a': 'A'}, {'c': 'D',
'b': 'B'})
```

В ситуациях, когда новый контекст известен или создан заранее, его можно передать методу new_child().

Листинг 2.16. collections_chainmap_new_child_explicit.py

```
import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}
c = {'c': 'E'}
```

```
m1 = collections.ChainMap(a, b)
m2 = m1.new_child(c)

print('m1["c"] = {}'.format(m1['c']))
print('m2["c"] = {}'.format(m2['c']))
```

Этот код, эквивалентный вызову

```
m2 = collections.ChainMap(c, *m1.maps)
```

дает следующий результат:

```
$ python3 collections_chainmap_new_child_explicit.py

m1["c"] = C
m2["c"] = E
```

2.2.2. Counter: подсчет хешируемых экземпляров

Класс Counter – это контейнер, предназначенный для подсчета числа сохраненных эквивалентных значений. Он обеспечивает реализацию алгоритмов, для которых в других языках программирования обычно используют мультимножества.

2.2.2.1. Инициализация

Класс Counter поддерживает три формы инициализации. При вызове конструктора ему можно передать последовательность элементов, словарь, содержащий ключи и значения, или именованные аргументы, сопоставляющие строки имен со значениями счетчиков.

Листинг 2.17. collections_counter_init.py

```
import collections

print(collections.Counter(['a', 'b', 'c', 'a', 'b', 'b']))
print(collections.Counter({'a': 2, 'b': 3, 'c': 1}))
print(collections.Counter(a=2, b=3, c=1))
```

Все три формы инициализации дают один и тот же результат.

```
$ python3 collections_counter_init.py

Counter({'b': 3, 'a': 2, 'c': 1})
Counter({'b': 3, 'a': 2, 'c': 1})
Counter({'b': 3, 'a': 2, 'c': 1})
```

Можно создать пустой экземпляр Counter, вызвав конструктор без аргументов, и заполнить его значениями с помощью метода update().

Листинг 2.18. collections_counter_update.py

```
import collections

c = collections.Counter()
```

```
print('Initial :', c)

c.update('abcdaab')
print('Sequence:', c)

c.update({'a': 1, 'd': 5})
print('Dict :', c)
```

Значения счетчиков увеличиваются на основании новых данных, а не заменяются ими. В предыдущем примере значение счетчика для a увеличивается с 3 до 4.

```
$ python3 collections_counter_update.py

Initial : Counter()
Sequence: Counter({'a': 3, 'b': 2, 'c': 1, 'd': 1})
Dict    : Counter({'d': 6, 'a': 4, 'b': 2, 'c': 1})
```

2.2.2.2. Доступ к значениям счетчиков

После заполнения экземпляра Counter значениями их можно извлекать с помощью API словаря.

Листинг 2.19. collections_counter_get_values.py

```
import collections

c = collections.Counter('abcdaab')

for letter in 'abcde':
    print('{} : {}'.format(letter, c[letter]))
```

Объект Counter не возбуждает исключения KeyError для неизвестных элементов. Если значение отсутствует во входной строке (как в случае буквы e в данном примере), то значение его счетчика равно 0.

```
$ python3 collections_counter_get_values.py

a : 3
b : 2
c : 1
d : 1
e : 0
```

Метод elements() возвращает итератор, обеспечивающий поочередное извлечение всех элементов, известных экземпляру Counter.

Листинг 2.20. collections_counter_elements.py

```
import collections

c = collections.Counter('extremely')
c['z'] = 0
print(c)
print(list(c.elements()))
```

Итератор возвращает элементы в произвольном порядке, причем элементы, значения счетчиков которых меньше или равны нулю, игнорируются.

```
$ python3 collections_counter_elements.py

Counter({'e': 3, 'x': 1, 'm': 1, 't': 1, 'y': 1, 'l': 1, 'r': 1,
'z': 0})
['x', 'm', 't', 'e', 'e', 'e', 'y', 'l', 'r']
```

Для получения последовательности n наиболее часто встречающихся входных значений используйте метод `most_common()`.

Листинг 2.21. `collections_counter_most_common.py`

```
import collections

c = collections.Counter()
with open('/usr/share/dict/words', 'rt') as f:
    for line in f:
        c.update(line.rstrip().lower())

print('Most common:')
for letter, count in c.most_common(3):
    print('{}: {:>7}'.format(letter, count))
```

В этом примере сначала выполняется подсчет букв, встречающихся во всех словах системного словаря, для нахождения их частотного распределения, после чего выводятся три наиболее часто встречающиеся буквы. Результатом вызова метода `most_common()` без аргументов является список всех элементов в порядке убывания их частоты.

```
$ python3 collections_counter_most_common.py

Most common:
e: 235331
i: 201032
a: 199554
```

2.2.2.3. Арифметические операции

Экземпляры `Counter` поддерживают арифметические операции и операции над множествами для агрегирования результатов. Следующий пример демонстрирует создание новых экземпляров `Counter` с помощью стандартных операторов, но поддерживаются также операции, выполняемые на месте с помощью операторов `+=`, `-=`, `&=` и `|=`.

Листинг 2.22. `collections_counter_arithmetic.py`

```
import collections

c1 = collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
c2 = collections.Counter('alphabet')
```

```
print('C1:', c1)
print('C2:', c2)

print('\nCombined counts:')
print(c1 + c2)

print('\nSubtraction:')
print(c1 - c2)

print('\nIntersection (taking positive minimums):')
print(c1 & c2)

print('\nUnion (taking maximums):')
print(c1 | c2)
```

При создании новых объектов `Counter` посредством выполнения одной из этих операций элементы с нулевыми и отрицательными значениями счетчиков отбрасываются. Счетчик для буквы `a` имеет одно и то же значение в экземплярах `c1` и `c2`, и поэтому она выпадает из их разности (`Subtraction`).

```
$ python3 collections_counter_arithmetic.py

C1: Counter({'b': 3, 'a': 2, 'c': 1})
C2: Counter({'a': 2, 'b': 1, 'p': 1, 't': 1, 'l': 1, 'e': 1,
'h': 1})

Combined counts:
Counter({'b': 4, 'a': 4, 'p': 1, 't': 1, 'c': 1, 'e': 1, 'l': 1,
'h': 1})

Subtraction:
Counter({'b': 2, 'c': 1})

Intersection (taking positive minimums):
Counter({'a': 2, 'b': 1})

Union (taking maximums):
Counter({'b': 3, 'a': 2, 'p': 1, 't': 1, 'c': 1, 'e': 1, 'l': 1,
'h': 1})
```

2.2.3. `defaultdict`: возврат значения по умолчанию для отсутствующего ключа

Стандартный тип словаря включает метод `setdefault()`, предназначенный для извлечения значения и установки значения по умолчанию при обращении к несуществующему ключу. В отличие от этого функция `defaultdict()` предоставляет вызывающему коду возможность заранее определить значение по умолчанию во время инициализации контейнера.

Листинг 2.23. collections_defaultdict.py

```
import collections


def default_factory():
    return 'default value'

d = collections.defaultdict(default_factory, foo='bar')
print('d:', d)
print('foo =>', d['foo'])
print('bar =>', d['bar'])
```

Этот метод хорошо работает при условии, что использование одного и того же значения по умолчанию подходит для всех ключей. Он особенно удобен, если значением по умолчанию является тип, используемый для агрегирования или аккумулирования таких значений, как списки, множества и даже целые числа. В документации стандартной библиотеки приведены примеры, в которых тип defaultdict используется именно таким способом.

```
$ python3 collections_defaultdict.py

d: defaultdict(<function default_factory at 0x101921950>,
{'foo': 'bar'})
foo => bar
bar => default value
```

Дополнительные ссылки

- defaultdict[4]. Примеры использования функции defaultdict, приведенные в документации стандартной библиотеки.
- *Evolution of Default Dictionaries in Python*[5] (James Tauber). Обсуждение того, как функция defaultdict() соотносится с другими средствами инициализации словарей.

2.2.4. deque: двухсторонняя очередь

Двухсторонняя очередь, deque, поддерживает добавление и удаление элементов на любом конце очереди. Более известные и чаще используемые стеки и очереди представляют собой вырожденные формы двухсторонних очередей, в которых входные и выходные данные ограничены одним концом очереди.

Листинг 2.24. collections_deque.py

```
import collections

d = collections.deque('abcdefg')
print('Deque:', d)
print('Length:', len(d))
print('Left end:', d[0])
```

[4] https://docs.python.org/3.5/library/collections.html#defaultdict-examples

[5] http://jtauber.com/blog/2008/02/27/evolution_of_default_dictionaries_in_python/

```
print('Right end:', d[-1])

d.remove('c')
print('remove(c):', d)
```

Поскольку двухсторонние очереди относятся к линейным контейнерам, в число поддерживаемых ими операций входят такие операции, поддерживаемые типом `list`, как получение содержимого с помощью метода `__getitem__()`, определение размера (длины) очереди и удаление первого вхождения значения.

```
$ python3 collections_deque.py

Deque: deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
Length: 7
Left end: a
Right end: g
remove(c): deque(['a', 'b', 'd', 'e', 'f', 'g'])
```

2.2.4.1. Добавление элементов

Двухсторонняя очередь может заполняться элементами на любом из ее концов, которые в терминологии Python называются "левым" и "правым" концами.

Листинг 2.25. `collections_deque_populating.py`

```
import collections

# Добавление справа
d1 = collections.deque()
d1.extend('abcdefg')
print('extend :', d1)
d1.append('h')
print('append :', d1)

# Добавление слева
d2 = collections.deque()
d2.extendleft(range(6))
print('extendleft:', d2)
d2.appendleft(6)
print('appendleft:', d2)
```

Метод `extendleft()` расширяет очередь, итерируя по всем элементам переданного ему итерируемого объекта и выполняя по отношению к каждому из них операцию, эквивалентную той, которую выполняет метод `appendleft()`. Конечным результатом является добавление в очередь всей входной последовательности в обратном порядке.

```
$ python3 collections_deque_populating.py

extend : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
append : deque(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
extendleft: deque([5, 4, 3, 2, 1, 0])
appendleft: deque([6, 5, 4, 3, 2, 1, 0])
```

2.2.4.2. Извлечение элементов

Точно так же, в зависимости от применяемого алгоритма, двухсторонняя очередь допускает извлечение элементов на любом из ее концов.

Листинг 2.26. `collections_deque_consuming.py`

```
import collections

print('From the right:')
d = collections.deque('abcdefg')
while True:
    try:
        print(d.pop(), end='')
    except IndexError:
        break
print

print('\nFrom the left:')
d = collections.deque(range(6))
while True:
    try:
        print(d.popleft(), end='')
    except IndexError:
        break
print
```

Метод `pop()` удаляет и возвращает элемент, находящийся на правом конце двухсторонней очереди; метод `popleft()` выполняет аналогичную операцию на левом конце очереди.

```
$ python3 collections_deque_consuming.py

From the right:
gfedcba
From the left:
012345
```

Поскольку двухсторонние очереди потокобезопасны, возможно одновременное извлечение элементов на обоих концах двухсторонней очереди с использованием независимых потоков.

Листинг 2.27. `collections_deque_both_ends.py`

```
import collections
import threading
import time

candle = collections.deque(range(5))


def burn(direction, nextSource):
    while True:
        try:
```

```
            next = nextSource()
        except IndexError:
            break
        else:
            print('{:>8}: {}'.format(direction, next))
            time.sleep(0.1)
    print('{:>8} done'.format(direction))
    return


left = threading.Thread(target=burn,
                        args=('Left', candle.popleft))
right = threading.Thread(target=burn,
                         args=('Right', candle.pop))

left.start()
right.start()

left.join()
right.join()
```

В этом примере потоки поочередно удаляют элементы на каждом из концов двухсторонней очереди, пока она не станет пустой.

```
$ python3 collections_deque_both_ends.py

 Left: 0
Right: 4
Right: 3
 Left: 1
Right: 2
 Left done
Right done
```

2.2.4.3. Прокрутка элементов

Еще одним полезным свойством двухсторонних очередей является возможность прокрутки элементов с целью пропуска некоторых из них.

Листинг 2.28: `collections_deque_rotate.py`

```
import collections
d = collections.deque(range(10))
print('Normal :', d)
d = collections.deque(range(10))
d.rotate(2)
print('Right rotation:', d)
d = collections.deque(range(10))
d.rotate(-2)
print('Left rotation :', d)
```

Прокрутка двухсторонней очереди вправо (положительное значение аргумента) приводит к удалению элементов на правом конце очереди и добавлению их на левом конце. Прокрутка влево (отрицательное значение аргумента) приводит

к удалению элементов на левом конце очереди и добавлению их на правом. В данном случае полезной визуальной аналогией этих процессов может послужить круговой циферблат.

```
$ python3 collections_deque_rotate.py

Normal        : deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Right rotation: deque([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
Left rotation : deque([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
```

2.2.4.4. Ограничение длины очереди

Размер экземпляра двухсторонней очереди можно ограничить так, чтобы ее длина никогда не превышала заданной величины. При достижении очередью указанной длины новые элементы добавляются за счет удаления существующих. Такое поведение очереди может быть использовано для нахождения последних n элементов потока неопределенной длины.

Листинг 2.29. collections_deque_maxlen.py

```
import collections
import random

# Установить для random затравочное значение, чтобы обеспечить
# получение одного и того же вывода при каждом запуске сценария
random.seed(1)

d1 = collections.deque(maxlen=3)
d2 = collections.deque(maxlen=3)

for i in range(5):
    n = random.randint(0, 100)
    print('n =', n)
    d1.append(n)
    d2.appendleft(n)
    print('D1:', d1)
    print('D2:', d2)
```

Длина двухсторонней очереди поддерживается постоянной независимо от того, на каком конце добавляются элементы.

```
$ python3 collections_deque_maxlen.py

n = 17
D1: deque([17], maxlen=3)
D2: deque([17], maxlen=3)
n = 72
D1: deque([17, 72], maxlen=3)
D2: deque([72, 17], maxlen=3)
n = 97
D1: deque([17, 72, 97], maxlen=3)
D2: deque([97, 72, 17], maxlen=3)
n = 8
```

```
D1: deque([72, 97, 8], maxlen=3)
D2: deque([8, 97, 72], maxlen=3)
n = 32
D1: deque([97, 8, 32], maxlen=3)
D2: deque([32, 8, 97], maxlen=3)
```

Дополнительные ссылки

- Википедия: *Двухсторонняя очередь*[6]. Обсуждение структуры двухсторонней очереди.
- *deque Recipes*[7]. Примеры использования двухсторонних очередей в алгоритмах, приведенные в документации стандартной библиотеки.

2.2.5. `namedtuple`: подкласс `Tuple` с именованными полями

Для доступа к элементам стандартного типа `tuple` используются числовые индексы.

Листинг 2.30. `collections_tuple.py`

```
bob = ('Bob', 30, 'male')
print('Representation:', bob)

jane = ('Jane', 29, 'female')
print('\nField by index:', jane[0])

print('\nFields by index:')
for p in [bob, jane]:
    print('{} is a {} year old {}'.format(*p))
```

Этот способ доступа удобно использовать в случае простых структур данных.

```
$ python3 collections_tuple.py

Representation: ('Bob', 30, 'male')

Field by index: Jane

Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female
```

Однако попытки запомнить, какой индекс следует использовать для каждого значения, могут приводить к ошибкам, особенно если многие поля кортежа конструируются вдали от тех мест кода, в которых они используются. Именованные кортежи обеспечивают возможность обращения к элементам не только по индексам, но и по именам.

[6] https://ru.wikipedia.org/wiki/Двухсторонняя_очередь

[7] https://docs.python.org/3.5/library/collections.html#deque-recipes

2.2.5.1. Определение

Ввиду того что экземпляры namedtuple не используют словарей, их эффективность в отношении использования памяти та же, что и у экземпляров обычных кортежей. Каждый тип именованного кортежа представляется собственным классом, который создается с помощью функции-фабрики namedtuple(). Аргументами этой функции являются имя нового класса и строка, содержащая имена элементов.

Листинг 2.31. collections_namedtuple_person.py

```
import collections

Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('\nRepresentation:', bob)

jane = Person(name='Jane', age=29)
print('\nField by name:', jane.name)

print('\nFields by index:')
for p in [bob, jane]:
    print('{} is {} years old'.format(*p))
```

Как продемонстрировано в этом примере, к полям именованных кортежей можно обращаться не только по их позиционным индексам, как в случае обычных кортежей, но и по именам с использованием точечной нотации (*объект.атрибут*).

```
$ python3 collections_namedtuple_person.py


Representation: Person(name='Bob', age=30)

Field by name: Jane

Fields by index:
Bob is 30 years old
Jane is 29 years old
```

Как и обычный тип tuple, тип namedtuple — неизменяемый. Благодаря этому экземпляры кортежа могут иметь хеш-значения, что позволяет использовать их в качестве ключей в словарях и включать в состав множеств.

Листинг 2.32. collections_namedtuple_immutable.py

```
import collections

Person = collections.namedtuple('Person', 'name age')

pat = Person(name='Pat', age=12)
print('\nRepresentation:', pat)

pat.age = 21
```

Попытки изменения значения посредством его именованного атрибута приводят к возбуждению исключения `AttributeError`.

```
$ python3 collections_namedtuple_immutable.py


Representation: Person(name='Pat', age=12)
Traceback (most recent call last):
  File "collections_namedtuple_immutable.py", line 17, in
<module>
    pat.age = 21
AttributeError: can't set attribute
```

2.2.5.2. Недопустимые имена полей

Допустимыми являются имена полей, которые не повторяются и не вступают в конфликт с ключевыми словами Python.

Листинг 2.33. `collections_namedtuple_bad_fields.py`

```
import collections

try:
    collections.namedtuple('Person', 'name class age')
except ValueError as err:
    print(err)

try:
    collections.namedtuple('Person', 'name age age')
except ValueError as err:
    print(err)
```

Если в процессе синтаксического разбора имен полей анализатор обнаруживает недопустимое имя, возбуждается исключение `ValueError`.

```
$ python3 collections_namedtuple_bad_fields.py

Type names and field names cannot be a keyword: 'class'
Encountered duplicate field name: 'age'
```

В ситуациях, когда именованный кортеж создается на основе значений, не контролируемых программой (например, если кортеж используется для представления строк, возвращаемых запросом к базе данных, схема которой не известна заранее), следует установить для опции `rename` значение `True`, обеспечивающее замену недопустимых имен.

Листинг 2.34. `collections_namedtuple_rename.py`

```
import collections

with_class = collections.namedtuple(
    'Person', 'name class age',
    rename=True)
```

```
print(with_class._fields)

two_ages = collections.namedtuple(
    'Person', 'name age age',
    rename=True)
print(two_ages._fields)
```

Новые имена для переименовываемых полей зависят от их индексов в кортеже, поэтому поле с именем `class` становится полем `_1`, а поле с повторяющимся именем `age` становится полем `_2`.

```
$ python3 collections_namedtuple_rename.py

('name', '_1', 'age')
('name', 'age', '_2')
```

2.2.5.3. Специальные атрибуты

Тип `namedtuple` предоставляет несколько полезных атрибутов и методов для работы с подклассами и экземплярами. Имена всех этих встроенных свойств снабжены префиксом в виде символа подчеркивания (_), который в соответствии с принимаемым в большинстве программ соглашением служит признаком частного (закрытого) атрибута. Однако в случае именованных кортежей этот префикс предназначен для того, чтобы не допустить конфликта с именами атрибутов, предоставляемыми пользователем.

Имена полей, переданных `namedtuple` для определения нового класса, сохраняются в атрибуте `_fields`.

Листинг 2.35. `collections_namedtuple_fields.py`

```
import collections

Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('Representation:', bob)
print('Fields:', bob._fields)
```

Несмотря на то что аргументом является единственная строка, в котором имена полей разделены пробелами, значение сохраняется в виде последовательности отдельных имен.

```
$ python3 collections_namedtuple_fields.py

Representation: Person(name='Bob', age=30)
Fields: ('name', 'age')
```

Экземпляры именованного кортежа `namedtuple` можно преобразовать в экземпляры упорядоченного словаря `OrderedDict` с помощью метода `_asdict()`.

Листинг 2.36. collections_namedtuple_asdict.py

```
import collections

Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('Representation:', bob)
print('As Dictionary:', bob._asdict())
```

Ключи OrderedDict располагаются в том же порядке, что и поля namedtuple.

```
$ python3 collections_namedtuple_asdict.py

Representation: Person(name='Bob', age=30)
As Dictionary: OrderedDict([('name', 'Bob'), ('age', 30)])
```

Метод _replace() позволяет создавать новые экземпляры на основе существующих с заменой значений некоторых полей.

Листинг 2.37. collections_namedtuple_replace.py

```
import collections

Person = collections.namedtuple('Person', 'name age')

bob = Person(name='Bob', age=30)
print('\nBefore:', bob)
bob2 = bob._replace(name='Robert')
print('After:', bob2)
print('Same?:', bob is bob2)
```

Может сложиться впечатление, будто в этом примере изменяется существующий объект, однако, поскольку экземпляры namedtuple неизменяемы, на самом деле возвращается новый объект.

```
$ python3 collections_namedtuple_replace.py

Before: Person(name='Bob', age=30)
After: Person(name='Robert', age=30)
Same?: False
```

2.2.6. OrderedDict: запоминание порядка добавляемых ключей

Класс OrderedDict — это подкласс словаря, запоминающий порядок добавления содержимого.

Листинг 2.38. collections_ordereddict_iter.py

```
import collections

print('Regular dictionary:')
```

```
d = {}
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'

for k, v in d.items():
    print(k, v)

print('\nOrderedDict:')
d = collections.OrderedDict()
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'

for k, v in d.items():
    print(k, v)
```

Обычный словарь не отслеживает порядок вставки элементов, и порядок получения значений при итерировании по словарю зависит от порядка сохранения ключей в хеш-таблице, который, в свою очередь, определяется случайными значениями, генерируемыми, чтобы уменьшить вероятность возникновения конфликтов. В противоположность этому порядок вставки элементов в словарь `OrderedDict` запоминается и используется во время создания итератора.

```
$ python3 collections_ordereddict_iter.py

Regular dictionary:
c C
b B
a A

OrderedDict:
a A
b B
c C
```

2.2.6.1. Равенство

При проверке равенства двух экземпляров `dict` сравнивается их содержимое. В случае экземпляров `OrderedDict` при этом учитывается также порядок вставки элементов.

Листинг 2.39. `collections_ordereddict_equality.py`

```
import collections
print('dict :', end=' ')
d1 = {}
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'
d2 = {}
d2['c'] = 'C'
d2['b'] = 'B'
```

```
d2['a'] = 'A'
print(d1 == d2)
print('OrderedDict:', end=' ')
d1 = collections.OrderedDict()
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'
d2 = collections.OrderedDict()
d2['c'] = 'C'
d2['b'] = 'B'
d2['a'] = 'A'
print(d1 == d2)
```

В этом примере упорядоченные словари, созданные с использованием различного порядка предоставления значений, оказываются разными, тогда как тестирование аналогичных обычных словарей подтверждает их равенство.

```
$ python3 collections_ordereddict_equality.py

dict : True
OrderedDict: False
```

2.2.6.2. Изменение порядка следования элементов

Тип `OrderedDict` предоставляет возможность изменения порядка следования в нем ключей посредством перемещения их в начало или в конец словаря с помощью метода `move_to_end()`.

Листинг 2.40. `collections_ordereddict_move_to_end.py`

```
import collections

d = collections.OrderedDict(
[('a', 'A'), ('b', 'B'), ('c', 'C')]
)

print('Before:')
for k, v in d.items():
    print(k, v)

d.move_to_end('b')

print('\nmove_to_end():')
for k, v in d.items():
    print(k, v)

d.move_to_end('b', last=False)

print('\nmove_to_end(last=False):')
for k, v in d.items():
    print(k, v)
```

Аргумент `last` метода `move_to_end()` задает перемещение элемента в конец последовательности ключей (значение `True`) или в начало (значение `False`).

```
$ python3 collections_ordereddict_move_to_end.py

Before:
a A
b B
c C

move_to_end():
a A
c C
b B

move_to_end(last=False):
b B
a A
c C
```

Дополнительные ссылки

- PYTHONHASHSEED[8]. Переменная среды, управляющая начальным случайным значением, которое вводится в алгоритм вычисления хеш-значения для определения местоположения ключа в словаре.

2.2.7. collections.abc: абстрактные базовые классы контейнеров

Модуль collections.abc содержит абстрактные базовые классы, определяющие API для контейнерных структур данных, встроенных в Python и предоставляемых модулем collections. Список классов вместе с описанием назначения каждого из них приведен в табл. 2.1.

Таблица 2.1. Абстрактные базовые классы

Класс	Базовый класс	Назначение API
Container		Предоставляет базовые возможности контейнера, такие как оператор in
Hashable		Добавляет поддержку хеш-значения для экземпляра контейнера
Iterable		Может создавать итератор по содержимому контейнера
Iterator	Iterable	Является итератором по содержимому контейнера
Generator	Iterator	Дополняет итераторы протоколом генератора из PEP 342
Sized		Добавляет методы для контейнеров, которым известен их размер
Callable		Предназначен для контейнеров, которые могут вызываться как функции

[8] https://docs.python.org/3.5/using/cmdline.html#envvar-PYTHONHASHSEED

Окончание табл. 2.1

Класс	Базовый класс	Назначение API
`Sequence`	`Sized, Iterable, Container`	Поддерживает извлечение элементов, итерирование и изменение порядка следования элементов
`MutableSequence`	`Sequence`	Поддерживает добавление и удаление элементов экземпляра после его создания
`ByteString`	`Sequence`	Объединенный API байтовых значений и байтовых массивов
`Set`	`Sized, Iterable, Container`	Поддерживает операции над множествами, такие как пересечение и объединение
`MutableSet`	`Set`	Добавляет методы, обеспечивающие манипулирование содержимым множества после его создания
`Mapping`	`Sized, Iterable, Container`	Определяет доступный только для чтения API, используемый типом `dict`
`MutableMapping`	`Mapping`	Добавляет методы, обеспечивающие манипулирование содержимым отображения после его создания
`MappingView`	`Sized`	Определяет API представления для доступа к отображению из итератора
`ItemsView`	`MappingView, Set`	Часть API представления
`KeysView`	`MappingView, Set`	Часть API представления
`ValuesView`	`MappingView`	Часть API представления
`Awaitable`		API для объектов, которые могут быть использованы в выражениях `await`, таких как сопрограммы
`Coroutine`	`Awaitable`	API для классов, реализующих протокол сопрограмм
`AsyncIterable`		API для итерируемых объектов, совместимых с циклами `async for`, в соответствии с документом PEP 492
`AsyncIterator`	`AsyncIterable`	API для асинхронных итераторов

Абстрактные базовые классы не только четко определяют API для контейнеров с различной семантикой, но и позволяют использовать метод `isinstance()` для тестирования поддержки объектом API еще до его вызова. Кроме того, некоторые классы предоставляют реализации методов и могут использоваться в качестве примесных классов для создания нестандартных типов контейнеров без необходимости реализации каждого метода с нуля.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `collections`[9].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `collections` (раздел A.6.8).
- **PEP 342**[10]. *Coroutines via Enhanced Generators.*
- **PEP 492**[11]. *Coroutines with `async` and `await` syntax.*

[9] `https://docs.python.org/3.5/library/collections.html`

[10] `www.python.org/dev/peps/pep-0342`

[11] `www.python.org/dev/peps/pep-0492`

2.3. array: последовательность данных фиксированного типа

Модуль array определяет линейную структуру данных, которая во многом напоминает список, за исключением того, что все члены этого списка должны относиться к одному и тому же элементарному типу. К поддерживаемым типам относятся все числовые и другие элементарные типы, такие как байты.

Некоторые из поддерживаемых типов приведены в табл. 2.2. Полный список кодов типов можно найти в документации стандартной библиотеки.

2.3.1. Инициализация

Массив инстанциализируется с передачей конструктору аргумента, задающего допустимый тип данных, и, возможно, начальной последовательности данных, сохраняемой в массиве.

Таблица 2.2. Коды типов для элементов массива

Код	Тип	Минимальный размер (байты)
b	Int	1
B	Int	1
h	Signed short	2
H	Unsigned short	2
i	Signed int	2
I	Unsigned int	2
l	Signed long	4
L	Unsigned long	4
q	Signed long long	8
Q	Unsigned long long	8
f	Float	4
d	Double float	8

Листинг 2.41. array_string.py

```
import array
import binascii

s = b'This is the array.'
a = array.array('b', s)

print('As byte string:', s)
print('As array :', a)
print('As hex :', binascii.hexlify(a))
```

В этом примере массив сконфигурирован для хранения последовательности байтов и инициализируется простой байтовой строкой.

```
$ python3 array_string.py

As byte string: b'This is the array.'
As array : array('b', [84, 104, 105, 115, 32, 105, 115, 32,
 116, 104, 101, 32, 97, 114, 114, 97, 121, 46])
As hex : b'546869732069732074686520617261792e'
```

2.3.2. Манипулирование массивами

К массивам применимы те же способы расширения и выполнения различных манипуляций, которые используются по отношению к другим последовательностям Python.

Листинг 2.42. `array_sequence.py`

```
import array
import pprint

a = array.array('i', range(3))

print('Initial :', a)

a.extend(range(3))
print('Extended:', a)

print('Slice :', a[2:5])

print('Iterator:')
print(list(enumerate(a)))
```

К поддерживаемым операциям относятся выделение срезов, итерирование и добавление элементов в конец массива.

```
$ python3 array_sequence.py

Initial : array('i', [0, 1, 2])
Extended: array('i', [0, 1, 2, 0, 1, 2])
Slice : array('i', [2, 0, 1])
Iterator:
[(0, 0), (1, 1), (2, 2), (3, 0), (4, 1), (5, 2)]
```

2.3.3. Массивы и файлы

Содержимое массива может записываться в файл и читаться из файла с помощью встроенных методов, оптимизированных для этих целей.

Листинг 2.43. `array_file.py`

```
import array
import binascii
import tempfile
```

```
a = array.array('i', range(5))
print('A1:', a)

# Запись массива чисел во временный файл
output = tempfile.NamedTemporaryFile()
a.tofile(output.file) # Нужно передать реальный файл
output.flush()

# Чтение "сырых" байтовых данных
with open(output.name, 'rb') as input:
    raw_data = input.read()
    print('Raw Contents:', binascii.hexlify(raw_data))

    # Чтение данных в массив
    input.seek(0)
    a2 = array.array('i')
    a2.fromfile(input, len(a))
    print('A2:', a2)
```

В этом примере чтение "сырых" данных, т.е. чтение данных непосредственно из двоичного файла, сравнивается с чтением данных в новый массив и преобразованием байтов в соответствующие типы.

```
$ python3 array_file.py

A1: array('i', [0, 1, 2, 3, 4])
Raw Contents: b'0000000001000000020000003000000040000000'
A2: array('i', [0, 1, 2, 3, 4])
```

Метод `tofile()` использует метод `tobytes()` для форматирования данных, а метод `fromfile()` использует метод `frombytes()` для их обратного преобразования в экземпляр массива.

Листинг 2.44. `array_tobytes.py`

```
import array
import binascii

a = array.array('i', range(5))
print('A1:', a)

as_bytes = a.tobytes()
print('Bytes:', binascii.hexlify(as_bytes))

a2 = array.array('i')
a2.frombytes(as_bytes)
print('A2:', a2)
```

Оба метода, `tobytes()` и `frombytes()`, работают с байтовыми строками, а не со строками Unicode.

```
$ python3 array_tobytes.py

A1: array('i', [0, 1, 2, 3, 4])
```

```
Bytes: b'00000000010000000200000003000000040000000'
A2: array('i', [0, 1, 2, 3, 4])
```

2.3.4. Альтернативные варианты порядка байтов

Если порядок следования байтов не совпадает с тем, который используется на данной платформе, или он должен быть изменен перед передачей данных системе с другим порядком байтов (либо по сети), можно преобразовать весь массив сразу, не выполняя итерирование по элементам массива.

Листинг 2.45. `array_byteswap.py`

```
import array
import binascii


def to_hex(a):
    chars_per_item = a.itemsize * 2  # две 16-ричные цифры
    hex_version = binascii.hexlify(a)
    num_chunks = len(hex_version) // chars_per_item
    for i in range(num_chunks):
        start = i * chars_per_item
        end = start + chars_per_item
        yield hex_version[start:end]


start = int('0x12345678', 16)
end = start + 5
a1 = array.array('i', range(start, end))
a2 = array.array('i', range(start, end))
a2.byteswap()

fmt = '{:>12} {:>12} {:>12} {:>12}'
print(fmt.format('A1 hex', 'A1', 'A2 hex', 'A2'))
print(fmt.format('-' * 12, '-' * 12, '-' * 12, '-' * 12))
fmt = '{!r:>12} {:12} {!r:>12} {:12}'
for values in zip(to_hex(a1), a1, to_hex(a2), a2):
    print(fmt.format(*values))
```

Метод `byteswap()` изменяет порядок следования байтов в элементах массива, используя код, написанный на языке C и поэтому работающий намного эффективнее по сравнению с обработкой данных в цикле с помощью кода на языке Python.

```
$ python3 array_byteswap.py
      A1 hex           A1       A2 hex           A2
------------ ------------ ------------ ------------
 b'78563412'    305419896  b'12345678'   2018915346
 b'79563412'    305419897  b'12345679'   2035692562
 b'7a563412'    305419898  b'1234567a'   2052469778
 b'7b563412'    305419899  b'1234567b'   2069246994
 b'7c563412'    305419900  b'1234567c'   2086024210
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `array`[12].
- `struct` (раздел 2.7). Описание модуля `struct`.
- Python для численных расчетов[13]. NumPy — это библиотека Python, предназначенная для эффективной обработки больших наборов данных.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `array` (раздел A.6.4).

2.4. heapq: алгоритм сортировки кучи

Куча — это древовидная структура данных, в которой между дочерними и родительскими узами установлены отношения порядка сортировки. Двоичная куча может быть представлена списком или массивом, организованным таким образом, что дочерние узлы элемента N находятся в позициях $2*N+1$ и $2*N+2$ (индексы отсчитываются от нуля). Такая компоновка делает возможной реорганизацию кучи на месте, что избавляет от необходимости перераспределения больших объемов памяти при добавлении или удалении элементов.

В кучах *max-heap* гарантируется, что значение в любом родительском узле не может быть меньшим, чем значение в любом из двух его дочерних узлов. В кучах *min-heap* выполняется противоположное условие: значение в любом родительском узле не может быть большим, чем значение в любом из двух его дочерних узлов. Модуль `heapq` в Python реализует кучу *min-heap*.

2.4.1. Данные для примеров

В примерах, приведенных в этом разделе, используются данные, хранящиеся в файле `heapq_heapdata.py`.

Листинг 2.46. `heapq_heapdata.py`

```
# Эти данные сгенерированы с помощью модуля random
data = [19, 9, 4, 10, 11]
```

Для вывода данных кучи используется файл `heapq_showtree.py`.

Листинг 2.47. `heapq_showtree.py`

```
import math
from io import StringIO


def show_tree(tree, total_width=36, fill=' '):
    """Красивый вывод дерева."""
    output = StringIO()
    last_row = -1
    for i, n in enumerate(tree):
        if i:
            row = int(math.floor(math.log(i + 1, 2)))
```

[12] https://docs.python.org/3.5/library/array.html

[13] www.scipy.org

```
        else:
            row = 0
        if row != last_row:
            output.write('\n')
        columns = 2 ** row
        col_width = int(math.floor(total_width / columns))
        output.write(str(n).center(col_width, fill))
        last_row = row
    print(output.getvalue())
    print('-' * total_width)
    print()
```

2.4.2. Создание кучи

Существуют два основных способа создания кучи: с помощью методов `heappush()` и `heapify()`.

Листинг 2.48. `heapq_heappush.py`

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

heap = []
print('random :', data)
print()

for n in data:
    print('add {:>3}:'.format(n))
    heapq.heappush(heap, n)
    show_tree(heap)
```

При добавлении в кучу новых элементов из источника данных с помощью метода `heappush()` в ней сохраняется установленный для нее порядок сортировки.

```
$ python3 heapq_heappush.py

random : [19, 9, 4, 10, 11]

add  19:

                 19
-----------------------------------

add   9:

                 9
        19
-----------------------------------

add   4:

                 4
```

```
        19                  9
-----------------------------------

add  10:

                    4
        10                  9
    19
-----------------------------------

add  11:

                    4
        10                  9
    19      11
-----------------------------------
```

Если данные уже находятся в памяти, то более эффективным способом перестановки элементов списка на месте является использование метода `heapify()`.

Листинг 2.49. `heapq_heapify.py`

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print('random :', data)
heapq.heapify(data)
print('heapified :')
show_tree(data)
```

Создание списка путем добавления по одному элементу за раз с соблюдением установленного для кучи порядка сортировки приводит к тому же результату, что и создание неупорядоченного списка с последующим вызовом метода `heapify()`.

```
$ python3 heapq_heapify.py

random    : [19, 9, 4, 10, 11]
heapified :

                    4
        9                   19
    10      11
-----------------------------------
```

2.4.3. Доступ к содержимому кучи

Для удаления элемента с наименьшим значением из корректно организованной кучи используется метод `heappop()`.

Листинг 2.50. heapq_heappop.py

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data

print('random :', data)
heapq.heapify(data)
print('heapified :')
show_tree(data)
print

for i in range(2):
    smallest = heapq.heappop(data)
    print('pop {:>3}:'.format(smallest))
    show_tree(data)
```

В этом примере, адаптированном на основе документации стандартной библиотеки, методы `heapify()` и `heappop()` используются для сортировки числового списка.

```
$ python3 heapq_heappop.py

random    : [19, 9, 4, 10, 11]
heapified :

                4
        9                19
    10      11
-----------------------------------


pop      4:

                9
        10               19
    11
-----------------------------------

pop      9:

                10
        11               19
-----------------------------------
```

Для удаления существующих элементов и замены их новыми значениями в рамках одной операции используйте метод `heapreplace()`.

Листинг 2.51. heapq_heapreplace.py

```
import heapq
from heapq_showtree import show_tree
from heapq_heapdata import data
```

```
heapq.heapify(data)
print('start:')
show_tree(data)

for n in [0, 13]:
    smallest = heapq.heapreplace(data, n)
    print('replace {:>2} with {:>2}:'.format(smallest, n))
    show_tree(data)
```

Замена элементов на месте делает возможным поддержание постоянного размера кучи, как, например, в случае очереди заданий с приоритетом.

```
$ python3 heapq_heapreplace.py

start:

                4
        9                19
    10      11
------------------------------------

replace  4 with  0:

                0
        9                19
    10      11
------------------------------------

replace  0 with 13:

                9
        10               19
    13      11
------------------------------------
```

2.4.4. Получение наибольших и наименьших элементов кучи

Модуль heapq предоставляет две функции, позволяющие находить диапазоны наибольших и наименьших значений, содержащихся в итерируемом объекте.

Листинг 2.52. heapq_extremes.py

```
import heapq
from heapq_heapdata import data

print('all       :', data)
print('3 largest :', heapq.nlargest(3, data))
print('from sort :', list(reversed(sorted(data)[-3:])))
print('3 smallest:', heapq.nsmallest(3, data))
print('from sort :', sorted(data)[:3])
```

Использование функций `nlargest()` и `nsmallest()` наиболее эффективно лишь при относительно небольших значениях $n > 1$, однако в некоторых ситуациях они могут быть весьма кстати.

```
$ python3 heapq_extremes.py

all       : [19, 9, 4, 10, 11]
3 largest : [19, 11, 10]
from sort : [19, 11, 10]
3 smallest: [4, 9, 10]
from sort : [4, 9, 10]
```

2.4.5. Эффективное слияние отсортированных последовательностей

В случае небольших наборов данных объединение двух отсортированных последовательностей в одну не составляет большого труда.

```
list(sorted(itertools.chain(*data)))
```

Однако в случае крупных наборов данных этот прием может потребовать использования больших объемов памяти. Вместо сортировки всей объединенной последовательности метод `merge()` использует кучу для генерации новой последовательности по одному элементу за раз, используя для определения очередного элемента память фиксированного объема.

Листинг 2.53. `heapq_merge.py`

```
import heapq
import random

random.seed(2016)

data = []
for i in range(4):
    new_data = list(random.sample(range(1, 101), 5))
    new_data.sort()
    data.append(new_data)

for i, d in enumerate(data):
    print('{}: {}'.format(i, d))

print('\nMerged:')
for i in heapq.merge(*data):
    print(i, end=' ')
print()
```

Поскольку реализация метода `merge()` использует кучу, расход памяти зависит от количества объединяемых последовательностей, а не от количества элементов в каждой из них.

```
$ python3 heapq_merge.py

0: [33, 58, 71, 88, 95]
1: [10, 11, 17, 38, 91]
2: [13, 18, 39, 61, 63]
3: [20, 27, 31, 42, 45]

Merged:
10 11 13 17 18 20 27 31 33 38 39 42 45 58 61 63 71 88 91 95
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `heapq`[14].
- Википедия: *Куча (структура данных)*[15]. Общее описание структур данных типа кучи.
- Раздел 2.6.3. Реализация очередей с приоритетом из класса `Queue` стандартной библиотеки.

2.5. `bisect`: поддержание сортированного состояния списков

Модуль `bisect` реализует алгоритм вставки элементов в список, обеспечивающий поддержание сортированного состояния списка.

2.5.1. Сортировка при вставке

Ниже приведен простой пример, в котором список сортируется при вставке элементов с помощью метода `insort()`.

Листинг 2.54. `bisect_example.py`

```
import bisect

# Последовательность случайных чисел
values = [14, 85, 77, 26, 50, 45, 66, 79, 10, 3, 84, 77, 1]

print('New Pos Contents')
print('--- --- --------')

l = []
for i in values:
    position = bisect.bisect(l, i)
    bisect.insort(l, i)
    print('{:3} {:3}'.format(i, position), l)
```

В первом столбце вывода отображается новое случайное число, а во втором — позиция, в которую вставляется это число. Остальная часть каждой строки представляет текущее состояние отсортированного списка.

[14] https://docs.python.org/3.5/library/heapq.html

[15] https://ru.wikipedia.org/wiki/Куча_(структура_данных)

```
New  Pos  Contents
---  ---  --------
 14    0 [14]
 85    1 [14, 85]
 77    1 [14, 77, 85]
 26    1 [14, 26, 77, 85]
 50    2 [14, 26, 50, 77, 85]
 45    2 [14, 26, 45, 50, 77, 85]
 66    4 [14, 26, 45, 50, 66, 77, 85]
 79    6 [14, 26, 45, 50, 66, 77, 79, 85]
 10    0 [10, 14, 26, 45, 50, 66, 77, 79, 85]
  3    0 [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
 84    9 [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
 77    8 [3, 10, 14, 26, 45, 50, 66, 77, 77, 79, 84, 85]
  1    0 [1, 3, 10, 14, 26, 45, 50, 66, 77, 77, 79, 84, 85]
```

Это довольно простой пример. В действительности при небольших объемах данных быстрее и проще создать список и отсортировать его. Однако в случае длинных списков алгоритм сортировки при вставке обеспечивает значительную экономию времени и памяти, особенно если операция сравнения двух элементов списка требует проведения интенсивных вычислений.

2.5.2. Обработка повторяющихся значений

Представленный перед этим результирующий набор включает повторяющееся значение (77). Модуль `bisect` обеспечивает два способа обработки повторяющихся значений: новые значения могут вставляться либо слева от существующих, либо справа. В действительности функция `insort()` является псевдонимом функции `insort_right()`, которая вставляет элемент после существующего значения. Соответствующая ей функция `insort_left()` вставляет элемент перед существующим значением.

Листинг 2.55. `bisect_example2.py`

```
import bisect

# Последовательность случайных чисел
values = [14, 85, 77, 26, 50, 45, 66, 79, 10, 3, 84, 77, 1]

print('New Pos Contents')
print('--- --- --------')

# Используются функции bisect_left() и insort_left()
l = []
for i in values:
    position = bisect.bisect_left(l, i)
    bisect.insort_left(l, i)
    print('{:3} {:3}'.format(i, position), l)
```

Применение функций `bisect_left()` и `insort_left()` приводит к тому же результирующему отсортированному списку, но с другими позициями для повторяющихся значений.

```
New  Pos  Contents
---  ---  --------
 14    0 [14]
 85    1 [14, 85]
 77    1 [14, 77, 85]
 26    1 [14, 26, 77, 85]
 50    2 [14, 26, 50, 77, 85]
 45    2 [14, 26, 45, 50, 77, 85]
 66    4 [14, 26, 45, 50, 66, 77, 85]
 79    6 [14, 26, 45, 50, 66, 77, 79, 85]
 10    0 [10, 14, 26, 45, 50, 66, 77, 79, 85]
  3    0 [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
 84    9 [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
 77    7 [3, 10, 14, 26, 45, 50, 66, 77, 77, 79, 84, 85]
  1    0 [1, 3, 10, 14, 26, 45, 50, 66, 77, 77, 79, 84, 85]
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `bisect`[16].
- Википедия: *Сортировка вставками*[17]. Описание алгоритма сортировки при вставке.

2.6. queue: потокобезопасная реализация очереди FIFO

Модуль `queue` предоставляет структуру данных с дисциплиной обслуживания FIFO (first in, first out — "первым пришел — первым ушел"), пригодную для многопоточного программирования. Ее можно использовать для безопасного обмена данными между потоками-производителями и потоками-потребителями. Для вызывающего кода предусмотрена обработка блокировок, поэтому с одним экземпляром `Queue` может безопасно работать несколько потоков. Размер экземпляра `Queue` (количество содержащихся в нем элементов) можно ограничивать.

Примечание

В ходе дальнейшего обсуждения предполагается, что читатель уже знаком с понятием очереди. Если это не так, рекомендуется предварительно изучить эту тему, воспользовавшись другими доступными источниками, прежде чем продолжить чтение.

2.6.1. Базовая очередь FIFO

Класс `Queue` реализует базовый контейнер, работающий по принципу FIFO. Элементы добавляются на одном "конце" последовательности с помощью метода `put()` и удаляются на другом с помощью метода `get()`.

[16] https://docs.python.org/3.5/library/bisect.html

[17] https://ru.wikipedia.org/wiki/Сортировка_вставками

Листинг 2.56. queue_fifo.py

```
import queue

q = queue.Queue()

for i in range(5):
    q.put(i)

while not q.empty():
    print(q.get(), end=' ')
print()
```

В этом примере для демонстрации того, что элементы удаляются из очереди в том порядке, в котором они добавлялись, используется один поток.

```
$ python3 queue_fifo.py

0 1 2 3 4
```

2.6.2. Очередь LIFO

В отличие от стандартной реализации очереди `Queue`, действующей по принципу FIFO, очередь `LifoQueue` обеспечивает дисциплину обслуживания LIFO (last in, first out – "последним пришел – первым ушел"), которая обычно ассоциируется со структурами данных типа стека.

Листинг 2.57. queue_lifo.py

```
import queue

q = queue.LifoQueue()

for i in range(5):
    q.put(i)

while not q.empty():
    print(q.get(), end=' ')
print()
```

Метод `get()` удаляет из очереди элемент, который был помещен в нее последним.

```
$ python3 queue_lifo.py

4 3 2 1 0
```

2.6.3. Очередь с приоритетом

Иногда порядок обработки элементов очереди должен основываться не на последовательности их создания или добавления в очередь, а на их характеристиках. Например, документы, подготовленные бухгалтерией, могут иметь бо-

лее высокий приоритет в очереди заданий для вывода на печать по сравнению с листингом программы, подготовленным разработчиком. Класс `PriorityQueue` использует упорядочение очереди по приоритетам для принятия решения относительно того, какой из элементов следует извлечь.

Листинг 2.58. `queue_priority.py`

```
import functools
import queue
import threading


@functools.total_ordering
class Job:

    def __init__(self, priority, description):
        self.priority = priority
        self.description = description
        print('New job:', description)
        return

    def __eq__(self, other):
        try:
            return self.priority == other.priority
        except AttributeError:
            return NotImplemented

    def __lt__(self, other):
        try:
            return self.priority < other.priority
        except AttributeError:
            return NotImplemented


q = queue.PriorityQueue()

q.put(Job(3, 'Mid-level job'))
q.put(Job(10, 'Low-level job'))
q.put(Job(1, 'Important job'))


def process_job(q):
    while True:
        next_job = q.get()
        print('Processing job:', next_job.description)
        q.task_done()


workers = [
    threading.Thread(target=process_job, args=(q,)),
    threading.Thread(target=process_job, args=(q,)),
]
for w in workers:
```

```
    w.setDaemon(True)
    w.start()

q.join()
```

В этом примере задания обрабатываются несколькими потоками на основании приоритета элементов очереди во время вызова функции `get()`. Порядок обработки элементов, добавляемых в очередь во время выполнения потоков-потребителей, зависит от контекста переключения потоков.

```
$ python3 queue_priority.py

New job: Mid-level job
New job: Low-level job
New job: Important job
Processing job: Important job
Processing job: Mid-level job
Processing job: Low-level job
```

2.6.4. Создание многопоточного подкаст-клиента

Приведенный в этом разделе исходный код подкаст-клиента демонстрирует использование класса `Queue` с несколькими потоками. Программа читает один или несколько RSS-каналов, создает очередь вложений для пяти самых последних эпизодов из каждого канала, подлежащих скачиванию, и обрабатывает одновременно несколько скачиваний, используя потоки. Уровень обработки ошибок в программе недостаточно высок для ее промышленного использования, однако данная каркасная реализация хорошо иллюстрирует возможности модуля `queue`. (Возможно, вам потребуется установить пакет `feedparser` с помощью команды **`pip3 install feedparser`**. — *Примеч.ред.*)

Прежде всего в программе устанавливаются некоторые рабочие параметры. Обычно они предоставляются пользователем (например, в виде предпочтений или посредством базы данных). В данном примере количество потоков и список URL-адресов для скачивания заданы непосредственно в коде.

Листинг 2.59. `fetch_podcasts.py`

```
from queue import Queue
import threading
import time
import urllib
from urllib.parse import urlparse

import feedparser

# Установка некоторых глобальных переменных
num_fetch_threads = 2
enclosure_queue = Queue()

# В реальном приложении вы не будете задавать данные в коде...
feed_urls = [
```

```
    'http://talkpython.fm/episodes/rss',
]


def message(s):
    print('{}: {}'.format(threading.current_thread().name, s))
```

Функция download_enclosures() выполняется в рабочем потоке и обрабатывает закачки с помощью модуля urllib.

```
def download_enclosures(q):
    """Это функция рабочего потока.
    Она обрабатывает элементы очереди один за другим.
    Потоки этого процесса-демона входят в бесконечный
    цикл и завершаются, только когда завершается
    основной поток.
    """
    while True:
        message('looking for the next enclosure')
        url = q.get()
        filename = url.rpartition('/')[-1]
        message('downloading {}'.format(filename))
        response = urllib.request.urlopen(url)
        data = response.read()
        # Сохранить загруженный файл в текущем каталоге
        message('writing to {}'.format(filename))
        with open(filename, 'wb') as outfile:
            outfile.write(data)
        q.task_done()
```

Коль скоро для потоков определена целевая функция, можно запускать рабочие потоки. Когда функция download_enclosures() обрабатывает инструкцию url=q.get(), она блокируется и ожидает возврата из какого-либо потока. Это означает, что можно безопасно запускать потоки, даже если очередь пуста.

```
# Настройка потоков для извлечения вложений
for i in range(num_fetch_threads):
    worker = threading.Thread(
        target=download_enclosures,
        args=(enclosure_queue,),
        name='worker-{}'.format(i),
    )
    worker.setDaemon(True)
    worker.start()
```

Следующим шагом является извлечение содержимого канала с помощью модуля feedparser и помещение URL-адресов вложений в очередь. Как только в очередь добавляется первый URL-адрес, один из рабочих потоков выбирает его и запускает загрузку. Цикл продолжает добавлять элементы до исчерпания канала, тогда как рабочие потоки извлекают URL-адреса из очереди для загрузки содержимого.

```
# Загрузка каналов и помещение URL-адресов вложений в очередь
for url in feed_urls:
    response = feedparser.parse(url, agent='fetch_podcasts.py')
    for entry in response['entries'][:5]:
        for enclosure in entry.get('enclosures', []):
            parsed_url = urlparse(enclosure['url'])
            message('queuing {}'.format(
                parsed_url.path.rpartition('/')[-1]))
            enclosure_queue.put(enclosure['url'])
```

Теперь единственное, что остается сделать, — это дождаться исчерпания очереди, используя функцию `join()`.

```
# Дождаться исчерпания очереди, что будет свидетельствовать
# о завершении обработки всех закачек
message('*** main thread waiting')
enclosure_queue.join()
message('*** done')
```

Выполнив сценарий, вы должны получить примерно следующий вывод.

```
$ python3 fetch_podcasts.py

worker-0: looking for the next enclosure
worker-1: looking for the next enclosure
MainThread: queuing turbogears-and-the-future-of-python-web-
frameworks.mp3
MainThread: queuing continuum-scientific-python-and-the-business-
of-open-source.mp3
MainThread: queuing openstack-cloud-computing-built-on-python.mp3
MainThread: queuing pypy.js-pypy-python-in-your-browser.mp3
MainThread: queuing machine-learning-with-python-and-scikit-
learn.mp3
MainThread: *** main thread waiting
worker-0: downloading turbogears-and-the-future-of-python-web-
frameworks.mp3
worker-1: downloading continuum-scientific-python-and-the-
business-of-open-source.mp3
worker-0: looking for the next enclosure
worker-0: downloading openstack-cloud-computing-built-on-
python.mp3
worker-1: looking for the next enclosure
worker-1: downloading pypy.js-pypy-python-in-your-browser.mp3
worker-0: looking for the next enclosure
worker-0: downloading machine-learning-with-python-and-scikit-
learn.mp3
worker-1: looking for the next enclosure
worker-0: looking for the next enclosure
MainThread: *** done
```

Фактический вывод будет зависеть от содержимого используемого RSS-канала.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `queue`[18].
- `deque`. Двухсторонняя очередь (раздел 2.2.4) из модуля `collections` (раздел 2.2).
- Википедия: *Очередь (программирование)*[19]. Статья с общим обзором очередей.
- Википедия: *FIFO*[20]. Статья, посвященная структурам данных с дисциплиной обслуживания FIFO.
- `feedparser`[21]. Описание модуля, предназначенного для парсинга каналов в форматах RSS и Atom (разработчик — Марк Пилгрим, сопровождение — Курт Макки).

2.7. `struct`: структуры двоичных данных

Модуль `struct` включает функции, предназначенные для выполнения преобразований между строками байтов и собственными типами данных Python, такими как числа и строки.

2.7.1. Функции уровня модуля и класс `Struct`

Для работы со структурированными значениями в модуле `struct` предусмотрены функции уровня модуля и класс `Struct`. Спецификаторы формата преобразуются из их строкового формата в компилированное представление аналогично тому, как это делается при обработке регулярных выражений. Это преобразование потребляет определенные ресурсы, поэтому обычно более эффективно выполнить его один раз при создании экземпляра `Struct` и вызывать методы для экземпляра, чем использовать функции уровня модуля. Во всех нижеследующих примерах используется класс `Struct`.

2.7.2. Упаковка и распаковка

Структуры поддерживают операции *упаковки* данных в строки и обратной их *распаковки* с помощью спецификаторов формата, представляющих тип данных, а также необязательные счетчик байтов и индикатор порядка следования байтов. Полный список поддерживаемых спецификаторов формата можно найти в разделе документации стандартной библиотеки, посвященном модулю `struct`.

В этом примере спецификатор требует предоставления целого или длинного целого значения, двухбайтовой строки и числа с плавающей точкой. Пробелы включены в спецификатор формата для разделения указателей типа и игнорируются в процессе компиляции.

Листинг 2.60. `struct_pack.py`

```
import struct
import binascii

values = (1, 'ab'.encode('utf-8'), 2.7)
```

[18] https://docs.python.org/3.5/library/queue.html

[19] https://ru.wikipedia.org/wiki/Очередь_(программирование)

[20] https://ru.wikipedia.org/wiki/FIFO

[21] https://pypi.python.org/pypi/feedparser

```
s = struct.Struct('I 2s f')
packed_data = s.pack(*values)

print('Original values:', values)
print('Format string :', s.format)
print('Uses :', s.size, 'bytes')
print('Packed Value :', binascii.hexlify(packed_data))
```

В данном случае упакованное значение преобразуется в последовательность шестнадцатеричных байтов с помощью функции `binascii.hexlify()`, поскольку некоторые из символов являются нулями.

```
$ python3 struct_pack.py

Original values: (1, b'ab', 2.7)
Format string  : b'I 2s f'
Uses           : 12 bytes
Packed Value   : b'0100000061620000cdcc2c40'
```

Для извлечения данных из их упакованного представления используйте функцию `unpack()`.

Листинг 2.61. `struct_unpack.py`

```
import structimport binascii
packed_data = binascii.unhexlify(b'0100000061620000cdcc2c40')
s = struct.Struct('I 2s f')
unpacked_data = s.unpack(packed_data)
print('Unpacked Values:', unpacked_data)
```

Возвращаемые методом `unpack()` данные, по сути, совпадают с исходными (обратите внимание на расхождение для значения с плавающей точкой).

```
$ python3 struct_unpack.py

Unpacked Values: (1, b'ab', 2.700000047683716)
```

2.7.3. Индикатор порядка байтов

По умолчанию значения кодируются с использованием порядка следования байтов, установленного для библиотеки C. Однако вы можете изменить этот порядок, явно задав соответствующую директиву в строке формата.

Таблица 2.3. Спецификаторы порядка следования байтов для модуля `Struct`

Код	Порядок следования байтов
@	Свойственный данной аппаратной платформе
=	Свойственный данной аппаратной платформе, стандартные размеры и выравнивание
<	Обратный порядок
>	Прямой порядок
!	Сетевой порядок

Листинг 2.62. struct_endianness.py

```
import struct
import binascii

values = (1, 'ab'.encode('utf-8'), 2.7)
print('Original values:', values)

endianness = [
    ('@', 'native, native'),
    ('=', 'native, standard'),
    ('<', 'little-endian'),
    ('>', 'big-endian'),
    ('!', 'network'),
]

for code, name in endianness:
    s = struct.Struct(code + ' I 2s f')
    packed_data = s.pack(*values)
    print()
    print('Format string  :', s.format, 'for', name)
    print('Uses           :', s.size, 'bytes')
    print('Packed Value   :', binascii.hexlify(packed_data))
    print('Unpacked Value :', s.unpack(packed_data))
```

Спецификаторы порядка следования байтов, используемые модулем Struct, приведены в табл. 2.3.

```
$ python3 struct_endianness.py
Original values: (1, b'ab', 2.7)

Format string  : b'@ I 2s f' for native, native
Uses           : 12 bytes
Packed Value   : b'0100000061620000cdcc2c40'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : b'= I 2s f' for native, standard
Uses           : 10 bytes
Packed Value   : b'010000006162cdcc2c40'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : b'< I 2s f' for little-endian
Uses           : 10 bytes
Packed Value   : b'010000006162cdcc2c40'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : b'> I 2s f' for big-endian
Uses           : 10 bytes
Packed Value   : b'000000016162402ccccd'
Unpacked Value : (1, b'ab', 2.700000047683716)

Format string  : b'! I 2s f' for network
Uses           : 10 bytes
```

```
Packed Value   : b'000000016162402ccccd'
Unpacked Value : (1, b'ab', 2.700000047683716)
```

2.7.4. Буферизация

Как правило, упакованные двоичные данные применяются в ситуациях, в которых требуется как можно более высокая производительность или возникает необходимость в обмене данными с модулями расширения. В подобных случаях возможно дополнительное повышение производительности за счет отказа от выделения нового буфера для каждой упакованной структуры. Методы `pack_into()` и `unpack_from()` поддерживают непосредственную запись в предварительно выделенные буфера.

Листинг 2.63. `struct_buffers.py`

```
import array
import binascii
import ctypes
import struct

s = struct.Struct('I 2s f')
values = (1, 'ab'.encode('utf-8'), 2.7)
print('Original:', values)

print()
print('ctypes string buffer')

b = ctypes.create_string_buffer(s.size)
print('Before :', binascii.hexlify(b.raw))
s.pack_into(b, 0, *values)
print('After :', binascii.hexlify(b.raw))
print('Unpacked:', s.unpack_from(b, 0))

print()
print('array')

a = array.array('b', b'\0' * s.size)

print('Before :', binascii.hexlify(a))
s.pack_into(a, 0, *values)
print('After :', binascii.hexlify(a))
print('Unpacked:', s.unpack_from(a, 0))
```

Атрибут `size` сообщает о необходимом размере буфера.

```
$ python3 struct_buffers.py

Original: (1, b'ab', 2.7)

ctypes string buffer
Before : b'000000000000000000000000'
After : b'01000000616200000cdcc2c40'
```

```
Unpacked: (1, b'ab', 2.700000047683716)

array
Before : b'000000000000000000000000'
After : b'0100000061620000cdcc2c40'
Unpacked: (1, b'ab', 2.700000047683716)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `struct`[22].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `struct` (раздел A.6.42).
- `array` (раздел 2.3). Описание модуля `array`, предназначенного для работы с последовательностями значений фиксированного типа.
- `Binascii`. Модуль `binascii` предназначен для получения ASCII-представления двоичных данных.
- Википедия: *Порядок байтов*[23]. Описание порядка следования байтов, используемого при кодировании данных.

2.8. `weakref`: слабые ссылки на объекты

Модуль `weakref` обеспечивает поддержку слабых ссылок на объекты. Создание обычной ссылки приводит к увеличению счетчика ссылок на объект, что препятствует его удалению сборщиком мусора. Такое поведение не всегда желательно, особенно в ситуациях, когда имеются циклические ссылки или требуется освободить кеш-память, занимаемую объектами. Особенностью слабой ссылки является то, что она позволяет ссылаться на объект, не препятствуя его автоматическому удалению.

2.8.1. Ссылки

Со слабыми ссылками работают через класс `ref`. Чтобы обратиться к исходному объекту, следует вызвать объект ссылки.

Листинг 2.64. `weakref_ref.py`

```
import weakref


class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))


obj = ExpensiveObject()
r = weakref.ref(obj)
```

[22] https://docs.python.org/3.5/library/struct.html

[23] https://ru.wikipedia.org/wiki/Порядок_байтов

```
print('obj:', obj)
print('ref:', r)
print('r():', r())

print('deleting obj')
del obj
print('r():', r())
```

В данном случае второй вызов объекта ссылки, `r()`, возвращает значение `None`, поскольку к этому времени исходный объект `obj` уже был удален.

```
$ python3 weakref_ref.py

obj: <__main__.ExpensiveObject object at 0x1007b1a58>
ref: <weakref at 0x1007a92c8; to 'ExpensiveObject' at
0x1007b1a58>
r(): <__main__.ExpensiveObject object at 0x1007b1a58>
deleting obj
(Deleting <__main__.ExpensiveObject object at 0x1007b1a58>)
r(): None
```

2.8.2. Функции обратного вызова в слабых ссылках

Конструктор `ref` поддерживает необязательный аргумент в виде функции обратного вызова, которая будет автоматически вызываться при удалении исходного объекта.

Листинг 2.65. `weakref_ref_callback.py`

```
import weakref


class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))


def callback(reference):
    """Вызывается при удалении целевого объекта"""
    print('callback({!r})'.format(reference))


obj = ExpensiveObject()
r = weakref.ref(obj, callback)

print('obj:', obj)
print('ref:', r)
print('r():', r())

print('deleting obj')
del obj
print('r():', r())
```

Функция обратного вызова получает объект ссылки в качестве аргумента, когда ссылка становится недействительной по причине удаления объекта. Одним из возможных применений этого средства является удаление объекта слабой ссылки из кеша.

```
$ python3 weakref_ref_callback.py

obj: <__main__.ExpensiveObject object at 0x1010b1978>
ref: <weakref at 0x1010a92c8; to 'ExpensiveObject' at
0x1010b1978>
r(): <__main__.ExpensiveObject object at 0x1010b1978>
deleting obj
(Deleting <__main__.ExpensiveObject object at 0x1010b1978>)
callback(<weakref at 0x1010a92c8; dead>)
r(): None
```

2.8.3. Завершающие операции при удалении объектов

Надежность управления ресурсами при удалении слабых ссылок можно повысить, связывая с объектами функции обратного вызова при помощи функции `finalize()`. Экземпляр `finalize` (объект-финализатор) удерживается в памяти до тех пор, пока не будет удален связанный с ним объект, даже если в приложении отсутствуют ссылки на объект-финализатор.

Листинг 2.66. `weakref_finalize.py`

```
import weakref


class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

def on_finalize(*args):
    print('on_finalize({!r})'.format(args))

obj = ExpensiveObject()
weakref.finalize(obj, on_finalize, 'extra argument')

del obj
```

Аргументами функции `finalize()` являются отслеживаемый объект, вызываемый объект, подлежащий вызову при удалении объекта сборщиком мусора, и любые позиционные или именованные аргументы, передаваемые вызываемому объекту.

```
$ python3 weakref_finalize.py

(Deleting <__main__.ExpensiveObject object at 0x1019b10f0>)
on_finalize(('extra argument',))
```

Экземпляр finalize имеет перезаписываемое свойство atexit, позволяющее управлять вызовом функции обратного вызова при выходе из программы, если она до этого не вызывалась.

Листинг 2.67. weakref_finalize_atexit.py

```
import sys
import weakref


class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

def on_finalize(*args):
    print('on_finalize({!r})'.format(args))

obj = ExpensiveObject()
f = weakref.finalize(obj, on_finalize, 'extra argument')
f.atexit = bool(int(sys.argv[1]))
```

Поведению по умолчанию соответствует вызов функции обратного вызова. Это поведение можно изменить, задав для свойства atexit ложное значение.

```
$ python3 weakref_finalize_atexit.py 1

on_finalize(('extra argument',))
(Deleting <__main__.ExpensiveObject object at 0x1007b10f0>)

$ python3 weakref_finalize_atexit.py 0
```

Предоставление экземпляру finalize ссылки на объект приводит к тому, что она удерживается в памяти, поэтому сборщик мусора не удалит данный объект.

Листинг 2.68. weakref_finalize_reference.py

```
import gc
import weakref


class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

def on_finalize(*args):
    print('on_finalize({!r})'.format(args))

obj = ExpensiveObject()
obj_id = id(obj)

f = weakref.finalize(obj, on_finalize, obj)
f.atexit = False
```

```
del obj

for o in gc.get_objects():
    if id(o) == obj_id:
        print('found uncollected object in gc')
```

Как следует из этого примера, объект obj сохраняется в памяти даже после удаления прямой ссылки на него и остается видимым для сборщика мусора посредством объекта-финализатора f.

```
$ python3 weakref_finalize_reference.py

found uncollected object in gc
```

Использование связанного метода отслеживаемого объекта также может препятствовать надлежащему выполнению завершающих операций по освобождению системных ресурсов.

Листинг 2.69. weakref_finalize_reference_method.py

```
import gc
import weakref


class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

    def do_finalize(self):
        print('do_finalize')


obj = ExpensiveObject()
obj_id = id(obj)

f = weakref.finalize(obj, obj.do_finalize)
f.atexit = False

del obj

for o in gc.get_objects():
    if id(o) == obj_id:
        print('found uncollected object in gc')
```

Поскольку вызываемым объектом, передаваемым функции finalize(), является связанный метод экземпляра obj, объект-финализатор удерживает ссылку на obj, который поэтому не может быть удален и обработан сборщиком мусора.

```
$ python3 weakref_finalize_reference_method.py

found uncollected object in gc
```

2.8.4. Прокси-объекты

Иногда вместо слабой ссылки удобнее использовать *прокси-объект* (или просто *прокси*). Прокси-объект ведет себя так, как если бы это был исходный объект, но он не должен вызываться до тех пор, пока не станет доступным объект, который он представляет. Вследствие этого прокси-объект можно передать библиотеке, которой не известно, что она получает ссылку вместо реального объекта.

Листинг 2.70. `weakref_proxy.py`

```
import weakref


class ExpensiveObject:

    def __init__(self, name):
        self.name = name

    def __del__(self):
        print('(Deleting {})'.format(self))

obj = ExpensiveObject('My Object')
r = weakref.ref(obj)
p = weakref.proxy(obj)

print('via obj:', obj.name)
print('via ref:', r().name)
print('via proxy:', p.name)
del obj
print('via proxy:', p.name)
```

В случае обращения к прокси-объекту после удаления объекта, который он представляет, возбуждается исключение `ReferenceError`.

```
$ python3 weakref_proxy.py

via obj: My Object
via ref: My Object
via proxy: My Object
(Deleting <__main__.ExpensiveObject object at 0x1007aa7b8>)
Traceback (most recent call last):
  File "weakref_proxy.py", line 30, in <module>
  print('via proxy:', p.name)
ReferenceError: weakly-referenced object no longer exists
```

2.8.5. Объекты кеширования

Классы `ref` и `proxy` считаются "низкоуровневыми". И хотя они удобны для работы со слабыми ссылками на отдельные объекты и позволяют избежать проблем с автоматическим удалением объектов, связанных циклическими ссылками, сборщиком мусора, классы `WeakKeyDictionary` и `WeakValueDictionary` предоставляют более подходящий API для создания кеша из нескольких объектов.

Класс WeakValueDictionary позволяет создать словарь, в котором значения представлены слабыми ссылками и могут автоматически удаляться сборщиком мусора, если они больше не используются другим кодом. Ниже приведен пример, в котором явные вызовы сборщика мусора используются для демонстрации различий между управлением памятью в случае обычного словаря и словаря WeakValueDictionary.

Листинг 2.71. weakref_valuedict.py

```
import gc
from pprint import pprint
import weakref

gc.set_debug(gc.DEBUG_UNCOLLECTABLE)


class ExpensiveObject:

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return 'ExpensiveObject({})'.format(self.name)

    def __del__(self):
        print('    (Deleting {})'.format(self))


def demo(cache_factory):
    # Удержать объекты для того, чтобы предотвратить
    # немедленное удаление слабых ссылок
    all_refs = {}
    # Создать кеш, используя функцию-фабрику
    print('CACHE TYPE:', cache_factory)
    cache = cache_factory()
    for name in ['one', 'two', 'three']:
        o = ExpensiveObject(name)
        cache[name] = o
        all_refs[name] = o
        del o  # уменьшить счетчик ссылок

    print('  all_refs =', end=' ')
    pprint(all_refs)
    print('\n  Before, cache contains:', list(cache.keys()))
    for name, value in cache.items():
        print('    {} = {}'.format(name, value))
        del value  # уменьшить счетчик ссылок

    # Удалить все ссылки на объекты, за исключением тех,
    # которые находятся в кеше
    print('\n  Cleanup:')
    del all_refs
    gc.collect()
```

```
    print('\n  After, cache contains:', list(cache.keys()))
    for name, value in cache.items():
        print('    {} = {}'.format(name, value))
    print('  demo returning')
    return


demo(dict)
print()

demo(weakref.WeakValueDictionary)
```

Используемые в циклах `for` переменные, ссылающиеся на кешируемые значения, должны удаляться явным образом, что и обусловливает необходимость уменьшения счетчиков ссылок на соответствующие объекты. Если этого не сделать, то сборщик мусора не сможет удалить объекты, и они будут оставаться в кеше. Переменная `all_refs` используется для сохранения ссылок с той целью, чтобы они не были преждевременно удалены сборщиком мусора.

```
$ python3 weakref_valuedict.py

CACHE TYPE: <class 'dict'>
  all_refs = {'one': ExpensiveObject(one),
 'three': ExpensiveObject(three),
 'two': ExpensiveObject(two)}

  Before, cache contains: ['two', 'one', 'three']
    two = ExpensiveObject(two)
    one = ExpensiveObject(one)
    three = ExpensiveObject(three)

  Cleanup:

  After, cache contains: ['two', 'one', 'three']
    two = ExpensiveObject(two)
    one = ExpensiveObject(one)
    three = ExpensiveObject(three)
  demo returning
    (Deleting ExpensiveObject(two))
    (Deleting ExpensiveObject(one))
    (Deleting ExpensiveObject(three))

CACHE TYPE: <class 'weakref.WeakValueDictionary'>
  all_refs = {'one': ExpensiveObject(one),
 'three': ExpensiveObject(three),
 'two': ExpensiveObject(two)}

  Before, cache contains: ['two', 'one', 'three']
    two = ExpensiveObject(two)
    one = ExpensiveObject(one)
    three = ExpensiveObject(three)

  Cleanup:
```

```
    (Deleting ExpensiveObject(two))
    (Deleting ExpensiveObject(one))
    (Deleting ExpensiveObject(three))

  After, cache contains: []
  demo returning
```

Предупреждение

В разделе документации стандартной библиотеки, посвященном модулю `weakref`, содержится следующее предостережение:

> "Поскольку словарь `WeakValueDictionary` создается поверх словаря Python, его размер не должен изменяться при итерировании по элементам. Добиться гарантированного выполнения этого условия для словаря `WeakValueDictionary` непросто, поскольку действия программы при выполнении итераций могут приводить к "загадочному" исчезновению элементов (обусловленному побочными эффектами сборки мусора)".

Дополительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `weakref`[24].
- `gc` (раздел 17.6). Модуль `gc` предоставляет интерфейс для доступа к средствам сборки мусора, предлагаемым интерпретатором.
- **PEP 205**[25]. *Weak References*. Предложения по улучшению слабых ссылок.

2.9. copy: создание дубликатов объектов

Модуль `copy` включает две функции, `copy()` и `deepcopy()`, предназначенные для создания дубликатов существующих объектов.

2.9.1. Мелкие копии

Функция `copy()` создает *мелкую (поверхностную) копию* объекта, которая представляет собой новый контейнер, заполненный ссылками на содержимое исходного объекта. Когда создается мелкая копия объекта `list`, конструируется новый объект `list`, к которому присоединяются элементы исходного объекта.

Листинг 2.72. `copy_shallow.py`

```
import copy
import functools


@functools.total_ordering
class MyClass:

    def __init__(self, name):
        self.name = name
```

[24] https://docs.python.org/3.5/library/weakref.html

[25] www.python.org/dev/peps/pep-0205

```
    def __eq__(self, other):
        return self.name == other.name

    def __gt__(self, other):
        return self.name > other.name


a = MyClass('a')
my_list = [a]
dup = copy.copy(my_list)

print('             my_list:', my_list)
print('                 dup:', dup)
print('      dup is my_list:', (dup is my_list))
print('      dup == my_list:', (dup == my_list))
print('dup[0] is my_list[0]:', (dup[0] is my_list[0]))
print('dup[0] == my_list[0]:', (dup[0] == my_list[0]))
```

В случае мелкого копирования дубликат экземпляра `MyClass` не создается, поэтому ссылка, хранящаяся в переменной `dup`, указывает на тот же объект, что и ссылка, хранящаяся в переменной `my_list`.

```
$ python3 copy_shallow.py

            my_list: [<__main__.MyClass object at 0x1007a87b8>]
                dup: [<__main__.MyClass object at 0x1007a87b8>]
     dup is my_list: False
     dup == my_list: True
dup[0] is my_list[0]: True
dup[0] == my_list[0]: True
```

2.9.2. Глубокие копии

Функция `deepcopy()` создает *глубокую копию* объекта, которая представляет собой новый контейнер, заполненный копиями содержимого исходного объекта. Когда создается глубокая копия объекта `list`, конструируется новый объект `list`, а затем копируются элементы исходного списка, после чего эти копии присоединяются к новому списку.

Замена вызова `copy()` вызовом `deepcopy()` приводит к результату, существенно отличающемуся от предыдущего.

Листинг 2.73. copy_deep.py

```
import copy
import functools


@functools.total_ordering
class MyClass:

    def __init__(self, name):
        self.name = name
```

```
    def __eq__(self, other):
        return self.name == other.name

    def __gt__(self, other):
        return self.name > other.name


a = MyClass('a')
my_list = [a]
dup = copy.deepcopy(my_list)

print('             my_list:', my_list)
print('                 dup:', dup)
print('      dup is my_list:', (dup is my_list))
print('      dup == my_list:', (dup == my_list))
print('dup[0] is my_list[0]:', (dup[0] is my_list[0]))
print('dup[0] == my_list[0]:', (dup[0] == my_list[0]))
```

Первый элемент списка уже не является ссылкой на тот же объект, но сравнение обоих объектов между собой показывает, что они равны.

```
$ python3 copy_deep.py

             my_list: [<__main__.MyClass object at 0x1018a87b8>]
                 dup: [<__main__.MyClass object at 0x1018b1b70>]
      dup is my_list: False
      dup == my_list: True
dup[0] is my_list[0]: False
dup[0] == my_list[0]: True
```

2.9.3. Настройка копирования

Процессами копирования можно управлять с помощью специальных методов `__copy__()` и `__deepcopy__()`.

- Метод `__copy__()` вызывается без аргументов и должен возвращать мелкую копию объекта.
- Метод `__deepcopy__()` вызывается с передачей ему вспомогательного словаря и должен возвращать глубокую копию объекта. Для управления рекурсией каждый атрибут, нуждающийся в глубоком копировании, должен передаваться функции `copy.deepcopy()` вместе со вспомогательным словарем. (Далее вспомогательный словарь обсуждается более подробно.)

В приведенном ниже примере показано, как должны вызываться эти методы.

Листинг 2.74. copy_hooks.py

```
import copy
import functools


@functools.total_ordering
class MyClass:
```

```
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

    def __gt__(self, other):
        return self.name > other.name

    def __copy__(self):
        print('__copy__()')
        return MyClass(self.name)

    def __deepcopy__(self, memo):
        print('__deepcopy__({})'.format(memo))
        return MyClass(copy.deepcopy(self.name, memo))


a = MyClass('a')

sc = copy.copy(a)
dc = copy.deepcopy(a)
```

Вспомогательный словарь (`memo`) используется для отслеживания уже скопированных значений, чтобы избежать бесконечной рекурсии.

```
$ python3 copy_hooks.py

__copy__()
__deepcopy__({})
```

2.9.4. Рекурсия при глубоком копировании

Чтобы избежать проблем с копированием рекурсивных структур данных, метод `deepcopy()` использует вспомогательный словарь, с помощью которого отслеживает уже скопированные данные. Этот словарь передается методу `__deepcopy__()`, чтобы тот также мог просматривать его содержимое.

В следующем примере показано, каким образом связанная структура данных, такая как направленный граф, может помочь в защите от рекурсии путем реализации метода `__deepcopy__()`.

Листинг 2.75. `copy_recursion.py`

```
import copy


class Graph:

    def __init__(self, name, connections):
        self.name = name
        self.connections = connections

    def add_connection(self, other):
        self.connections.append(other)
```

```
    def __repr__(self):
        return 'Graph(name={}, id={})'.format(
            self.name, id(self))

    def __deepcopy__(self, memo):
        print('\nCalling __deepcopy__ for {!r}'.format(self))
        if self in memo:
            existing = memo.get(self)
            print('  Already copied to {!r}'.format(existing))
            return existing
        print('  Memo dictionary:')
        if memo:
            for k, v in memo.items():
                print('    {}: {}'.format(k, v))
        else:
            print('    (empty)')
        dup = Graph(copy.deepcopy(self.name, memo), [])
        print('  Copying to new object {}'.format(dup))
        memo[self] = dup
        for c in self.connections:
            dup.add_connection(copy.deepcopy(c, memo))
        return dup


root = Graph('root', [])
a = Graph('a', [root])
b = Graph('b', [a, root])
root.add_connection(a)
root.add_connection(b)

dup = copy.deepcopy(root)
```

Класс `Graph` включает ряд базовых методов ориентированного графа. Экземпляр может быть инициализирован именем и списком существующих узлов, с которым связано данное имя. Метод `add_connection()` используется для установления двунаправленных связей. Он также используется оператором глубокого копирования.

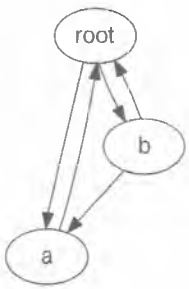

Рис. 2.1. Глубокое копирование объекта циклического графа

Метод `__deepcopy__()` выводит сообщения, информирующие о его вызовах, и осуществляет необходимое управление содержимым вспомогательного слова-

ря. Вместо копирования всего списка он создает новый список и присоединяет к нему копии индивидуальных связей. Это гарантирует обновление вспомогательного словаря всякий раз, когда создается дубликат нового узла, и позволяет избежать проблем с рекурсией или лишними копиями узлов. Как и перед этим, данный метод возвращает копию объекта.

Представленный на рис. 2.1 граф включает несколько циклов, но обработка рекурсии с помощью вспомогательного словаря предотвращает переполнение стека при обходе узлов. Ниже представлен вывод, получаемый при копировании корневого узла.

```
$ python3 copy_recursion.py

Calling __deepcopy__ for Graph(name=root, id=4314569528)
  Memo dictionary:
    (empty)
  Copying to new object Graph(name=root, id=4315093592)

Calling __deepcopy__ for Graph(name=a, id=4314569584)
  Memo dictionary:
    Graph(name=root, id=4314569528): Graph(name=root,
id=4315093592)
  Copying to new object Graph(name=a, id=4315094208)

Calling __deepcopy__ for Graph(name=root, id=4314569528)
  Already copied to Graph(name=root, id=4315093592)

Calling __deepcopy__ for Graph(name=b, id=4315092248)
  Memo dictionary:
    4314569528: Graph(name=root, id=4315093592)
    4315692808: [Graph(name=root, id=4314569528), Graph(name=a,
id=4314569584)]
    Graph(name=root, id=4314569528): Graph(name=root,
id=4315093592)
    4314569584: Graph(name=a, id=4315094208)
    Graph(name=a, id=4314569584): Graph(name=a, id=4315094208)
  Copying to new object Graph(name=b, id=4315177536)
```

Если при копировании какого-либо узла корневой узел встречается второй раз, то метод `__deepcopy__()` обнаруживает рекурсию и повторно использует существующее значение из вспомогательного словаря, а не создает новый объект.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `copy`[26].

2.10. `pprint`: "красивая печать" структур данных

Модуль `pprint` содержит средства "красивой печати", обеспечивающие привлекательный внешний вид выводимого представления структур данных. Форма-

[26] https://docs.python.org/3.5/library/copy.html

тировщик создает представления структур данных, пригодные для анализа интерпретатором и удобные для чтения человеком. По возможности вывод ограничивается одной строкой и выделяется отступами, если распространяется на несколько строк.

Во всех приведенных ниже примерах используются данные из модуля pprint_data.py.

Листинг 2.76. pprint_data.py

```
data = [
    (1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
    (2, {'e': 'E', 'f': 'F', 'g': 'G', 'h': 'H',
         'i': 'I', 'j': 'J', 'k': 'K', 'l': 'L'}),
    (3, ['m', 'n']),
    (4, ['o', 'p', 'q']),
    (5, ['r', 's', 't''u', 'v', 'x', 'y', 'z']),
]
```

2.10.1. Вывод на консоль

Доступ к простейшим возможностям модуля pprint обеспечивает непосредственное использование функции pprint().

Листинг 2.77. pprint_pprint.py

```
from pprint import pprint

from pprint_data import data

print('PRINT:')
print(data)
print()
print('PPRINT:')
pprint(data)
```

Функция pprint() форматирует объект и записывает его в поток данных, переданный ей в качестве аргумента (по умолчанию – sys.stdout).

```
$ python3 pprint_pprint.py

PRINT:
[(1, {'c': 'C', 'd': 'D', 'a': 'A', 'b': 'B'}), (2, {'l': 'L',
'g': 'G', 'f': 'F', 'h': 'H', 'i': 'I', 'k': 'K', 'j': 'J', 'e':
'E'}), (3, ['m', 'n']), (4, ['o', 'p', 'q']), (5, ['r', 's', 'tu',
'v', 'x', 'y', 'z'])]

PPRINT:
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
```

```
   'j': 'J',
   'k': 'K',
   'l': 'L'}),
 (3, ['m', 'n']),
 (4, ['o', 'p', 'q']),
 (5, ['r', 's', 'tu', 'v', 'x', 'y', 'z'])]
```

2.10.2. Форматирование

Для форматирования структуры данных с целью создания ее строкового представления без непосредственной записи в поток (например, для протоколирования работы программы) используйте функцию `pformat()`.

Листинг 2.78. `pprint_pformat.py`

```
import logging
from pprint import pformat
from pprint_data import data

logging.basicConfig(
    level=logging.DEBUG,
    format='%(levelname)-8s %(message)s',
)

logging.debug('Logging pformatted data')
formatted = pformat(data)
for line in formatted.splitlines():
    logging.debug(line.rstrip())
```

Затем форматированную строку можно вывести на консоль или записать в журнал.

```
$ python3 pprint_pprint.py

DEBUG    Logging pformatted data
DEBUG    [(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
DEBUG     (2,
DEBUG      {'e': 'E',
DEBUG       'f': 'F',
DEBUG       'g': 'G',
DEBUG       'h': 'H',
DEBUG       'i': 'I',
DEBUG       'j': 'J',
DEBUG       'k': 'K',
DEBUG       'l': 'L'}),
DEBUG     (3, ['m', 'n']),
DEBUG     (4, ['o', 'p', 'q']),
DEBUG     (5, ['r', 's', 'tu', 'v', 'x', 'y', 'z'])]
```

2.10.3. Произвольные классы

Класс `PrettyPrinter`, используемый функцией `pprint()`, может также работать с пользовательскими классами, если они определяют метод `__repr__()`.

Листинг 2.79. pprint_arbitrary_object.py

```
from pprint import pprint


class node:

    def __init__(self, name, contents=[]):
        self.name = name
        self.contents = contents[:]

    def __repr__(self):
        return (
            'node(' + repr(self.name) + ', ' +
            repr(self.contents) + ')'
        )


trees = [
    node('node-1'),
    node('node-2', [node('node-2-1')]),
    node('node-3', [node('node-3-1')]),
]
pprint(trees)
```

Представления вложенных объектов комбинируются классом `PrettyPrinter` для возврата полного строкового представления составного объекта.

```
$ python3 pprint_arbitrary_object.py

[node('node-1', []),
 node('node-2', [node('node-2-1', [])]),
 node('node-3', [node('node-3-1', [])])]
```

2.10.4. Рекурсия

Для представления рекурсивных структур данных используется ссылка на первоначальный источник данных в формате `<Recursion on` *`имя_типа`* `with id=`*`число`*`>`.

Листинг 2.80. pprint_recursion.py

```
from pprint import pprint

local_data = ['a', 'b', 1, 2]
local_data.append(local_data)

print('id(local_data) =>', id(local_data))
pprint(local_data)
```

В этом примере список `local_data` добавляется к себе самому, создавая рекурсивную ссылку.

```
$ python3 pprint_recursion.py

id(local_data) => 4324368136
['a', 'b', 1, 2, <Recursion on list with id=4324368136>]
```

2.10.5. Ограничение уровня выводимых вложенных структур

Для структур с большой глубиной вложения подробный вывод всех вложенных структур данных не всегда желателен. В подобных случаях данные могут не форматироваться надлежащим образом, объем форматированного текста может оказаться очень большим, а некоторые данные могут оказаться инородными.

Листинг 2.81. pprint_depth.py

```
from pprint import pprint

from pprint_data import data

pprint(data, depth=1)
pprint(data, depth=2)
```

Глубиной рекурсии при выводе таких вложенных структур можно управлять с помощью аргумента `depth`. Уровни, не включенные в вывод, представляются многоточиями.

```
$ python3 pprint_depth.py

[(...), (...), (...), (...), (...)]
[(1, {...}), (2, {...}), (3, [...]), (4, [...]), (5, [...])]
```

2.10.6. Управление шириной вывода

Ширина вывода форматированного текста, используемая по умолчанию, составляет 80 столбцов. Этим параметром можно управлять, передавая функции `pprint()` аргумент `width`.

Листинг 2.82. pprint_width.py

```
from pprint import pprint

from pprint_data import data

for width in [80, 5]:
    print('WIDTH =', width)
    pprint(data, width=width)
    print()
```

При слишком малых значениях `width`, недостаточных для размещения форматированной структуры данных в заданных пределах, строки не усекаются и не переходят на следующую строку, если это приведет к нарушению синтаксиса.

```
$ python3 pprint_width.py

WIDTH = 80
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'}),
 (3, ['m', 'n']),
 (4, ['o', 'p', 'q']),
 (5, ['r', 's', 'tu', 'v', 'x', 'y', 'z'])]

WIDTH = 5
[(1,
  {'a': 'A',
   'b': 'B',
   'c': 'C',
   'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'}),
 (3,
  ['m',
   'n']),
 (4,
  ['o',
   'p',
   'q']),
 (5,
  ['r',
   's',
   'tu',
   'v',
   'x',
   'y',
   'z'])]
```

Флаг `compact` позволяет сообщить функции `pprint()`, чтобы она попыталась уместить в каждой строке как можно больше данных, а не распределяла сложные структуры данных по нескольким строкам.

Листинг 2.83. pprint_compact.py

```
from pprint import pprint

from pprint_data import data

print('DEFAULT:')
pprint(data, compact=False)
print('\nCOMPACT:')
pprint(data, compact=True)
```

Этот пример показывает, что в тех случаях, когда структура данных не умещается в одной строке, она распределяется по нескольким строкам (как в случае второго элемента в списке данных). Если же в одной строке могут уместиться несколько элементов (как с случае третьего и четвертого элементов), то именно так они и выводятся.

```
$ python3 pprint_compact.py

DEFAULT:
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'}),
 (3, ['m', 'n']),
 (4, ['o', 'p', 'q']),
 (5, ['r', 's', 'tu', 'v', 'x', 'y', 'z'])]

COMPACT:
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2,
  {'e': 'E',
   'f': 'F',
   'g': 'G',
   'h': 'H',
   'i': 'I',
   'j': 'J',
   'k': 'K',
   'l': 'L'}),
 (3, ['m', 'n']), (4, ['o', 'p', 'q']),
 (5, ['r', 's', 'tu', 'v', 'x', 'y', 'z'])]
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `pprint`[27].

[27] https://docs.python.org/3.5/library/pprint.html

Глава 3

Алгоритмы

Python включает несколько модулей, обеспечивающих элегантную и лаконичную реализацию алгоритмов с использованием стиля, наиболее подходящего для конкретной задачи. Эти модули поддерживают чисто процедурный, объектно-ориентированный и функциональный стили программирования, и эти стили часто смешиваются в различных частях одной и той же программы.

Модуль `functools` (раздел 3.1) включает функции и декораторы, которые предоставляют возможности аспектно-ориентированного программирования и повторного использования кода, позволяющие расширить рамки традиционного объектно-ориентированного подхода. Он также предоставляет декоратор класса для реализации богатых возможностей API сравнения с помощью сокращенных форм вызова функций и объекты типа `partial`, позволяющие создавать ссылки на функции с уже включенными аргументами.

Модуль `itertools` (раздел 3.2) включает функции для создания итераторов и генераторов, используемых в функциональном программировании. В случае применения функционального стиля программирования модуль `operator` (раздел 3.3) позволяет устранить необходимость в использовании многих тривиальных лямбда-функций за счет предоставления интерфейсов на основе функций к таким встроенным операциям, как арифметические операции или операции поиска элементов.

Какой бы стиль программирования ни использовался в программе, модуль `contextlib` (раздел 3.4) упрощает управление ресурсами, повышая надежность и лаконичность этого процесса. Сочетание менеджеров контекста с инструкциями `with` позволяет уменьшить количество блоков `try:finally` и уровней отступов в коде, одновременно обеспечивая закрытие и своевременное освобождение таких ресурсов, как файлы, сокеты, транзакции баз данных и многое другое.

3.1. `functools`: инструменты для манипулирования функциями

Предоставляемые модулем `functools` инструменты позволяют адаптировать или расширять функции и другие вызываемые объекты без полного их переписывания.

3.1.1. Декораторы

Основной инструмент, предоставляемый модулем `functools`, — класс `partial`, который можно использовать в качестве "обертки" вокруг вызываемого объекта, задавая значения по умолчанию для части его аргументов. Результирующий объект сам является вызываемым, и с ним можно обращаться так, как если бы это была исходная функция. Он получает те же аргументы, но может вызываться с до-

полнительными позиционными или именованными аргументами. Объект типа `partial` можно использовать вместо `lambda` для передачи функции аргументов, заданных по умолчанию, в то же время оставляя не указанными некоторые аргументы.

3.1.1.1. Объекты `partial`

В первом примере представлены два простых объекта `partial` для функции `myfunc()`. Функция `show_details()` выводит информацию об атрибутах `func`, `args` и `keywords` объекта `partial`.

Листинг 3.1. `functools_partial.py`

```
import functools


def myfunc(a, b=2):
    "Docstring for myfunc()."
    print('  called myfunc with:', (a, b))


def show_details(name, f, is_partial=False):
    "Показать детали вызываемого объекта."
    print('{}:'.format(name))
    print('  object:', f)
    if not is_partial:
        print('  __name__:', f.__name__)
    if is_partial:
        print('  func:', f.func)
        print('  args:', f.args)
        print('  keywords:', f.keywords)
    return


show_details('myfunc', myfunc)
myfunc('a', 3)
print()

# Задать другое значение по умолчанию для 'b', но потребовать,
# чтобы вызывающий код предоставил 'a'
p1 = functools.partial(myfunc, b=4)
show_details('partial with named default', p1, True)
p1('passing a')
p1('override b', b=5)
print()

# Задать значения по умолчанию для 'a' и 'b'
p2 = functools.partial(myfunc, 'default a', b=99)
show_details('partial with defaults', p2, True)
p2()
p2(b='override b')
print()
```

```
print('Insufficient arguments:')
p1()
```

В конце примера первый из объектов partial вызывается без передачи значения для a, что приводит к возбуждению исключения.

```
$ python3 functools_partial.py

myfunc:
  object: <function myfunc at 0x1007a6a60>
  __name__: myfunc
  called myfunc with: ('a', 3)

partial with named default:
  object: functools.partial(<function myfunc at 0x1007a6a60>, b=4)
  func: <function myfunc at 0x1007a6a60>
  args: ()
  keywords: {'b': 4}
  called myfunc with: ('passing a', 4)
  called myfunc with: ('override b', 5)

partial with defaults:
  object: functools.partial(<function myfunc at 0x1007a6a60>,
'default a', b=99)
  func: <function myfunc at 0x1007a6a60>
  args: ('default a',)
  keywords: {'b': 99}
  called myfunc with: ('default a', 99)
  called myfunc with: ('default a', 'override b')

Insufficient arguments:
Traceback (most recent call last):
  File "S:\Projects\py\test.py", line 42, in <module>
    p1()
TypeError: myfunc() missing 1 required positional argument: 'a'
```

3.1.1.2. Копирование и добавление свойств функции

По умолчанию объект partial не имеет атрибутов __name__ и __doc__, и их отсутствие затрудняет отладку декорированных функций. Функция update_wrapper() позволяет добавлять и копировать атрибуты из исходной функции в объект partial.

Листинг 3.2. functools_update_wrapper.py

```
import functools


def myfunc(a, b=2):
    "Docstring for myfunc()."
    print('  called myfunc with:', (a, b))
```

```
def show_details(name, f):
    "Показать детали вызываемого объекта."
    print('{}:'.format(name))
    print('  object:', f)
    print('  __name__:', end=' ')
    try:
        print(f.__name__)
    except AttributeError:
        print('(no __name__)')
    print('  __doc__', repr(f.__doc__))
    print()


show_details('myfunc', myfunc)

p1 = functools.partial(myfunc, b=4)
show_details('raw wrapper', p1)

print('Updating wrapper:')
print('  assign:', functools.WRAPPER_ASSIGNMENTS)
print('  update:', functools.WRAPPER_UPDATES)
print()

functools.update_wrapper(p1, myfunc)
show_details('updated wrapper', p1)
```

Значения по умолчанию атрибутов, добавляемых в функцию-обертку, определяются с помощью константы `WRAPPER_ASSIGNMENTS` уровня модуля, тогда как значения по умолчанию, обновляющие соответствующие атрибуты функции-обертки, — с помощью константы `WRAPPER_UPDATES`.

```
$ python3 functools_update_wrapper.py

myfunc:
  object: <function myfunc at 0x1018a6a60>
  __name__: myfunc
  __doc__ 'Docstring for myfunc().'

raw wrapper:
  object: functools.partial(<function myfunc at 0x1018a6a60>, b=4)
  __name__: (no __name__)
  __doc__ 'partial(func, *args, **keywords) - new function with
partial application\n    of the given arguments and keywords.\n'

Updating wrapper:
  assign: ('__module__', '__name__', '__qualname__', '__doc__',
'__annotations__')
  update: ('__dict__',)

updated wrapper:
  object: functools.partial(<function myfunc at 0x1018a6a60>, b=4)
  __name__: myfunc
  __doc__ 'Docstring for myfunc().'
```

3.1.1.3. Другие вызываемые объекты

Класс `partial` позволяет работать с любыми вызываемыми объектами, а не только с автономными функциями.

Листинг 3.3. `functools_callable.py`

```
import functools


class MyClass:
    "Демонстрационный класс для functools"

    def __call__(self, e, f=6):
        "Docstring for MyClass.__call__"
        print('  called object with:', (self, e, f))


def show_details(name, f):
    "Показать детали вызываемого объекта."
    print('{}:'.format(name))
    print('  object:', f)
    print('  __name__:', end=' ')
    try:
        print(f.__name__)
    except AttributeError:
        print('(no __name__)')
    print('  __doc__', repr(f.__doc__))
    return


o = MyClass()

show_details('instance', o)
o('e goes here')
print()

p = functools.partial(o, e='default for e', f=8)
functools.update_wrapper(p, o)
show_details('instance wrapper', p)
p()
```

В следующем примере объекты `partial` создаются из экземпляра класса с помощью метода `__call__()`.

```
$ python3 functools_callable.py

instance:
  object: <__main__.MyClass object at 0x1011b1cf8>
  __name__: (no __name__)
  __doc__ 'Demonstration class for functools'
  called object with: (<__main__.MyClass object at 0x1011b1cf8>,
'e goes here', 6)
```

```
instance wrapper:
  object: functools.partial(<__main__.MyClass object at
0x1011b1cf8>, e='default for e', f=8)
  __name__: (no __name__)
  __doc__ 'Demonstration class for functools'
  called object with: (<__main__.MyClass object at 0x1011b1cf8>,
'default for e', 8)
```

3.1.1.4. Методы и функции

В то время как функция `partial()` возвращает вызываемый объект, готовый к непосредственному использованию, функция `partialmethod()` возвращает объект, готовый к использованию в качестве несвязанного метода объекта. В следующем примере одна и та же автономная функция добавляется в качестве атрибута класса `MyClass` дважды: один раз с помощью функции `partialmethod()`, как `method1()`, и второй раз с помощью функции `partial()`, как `method2()`.

Листинг 3.4. `functools_partialmethod.py`

```
import functools


def standalone(self, a=1, b=2):
    "Автономная функция"
    print('  called standalone with:', (self, a, b))
    if self is not None:
        print('  self.attr =', self.attr)


class MyClass:
    "Демонстрационный класс для functools"

    def __init__(self):
        self.attr = 'instance attribute'

    method1 = functools.partialmethod(standalone)
    method2 = functools.partial(standalone)


o = MyClass()

print('standalone')
standalone(None)
print()

print('method1 as partialmethod')
o.method1()
print()

print('method2 as partial')
try:
    o.method2()
except TypeError as err:
    print('ERROR: {}'.format(err))
```

Метод method1() может вызываться для экземпляра MyClass, и экземпляр передается ему в качестве первого аргумента точно так же, как в случае методов, определенных обычным способом. Метод method2() не задан как связанный, поэтому аргумент self должен передаваться ему явным образом, иначе вызов приведет к возбуждению исключения TypeError.

```
$ python3 functools_partialmethod.py

standalone
  called standalone with: (None, 1, 2)

method1 as partialmethod
  called standalone with: (<__main__.MyClass object at
0x1007b1d30>, 1, 2)
  self.attr = instance attribute

method2 as partial
ERROR: standalone() missing 1 required positional argument:
'self'
```

3.1.1.5. Наделение декоратора свойствами декорируемой функции

Обновление свойств функции-обертки особенно полезно в случае декораторов, поскольку это позволяет сделать ее похожей на исходную "голую" функцию.

Листинг 3.5. functools_wraps.py

```
import functools


def show_details(name, f):
    "Показать детали вызываемого объекта."
    print('{}:'.format(name))
    print('  object:', f)
    print('  __name__:', end=' ')
    try:
        print(f.__name__)
    except AttributeError:
        print('(no __name__)')
    print('  __doc__', repr(f.__doc__))
    print()


def simple_decorator(f):
    @functools.wraps(f)
    def decorated(a='decorated defaults', b=1):
        print('  decorated:', (a, b))
        print('  ', end=' ')
        return f(a, b=b)
    return decorated


def myfunc(a, b=2):
```

```
    "myfunc() is not complicated"
    print('  myfunc:', (a, b))
    return


# Исходная функция
show_details('myfunc', myfunc)
myfunc('unwrapped, default b')
myfunc('unwrapped, passing b', 3)
print()

# Явное упаковывание
wrapped_myfunc = simple_decorator(myfunc)
show_details('wrapped_myfunc', wrapped_myfunc)
wrapped_myfunc()
wrapped_myfunc('args to wrapped', 4)
print()


# Упаковывание с помощью синтаксиса декоратора
@simple_decorator
def decorated_myfunc(a, b):
    myfunc(a, b)
    return


show_details('decorated_myfunc', decorated_myfunc)
decorated_myfunc()
decorated_myfunc('args to decorated', 4)
```

Модуль `functools` предоставляет декоратор `wraps()`, который применяет функцию `update_wrapper()` к декорируемой функции.

```
$ python3 functools_wraps.py

myfunc:
  object: <function myfunc at 0x101241b70>
  __name__: myfunc
  __doc__ 'myfunc() is not complicated'

  myfunc: ('unwrapped, default b', 2)
  myfunc: ('unwrapped, passing b', 3)

wrapped_myfunc:
  object: <function myfunc at 0x1012e62f0>
  __name__: myfunc
  __doc__ 'myfunc() is not complicated'

  decorated: ('decorated defaults', 1)
     myfunc: ('decorated defaults', 1)
  decorated: ('args to wrapped', 4)
     myfunc: ('args to wrapped', 4)
```

```
decorated_myfunc:
  object: <function decorated_myfunc at 0x1012e6400>
  __name__: decorated_myfunc
  __doc__ None

  decorated: ('decorated defaults', 1)
     myfunc: ('decorated defaults', 1)
  decorated: ('args to decorated', 4)
     myfunc: ('args to decorated', 4)
```

3.1.2. Сравнение

В Python 2 классы могут определять метод `__cmp__()`, который возвращает значение `-1`, `0` или `1`, в зависимости от того, является ли объект соответственно меньшим, равным или большим, чем элемент, с которым он сравнивается. В Python 2.1 появился API методов расширенного сравнения (`__lt__()`, `__le__()`, `__eq__()`, `__ne__()`, `__gt__()` и `__ge__()`), которые выполняют сравнение и возвращают булево значение. В Python 3 от метода `__cmp__()` отказались как от устаревшего в пользу этих новых методов, и модуль `functools` предоставляет инструменты, которые упрощают написание классов, совместимых с требованиями к операциям сравнения в Python 3.

3.1.2.1. Расширенное сравнение

API расширенного сравнения проектировался с таким расчетом, чтобы обеспечить максимально эффективную реализацию каждого теста для классов со сложными операциями сравнения. Однако в случае классов с относительно простыми операциями сравнения не имеет смысла создавать вручную каждый метод. Декоратор классов `total_ordering()` получает класс, который предоставляет некоторые из методов сравнения, и добавляет остальные методы.

Листинг 3.6. `functools_total_ordering.py`

```
import functools
import inspect
from pprint import pprint


@functools.total_ordering
class MyObject:

    def __init__(self, val):
        self.val = val

    def __eq__(self, other):
        print('  testing __eq__({}, {})'.format(
            self.val, other.val))
        return self.val == other.val

    def __gt__(self, other):
        print('  testing __gt__({}, {})'.format(
            self.val, other.val))
```

```
        return self.val > other.val


print('Methods:\n')
pprint(inspect.getmembers(MyObject, inspect.isfunction))

a = MyObject(1)
b = MyObject(2)

print('\nComparisons:')
for expr in ['a < b', 'a <= b', 'a == b', 'a >= b', 'a > b']:
    print('\n{:<6}:'.format(expr))
    result = eval(expr)
    print('  result of {}: {}'.format(expr, result))
```

Класс должен предоставить реализацию метода `__eq__()` и одного из оставшихся методов расширенного сравнения. Декоратор добавляет реализации остальных методов, которые работают, используя предоставленные реализации. Если сравнение не может быть выполнено, метод должен вернуть значение `NotImplemented`, чтобы можно было попытаться использовать операторы обратного сравнения с другим объектом, прежде чем признать операцию неудачной.

```
$ python3 functools_total_ordering.py

Methods:

[('__eq__', <function MyObject.__eq__ at 0x10139a488>),
 ('__ge__', <function _ge_from_gt at 0x1012e2510>),
 ('__gt__', <function MyObject.__gt__ at 0x10139a510>),
 ('__init__', <function MyObject.__init__ at 0x10139a400>),
 ('__le__', <function _le_from_gt at 0x1012e2598>),
 ('__lt__', <function _lt_from_gt at 0x1012e2488>)]

Comparisons:

a < b :
  testing __gt__(1, 2)
  testing __eq__(1, 2)
  result of a < b: True

a <= b:
  testing __gt__(1, 2)
  result of a <= b: True

a == b:
  testing __eq__(1, 2)
  result of a == b: False

a >= b:
  testing __gt__(1, 2)
  testing __eq__(1, 2)
  result of a >= b: False
```

```
a > b :
  testing __gt__(1, 2)
  result of a > b: False
```

3.1.2.2. Порядок сортировки

Поскольку функции сравнения старого стиля признаны устаревшими в Python 3, аргумент `cmp` в функциях наподобие `sort()` больше не поддерживается. Функция `cmp_to_key()` обеспечивает преобразование старых функций в функции, которые возвращают *ключ сортировки*, определяющий позицию объекта в конечной последовательности.

Листинг 3.7. `functools_cmp_to_key.py`

```
import functools


class MyObject:

    def __init__(self, val):
        self.val = val

    def __str__(self):
        return 'MyObject({})'.format(self.val)


def compare_obj(a, b):
    """Функция сравнения старого стиля.
    """
    print('comparing {} and {}'.format(a, b))
    if a.val < b.val:
        return -1
    elif a.val > b.val:
        return 1
    return 0


# Заставить функцию ключа использовать функцию cmp_to_key()
get_key = functools.cmp_to_key(compare_obj)

def get_key_wrapper(o):
    "Функция-обертка для get_key, разрешающая инструкции вывода."
    new_key = get_key(o)
    print('key_wrapper({}) -> {!r}'.format(o, new_key))
    return new_key


objs = [MyObject(x) for x in range(5, 0, -1)]

for o in sorted(objs, key=get_key_wrapper):
    print(o)
```

Обычно функцию `cmp_to_key()` используют непосредственно, но в данном примере привлекается функция-обертка, обеспечивающая вывод дополнительной информации при вызове функции сопоставления.

Как следует из приведенных ниже результатов, выполнение функции `sorted()` начинается с вызова функции `get_key_wrapper()` для каждого элемента последовательности с целью генерации ключей. Ключами, возвращаемыми функцией `cmp_to_key()`, являются экземпляры определенного в модуле `functools` класса, который реализует API расширенного сравнения с использованием передаваемой ему функции сравнения старого стиля. Созданные ключи используются для сортировки последовательности путем их сравнения.

```
$ python3 functools_cmp_to_key.py

key_wrapper(MyObject(5)) -> <functools.KeyWrapper object at
0x1011c5530>
key_wrapper(MyObject(4)) -> <functools.KeyWrapper object at
0x1011c5510>
key_wrapper(MyObject(3)) -> <functools.KeyWrapper object at
0x1011c54f0>
key_wrapper(MyObject(2)) -> <functools.KeyWrapper object at
0x1011c5390>
key_wrapper(MyObject(1)) -> <functools.KeyWrapper object at
0x1011c5710>
comparing MyObject(4) and MyObject(5)
comparing MyObject(3) and MyObject(4)
comparing MyObject(2) and MyObject(3)
comparing MyObject(1) and MyObject(2)
MyObject(1)
MyObject(2)
MyObject(3)
MyObject(4)
MyObject(5)
```

3.1.3. Кеширование

Декоратор `lru_cache()` обертывает функцию кешем LRU (от англ. *least recently used* — "вытесняется элемент, неиспользованный дольше всех"). Аргументы функции используются для создания хеш-значения, которое сопоставляется с результатом. Последующие вызовы функции с теми же аргументами будут заменяться извлечением соответствующего значения из кеша. Кроме того, этот декоратор добавляет в функцию методы, обеспечивающие проверку состояния (`cache_info()`) и очистку (`cache_clear()`) кеша.

Листинг 3.8. `functools_lru_cache.py`

```
import functools


@functools.lru_cache()
def expensive(a, b):
    print('expensive({}, {})'.format(a, b))
```

```
    return a * b


MAX = 2

print('First set of calls:')
for i in range(MAX):
    for j in range(MAX):
        expensive(i, j)
print(expensive.cache_info())

print('\nSecond set of calls:')
for i in range(MAX + 1):
    for j in range(MAX + 1):
        expensive(i, j)
print(expensive.cache_info())

print('\nClearing cache:')
expensive.cache_clear()
print(expensive.cache_info())

print('\nThird set of calls:')
for i in range(MAX):
    for j in range(MAX):
        expensive(i, j)
print(expensive.cache_info())
```

В этом примере выполняется серия вызовов функции `expensive()` в наборе вложенных циклов. При повторных вызовах функции с одними и теми же значениями аргументов результаты появляются в кеше. При повторном выполнении циклов после очистки кеша значения должны вычисляться заново.

```
$ python3 functools_lru_cache.py

First set of calls:
expensive(0, 0)
expensive(0, 1)
expensive(1, 0)
expensive(1, 1)
CacheInfo(hits=0, misses=4, maxsize=128, currsize=4)

Second set of calls:
expensive(0, 2)
expensive(1, 2)
expensive(2, 0)
expensive(2, 1)
expensive(2, 2)
CacheInfo(hits=4, misses=9, maxsize=128, currsize=9)

Clearing cache:
CacheInfo(hits=0, misses=0, maxsize=128, currsize=0)

Third set of calls:
```

```
expensive(0, 0)
expensive(0, 1)
expensive(1, 0)
expensive(1, 1)
CacheInfo(hits=0, misses=4, maxsize=128, currsize=4)
```

Неограниченный рост размера кеша в длительно выполняющихся процессах можно предотвратить, задав его максимальный размер. Размер по умолчанию — 128 записей, но его можно регулировать для каждого кеша с помощью аргумента `maxsize`.

Листинг 3.9. `functools_lru_cache_expire.py`

```
import functools


@functools.lru_cache(maxsize=2)
def expensive(a, b):
    print('called expensive({}, {})'.format(a, b))
    return a * b


def make_call(a, b):
    print('({}, {})'.format(a, b), end=' ')
    pre_hits = expensive.cache_info().hits
    expensive(a, b)
    post_hits = expensive.cache_info().hits
    if post_hits > pre_hits:
        print('cache hit')


print('Establish the cache')
make_call(1, 2)
make_call(2, 3)

print('\nUse cached items')
make_call(1, 2)
make_call(2, 3)

print('\nCompute a new value, triggering cache expiration')
make_call(3, 4)

print('\nCache still contains one old item')
make_call(2, 3)

print('\nOldest item needs to be recomputed')
make_call(1, 2)
```

В этом примере размер кеша ограничен двумя записями. При использовании третьего набора уникальных аргументов (3,4) самая давняя из записей заменяется новым результатом.

```
$ python3 functools_lru_cache_expire.py

Establish the cache
(1, 2) called expensive(1, 2)
(2, 3) called expensive(2, 3)

Use cached items
(1, 2) cache hit
(2, 3) cache hit

Compute a new value, triggering cache expiration
(3, 4) called expensive(3, 4)

Cache still contains one old item
(2, 3) cache hit

Oldest item needs to be recomputed
(1, 2) called expensive(1, 2)
```

Ключи для кеша, управляемого функцией `lru_cache()`, должны быть хешируемыми, и поэтому такими же должны быть аргументы, передаваемые функции, которая обернута кешем.

Листинг 3.10. `functools_lru_cache_arguments.py`

```
import functools


@functools.lru_cache(maxsize=2)
def expensive(a, b):
    print('called expensive({}, {})'.format(a, b))
    return a * b


def make_call(a, b):
    print('({}, {})'.format(a, b), end=' ')
    pre_hits = expensive.cache_info().hits
    expensive(a, b)
    post_hits = expensive.cache_info().hits
    if post_hits > pre_hits:
        print('cache hit')


make_call(1, 2)

try:
    make_call([1], 2)
except TypeError as err:
    print('ERROR: {}'.format(err))

try:
    make_call(1, {'2': 'two'})
except TypeError as err:
    print('ERROR: {}'.format(err))
```

Если функции передается объект, который нельзя хешировать, возбуждается исключение TypeError.

```
$ python3 functools_lru_cache_arguments.py

(1, 2) called expensive(1, 2)
([1], 2) ERROR: unhashable type: 'list'
(1, {'2': 'two'}) ERROR: unhashable type: 'dict'
```

3.1.4. Редукция набора данных

Функция reduce() получает вызываемый объект и последовательность данных в качестве аргументов. Результатом ее работы является единственное значение, основанное на вызове объекта со значениями из последовательности и накоплении результатов.

Листинг 3.11. functools_reduce.py

```
import functools


def do_reduce(a, b):
    print('do_reduce({}, {})'.format(a, b))
    return a + b


data = range(1, 5)
print(data)
result = functools.reduce(do_reduce, data)
print('result: {}'.format(result))
```

В этом примере суммируются числа, образующие входную последовательность.

```
$ python3 functools_reduce.py

range(1, 5)
do_reduce(1, 2)
do_reduce(3, 3)
do_reduce(6, 4)
result: 10
```

Необязательный аргумент initializer помещается перед последовательностью и обрабатывается наряду с другими элементами. Это может быть использовано для обновления ранее вычисленного значения новыми входными данными.

Листинг 3.12. functools_reduce_initializer.py

```
import functools


def do_reduce(a, b):
    print('do_reduce({}, {})'.format(a, b))
```

```
    return a + b


data = range(1, 5)
print(data)
result = functools.reduce(do_reduce, data, 99)
print('result: {}'.format(result))
```

В этом примере ранее найденная сумма, равная 99, используется для инициализации значения, вычисляемого функцией `reduce()`.

```
$ python3 functools_reduce_initializer.py

range(1, 5)

do_reduce(99, 1)
do_reduce(100, 2)
do_reduce(102, 3)
do_reduce(105, 4)
result: 109
```

Последовательности с единственным элементом редуцируются до этого значения, если не предоставлен аргумент `initializer`. В отсутствие этого аргумента пустые списки генерируют ошибку.

Листинг 3.13. `functools_reduce_short_sequences.py`

```
import functools


def do_reduce(a, b):
    print('do_reduce({}, {})'.format(a, b))
    return a + b


print('Single item in sequence:',
    functools.reduce(do_reduce, [1]))

print('Single item in sequence with initializer:',
    functools.reduce(do_reduce, [1], 99))

print('Empty sequence with initializer:',
    functools.reduce(do_reduce, [], 99))

try:
    print('Empty sequence:', functools.reduce(do_reduce, []))
except TypeError as err:
    print('ERROR: {}'.format(err))
```

Поскольку аргумент `initializer` выступает в качестве значения по умолчанию и при этом сочетается с новыми значениями, если входная последовательность не является пустой, очень важно тщательно проверять, используется ли он надлежащим образом. Если в сочетании значения по умолчанию с новыми значе-

ниями нет смысла, лучше перехватывать исключение TypeError, чем передавать аргумент initializer.

```
$ python3 functools_reduce_short_sequences.py

Single item in sequence: 1
do_reduce(99, 1)
Single item in sequence with initializer: 100
Empty sequence with initializer: 99
ERROR: reduce() of empty sequence with no initial value
```

3.1.5. Обобщенные функции

В языках программирования с динамической типизацией, таких как Python, часто возникает необходимость в изменении характера выполнения операций в зависимости от типа аргументов, особенно если речь идет о проведении различий между списком элементов и одиночным элементом. Непосредственная проверка типов не вызывает затруднений, однако в тех случаях, когда различия в поведении могут быть вынесены в отдельные функции, модуль functools предоставляет декоратор singledispatch(), который обеспечивает регистрацию набора *обобщенных функций* с возможностью автоматического выбора нужной функции на основании типа первого аргумента.

Листинг 3.14. functools_singledispatch.py

```
import functools


@functools.singledispatch
def myfunc(arg):
    print('default myfunc({!r})'.format(arg))


@myfunc.register(int)
def myfunc_int(arg):
    print('myfunc_int({})'.format(arg))


@myfunc.register(list)
def myfunc_list(arg):
    print('myfunc_list()')
    for item in arg:
        print(' {}'.format(item))


myfunc('string argument')
myfunc(1)
myfunc(2.3)
myfunc(['a', 'b', 'c'])
```

Атрибут register() новой функции служит дополнительным декоратором для регистрации альтернативной реализации. Первая функция, обернутая деко-

ратором singledispatch(), является реализацией по умолчанию, если не обнаружена никакая другая функция, специфическая для данного типа, как в случае типа float в этом примере.

```
$ python3 functools_singledispatch.py

default myfunc('string argument')
myfunc_int(1)
default myfunc(2.3)
myfunc_list()
  a
  b
  c
```

Если не удается найти точного соответствия типу, определяется порядок наследования и используется наиболее соответствующий тип.

Листинг 3.15. functools_singledispatch_mro.py

```
import functools


class A:
    pass


class B(A):
    pass


class C(A):
    pass


class D(B):
    pass


class E(C, D):
    pass


@functools.singledispatch
def myfunc(arg):
    print('default myfunc({})'.format(arg.__class__.__name__))


@myfunc.register(A)
def myfunc_A(arg):
    print('myfunc_A({})'.format(arg.__class__.__name__))


@myfunc.register(B)
```

```
def myfunc_B(arg):
    print('myfunc_B({})'.format(arg.__class__.__name__))


@myfunc.register(C)
def myfunc_C(arg):
    print('myfunc_C({})'.format(arg.__class__.__name__))


myfunc(A())
myfunc(B())
myfunc(C())
myfunc(D())
myfunc(E())
```

В этом примере для классов `D` и `E` не находится точного соответствия с какой-либо зарегистрированной обобщенной функцией, и выбор функции зависит от иерархии классов.

```
$ python3 functools_singledispatch_mro.py

myfunc_A(A)
myfunc_B(B)
myfunc_C(C)
myfunc_B(D)
myfunc_C(E)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `functools`[1].
- Методы расширенного сравнения[2]. Описание методов расширенных операций сравнения в справочном руководстве Python.
- *Isolated @memoize*[3] (Ned Batchelder). Статья, посвященная созданию декораторов мемоизации, которые хорошо сочетаются с блочными тестами.
- **PEP 443**[4]. *Single-dispatch generic functions*. Описание динамической диспетчеризации вызовов обобщенных функций на основании одного аргумента.
- Модуль `inspect` (раздел 18.4). API интроспекции для активных объектов.

3.2. `itertools`: функции-итераторы

Модуль `itertools` включает ряд функций, предназначенных для обработки последовательностей. Их прообразом послужили аналогичные средства таких языков функционального программирования, как Clojure, Haskell, APL и SML. Указанные функции предназначены для повышения эффективности использова-

[1] https://docs.python.org/3.5/library/functools.html

[2] https://docs.python.org/reference/datamodel.html#object.__lt__

[3] http://nedbatchelder.com/blog/201601/isolated_memoize.html

[4] www.python.org/dev/peps/pep-0443

ния памяти и ускорения выполнения операций. Их также можно использовать совместно для создания более сложных итерационных алгоритмов.

Основанный на итераторах код обеспечивает лучшие характеристики использования памяти, чем код, основанный на использовании списков. Поскольку итератор не возвращает данные до тех пор, пока они не потребуются, отпадает необходимость в хранении всего набора данных в памяти. Такая модель отложенной обработки сглаживает отрицательное влияние подкачки и других побочных эффектов, связанных с обработкой больших объемов данных, на производительность.

Кроме функций, определенных в модуле `itertools`, в приведенных ниже примерах используются некоторые встроенные функции, предназначенные для работы с итераторами.

3.2.1. Объединение и разделение итераторов

Функция `chain()` получает несколько итераторов в качестве аргументов и создает итератор, который поочередно обрабатывает все входные итерируемые объекты, возвращая их элементы, как если бы они принадлежали одному входному итератору.

Листинг 3.16. `itertools_chain.py`

```
from itertools import *

for i in chain([1, 2, 3], ['a', 'b', 'c']):
    print(i, end=' ')
print()
```

Функция `chain()` упрощает обработку нескольких последовательностей, не требуя предварительного конструирования объединенного списка.

```
$ python3 itertools_chain.py

1 2 3 a b c
```

В тех случаях, когда объединяемые итерируемые объекты неизвестны заранее или должны определяться в режиме отложенных вычислений, для конструирования цепочки итераторов можно использовать функцию `chain.from_iterable()`.

Листинг 3.17. `itertools_chain_from_iterable.py`

```
from itertools import *


def make_iterables_to_chain():
    yield [1, 2, 3]
    yield ['a', 'b', 'c']


for i in chain.from_iterable(make_iterables_to_chain()):
    print(i, end=' ')
print()
```

```
$ python3 itertools_chain_from_iterable.py
1 2 3 a b c
```

Встроенная функция `zip()` возвращает итератор, объединяющий элементы нескольких итераторов в кортежи.

Листинг 3.18. `itertools_zip.py`

```
for i in zip([1, 2, 3], ['a', 'b', 'c']):
    print(i)
```

Как и в случае других функций, входящих в состав этого модуля, возвращаемое значение представляет собой итерируемый объект, выдающий значения по одному за раз.

```
$ python3 itertools_zip.py

(1, 'a')
(2, 'b')
(3, 'c')
```

Функция `zip()` прекращает работу, как только исчерпывается один из входных итераторов. Чтобы обеспечить обработку всех входных элементов даже в тех случаях, когда итераторы вырабатывают разное количество значений, используйте функцию `zip_longest()`.

Листинг 3.19. `itertools_zip_longest.py`

```
from itertools import *

r1 = range(3)
r2 = range(2)

print('zip stops early:')
print(list(zip(r1, r2)))

r1 = range(3)
r2 = range(2)

print('\nzip_longest processes all of the values:')
print(list(zip_longest(r1, r2)))
```

По умолчанию функция `zip_longest()` подставляет значение `None` вместо отсутствующих значений. Чтобы определить другое подстановочное значение, используйте аргумент `fillvalue`.

```
$ python3 itertools_zip_longest.py

zip stops early:
[(0, 0), (1, 1)]

zip_longest processes all of the values:
[(0, 0), (1, 1), (2, None)]
```

Функция `islice()` создает итератор, который возвращает итератор, выдающий входные элементы в соответствии с заданными индексами.

Листинг 3.20. `itertools_islice.py`

```
from itertools import *

print('Stop at 5:')
for i in islice(range(100), 5):
    print(i, end=' ')
print('\n')

print('Start at 5, Stop at 10:')
for i in islice(range(100), 5, 10):
    print(i, end=' ')
print('\n')

print('By tens to 100:')
for i in islice(range(100), 0, 100, 10):
    print(i, end=' ')
print('\n')
```

Функция `islice()` имеет те же аргументы, что и оператор взятия среза списка: `start`, `stop` и `step`. Аргументы `start` и `step` — необязательные.

```
$ python3 itertools_islice.py

Stop at 5:
0 1 2 3 4

Start at 5, Stop at 10:
5 6 7 8 9

By tens to 100:
0 10 20 30 40 50 60 70 80 90
```

Функция `tee()` позволяет создать несколько независимых итераторов (по умолчанию — 2) на основе одного и того же итерируемого объекта.

Листинг 3.21. `itertools_tee.py`

```
from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)

print('i1:', list(i1))
print('i2:', list(i2))
```

Семантика функции `tee()` аналогична семантике утилиты `tee` в Unix, которая повторяет значения, читаемые из входного потока, и записывает их в именованный файл или стандартный вывод. Итераторы, возвращаемые функцией `tee()`, могут быть использованы с целью передачи одного и того же набора данных нескольким алгоритмам для их параллельной обработки.

```
$ python3 itertools_tee.py

i1: [0, 1, 2, 3, 4]
i2: [0, 1, 2, 3, 4]
```

Новые итераторы, созданные функцией `tee()`, разделяют данные с исходным итератором, и поэтому после их создания исходный итератор использоваться не должен.

Листинг 3.22. `itertools_tee_error.py`

```
from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)

print('r:', end=' ')
for i in r:
    print(i, end=' ')
    if i > 1:
        break
print()

print('i1:', list(i1))
print('i2:', list(i2))
```

Значения, обработанные ранее исходным итератором, не будут возвращаться вновь созданными итераторами.

```
$ python3 itertools_tee_error.py

r: 0 1 2
i1: [3, 4]
i2: [3, 4]
```

3.2.2. Преобразование входных данных

Встроенная функция `map()` создает итератор, который вызывает заданную преобразующую функцию с аргументами, определяемыми значениями входных итераторов, и возвращает результаты. Этот процесс прекращается после исчерпания значений любого из входных итераторов.

Листинг 3.23. `itertools_map.py`

```
def times_two(x):
    return 2 * x


def multiply(x, y):
    return (x, y, x * y)


print('Doubles:')
```

```
for i in map(times_two, range(5)):
    print(i)

print('\nMultiples:')
r1 = range(5)
r2 = range(5, 10)
for i in map(multiply, r1, r2):
    print('{:d} * {:d} = {:d}'.format(*i))

print('\nStopping:')
r1 = range(5)
r2 = range(2)
for i in map(multiply, r1, r2):
    print(i)
```

В первом примере лямбда-функция умножает входные значения на 2. Во втором примере лямбда-функция перемножает пару аргументов, получаемых из независимых итераторов, и возвращает кортеж, включающий исходные аргументы и вычисленное значение. Выполнение третьего примера прекращается после создания двух кортежей ввиду исчерпания второго диапазона.

```
$ python3 itertools_map.py

Doubles:
0
2
4
6
8

Multiples:
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36

Stopping:
(0, 0, 0)
(1, 1, 1)
```

Функция `starmap()` аналогична функции `map()`, но имеет всего два аргумента: лямбда-функцию и последовательность кортежей, задаваемую с помощью синтаксиса *, в которой каждый кортеж предоставляет аргументы лямбда-функции.

Листинг 3.24. `itertools_starmap.py`

```
from itertools import *

values = [(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]

for i in starmap(lambda x, y: (x, y, x * y), values):
    print('{} * {} = {}'.format(*i))
```

Если преобразующая функция, передаваемая функции `map()`, вызывается как `f(i1,i2)`, то преобразующая функция, передаваемая функции `starmap()`, вызывается как `f(*i)`.

```
$ python3 itertools_starmap.py

0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

3.2.3. Создание новых значений

Функция `count()` создает итератор, вырабатывающий бесконечную последовательность целых чисел. Начальное значение может быть указано в качестве аргумента (по умолчанию – 0). Аргумент для задания верхней границы не предусмотрен (более полный контроль над результирующим набором значений обеспечивает встроенная функция `range()`).

Листинг 3.25. `itertools_count.py`

```
from itertools import *

for i in zip(count(1), ['a', 'b', 'c']):
    print(i)
```

Выполнение примера прекращается по исчерпании элементов списка, заданного в качестве аргумента.

```
$ python3 itertools_count.py

(1, 'a')
(2, 'b')
(3, 'c')
```

Аргументами `start` и `step` функции `count()` могут быть любые числовые значения, допускающие операцию сложения.

Листинг 3.26. `itertools_count_step.py`

```
import fractions
from itertools import *

start = fractions.Fraction(1, 3)
step = fractions.Fraction(1, 3)

for i in zip(count(start, step), ['a', 'b', 'c']):
    print('{}: {}'.format(*i))
```

В этом примере начальное значение и шаг представлены объектами `Fraction` из модуля `fraction`.

```
$ python3 itertools_count_step.py

1/3: a
2/3: b
1: c
```

Функция `cycle()` создает итератор, повторяющий содержимое аргументов бесконечное количество раз. Поскольку эта функция должна запоминать все содержимое входного итератора, она может потреблять много памяти в случае длинных входных последовательностей.

Листинг 3.27. `itertools_cycle.py`

```
from itertools import *

for i in zip(range(7), cycle(['a', 'b', 'c'])):
    print(i)
```

В этом примере выполнение цикла прерывается по исчерпании значений счетчика.

```
$ python3 itertools_cycle.py

(0, 'a')
(1, 'b')
(2, 'c')
(3, 'a')
(4, 'b')
(5, 'c')
(6, 'a')
```

Функция `repeat()` создает итератор, возвращающий одно и то же значение при каждом обращении к нему.

Листинг 3.28. `itertools_repeat.py`

```
from itertools import *

for i in repeat('over-and-over', 5):
    print(i)
```

Итератор, созданный функцией `repeat()`, может возвращать данные бесконечное число раз, однако количество повторений можно ограничить с помощью необязательного аргумента `times`.

```
$ python3 itertools_repeat.py

over-and-over
over-and-over
over-and-over
over-and-over
over-and-over
```

Функцию `repeat()` удобно использовать совместно с функциями `zip()` и `map()`, если со значениями, возвращаемыми другими итераторами, должно сочетаться некое инвариантное значение.

Листинг 3.29. `itertools_repeat_zip.py`

```
from itertools import *

for i, s in zip(count(), repeat('over-and-over', 5)):
    print(i, s)
```

В этом примере значение счетчика объединяется с константой, возвращаемой функцией `repeat()`.

```
$ python3 itertools_repeat_zip.py

0 over-and-over
1 over-and-over
2 over-and-over
3 over-and-over
4 over-and-over
```

В этом примере функция `map()` используется для умножения на 2 чисел в диапазоне 0–4.

Листинг 3.30. `itertools_repeat_map.py`

```
from itertools import *

for i in map(lambda x, y: (x, y, x * y), repeat(2), range(5)):
    print('{:d} * {:d} = {:d}'.format(*i))
```

В данном случае итератор, возвращаемый функцией `repeat()`, не нуждается в явном ограничении числа повторений, поскольку обработка с помощью функции `map()` прекращается сразу же, как только исчерпывается любой из ее входных итераторов, а функция `range()` возвращает только пять элементов.

```
$ python3 itertools_repeat_map.py

2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

3.2.4. Фильтрация

Функция `dropwhile()` создает итератор, который начинает воспроизводить элементы входного итератора сразу же после того, как для заданного условия будет получено ложное значение.

Листинг 3.31. itertools_dropwhile.py

```
from itertools import *


def should_drop(x):
    print('Testing:', x)
    return x < 1

for i in dropwhile(should_drop, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

Функция dropwhile() не тестирует все элементы входной последовательности. Как только условие принимает ложное значение, она начинает возвращать все оставшиеся элементы.

```
$ python3 itertools_dropwhile.py

Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Yielding: 2
Yielding: -2
```

Действия функции takewhile() противоположны действиям функции dropwhile(). Она создает итератор, который выдает элементы из входного итератора, пока тестирующая функция возвращает истинное значение.

Листинг 3.32. itertools_takewhile.py

```
from itertools import *


def should_take(x):
    print('Testing:', x)
    return x < 2

for i in takewhile(should_take, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

Как только функция should_take() возвращает ложное значение, функция takewhile() прекращает обработку входной последовательности.

```
$ python3 itertools_takewhile.py

Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Yielding: 1
Testing: 2
```

Встроенная функция `filter()` создает итератор, включающий только те элементы, для которых тестирующая функция возвращает истинное значение.

Листинг 3.33. itertools_filter.py

```
from itertools import *


def check_item(x):
    print('Testing:', x)
    return x < 1

for i in filter(check_item, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

Функция `filter()` отличается от функций `dropwhile()` и `takewhile()` тем, что тестирует каждый элемент входной последовательности, прежде чем вернуть его.

```
$ python3 itertools_filter.py

Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Testing: 2
Testing: -2
Yielding: -2
```

Функция `filterfalse()` создает итератор, включающий лишь те элементы, для которых тестирующая функция возвращает ложное значение.

Листинг 3.34. itertools_filterfalse.py

```
from itertools import *


def check_item(x):
    print('Testing:', x)
    return x < 1

for i in filterfalse(check_item, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

Тестовое выражение в `check_item()` осталось прежним, поэтому результаты, полученные в данном примере с использованием функции `filterfalse()`, противоположны тем, которые были получены в предыдущем примере.

```
$ python3 itertools_filterfalse.py

Testing: -1
Testing: 0
Testing: 1
```

```
Yielding: 1
Testing: 2
Yielding: 2
Testing: -2
```

Функция `compress()` предлагает другой способ фильтрации содержимого итерируемого объекта. Вместо того чтобы вызывать функцию, она использует значения другого итерируемого объекта для индикации того, следует ли принять значение или игнорировать его.

Листинг 3.35. `itertools_compress.py`

```
from itertools import *

every_third = cycle([False, False, True])
data = range(1, 10)

for i in compress(data, every_third):
    print(i, end=' ')
print()
```

Первый аргумент — это итерируемые данные, подлежащие обработке. Второй аргумент — это итерируемый селектор, который предоставляет значения, указывающие на то, какие входные значения следует принимать (истинному значению соответствует воспроизведение входных значений, ложному — их игнорирование).

```
$ python3 itertools_compress.py

3 6 9
```

3.2.5. Группирование данных

Функция `groupby()` возвращает итератор, который создает наборы значений, группируемые в соответствии с общим признаком. В следующем примере продемонстрировано группирование родственных значений на основании значений атрибута.

Листинг 3.36. `itertools_groupby_seq.py`

```
import functools
from itertools import *
import operator
import pprint


@functools.total_ordering
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __repr__(self):
        return '({}, {})'.format(self.x, self.y)

    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)

    def __gt__(self, other):
        return (self.x, self.y) > (other.x, other.y)


# Создание набора данных из экземпляров Point
data = list(map(Point,
                cycle(islice(count(), 3)),
                islice(count(), 7)))
print('Data:')
pprint.pprint(data, width=35)
print()

# Попытаться сгруппировать несортированные данные
# на основании значений X
print('Grouped, unsorted:')
for k, g in groupby(data, operator.attrgetter('x')):
    print(k, list(g))
print()

# Сортировка данных
data.sort()
print('Sorted:')
pprint.pprint(data, width=35)
print()

# Группирование сортированных данных на основании значений X
print('Grouped, sorted:')
for k, g in groupby(data, operator.attrgetter('x')):
    print(k, list(g))
print()
```

Чтобы группирование работало так, как ожидается, входная последовательность должна быть отсортирована по ключу.

```
$ python3 itertools_groupby_seq.py

Data:
[(0, 0),
 (1, 1),
 (2, 2),
 (0, 3),
 (1, 4),
 (2, 5),
 (0, 6)]

Grouped, unsorted:
0 [(0, 0)]
```

```
1 [(1, 1)]
2 [(2, 2)]
0 [(0, 3)]
1 [(1, 4)]
2 [(2, 5)]
0 [(0, 6)]

Sorted:
[(0, 0),
 (0, 3),
 (0, 6),
 (1, 1),
 (1, 4),
 (2, 2),
 (2, 5)]

Grouped, sorted:
0 [(0, 0), (0, 3), (0, 6)]
1 [(1, 1), (1, 4)]
2 [(2, 2), (2, 5)]
```

3.2.6. Комбинирование входных данных

Функция `accumulate()` обрабатывает входные итерируемые объекты, передавая n-й и (n+1)-й элементы функции и возвращая получаемый с помощью этой функции результат вместо входных значений. Функция по умолчанию суммирует два значения, поэтому функцию `accumulate()` можно использовать для получения накопительной суммы входных данных.

Листинг 3.37. `itertools_accumulate.py`

```
from itertools import *

print(list(accumulate(range(5))))
print(list(accumulate('abcde')))
```

В случае применения этой функции к значениям, не являющимся целыми числами, результат зависит от смысла операции "сложения", объединяющей оба элемента. Как показывает второй из приведенных в этом сценарии примеров, при передаче функции `accumulate()` строки она возвращает с каждым разом все более удлиняющийся префикс этой строки.

```
$ python3 itertools_accumulate.py

[0, 1, 3, 6, 10]
['a', 'ab', 'abc', 'abcd', 'abcde']
```

Функцию `accumulate()` можно использовать в сочетании с любой другой функцией, принимающей два входных значения, для получения других результатов.

Листинг 3.38. `itertools_accumulate_custom.py`

```
from itertools import *


def f(a, b):
    print(a, b)
    return b + a + b

print(list(accumulate('abcde', f)))
```

В этом примере в результате комбинирования строк образуется серия палиндромов (не несущих в себе никакого смысла). Каждый раз, когда вызывается функция `f()`, она выводит входные значения, переданные ей функцией `accumulate()`.

```
$ python3 itertools_accumulate_custom.py

a b
bab c
cbabc d
dcbabcd e
['a', 'bab', 'cbabc', 'dcbabcd', 'edcbabcde']
```

Вместо вложенных циклов, выполняющих итерации по нескольким последовательностям, часто можно использовать функцию `product()`, возвращающую итерируемый объект, значениями которого являются декартовы произведения входных значений.

Листинг 3.39. `itertools_product.py`

```
from itertools import *
import pprint

FACE_CARDS = ('J', 'Q', 'K', 'A')
SUITS = ('H', 'D', 'C', 'S')

DECK = list(
    product(
        chain(range(2, 11), FACE_CARDS),
        SUITS,
    )
)

for card in DECK:
    print('{:>2}{}'.format(*card), end=' ')
    if card[1] == SUITS[-1]:
        print()
```

Значения, возвращаемые функцией `product()`, представляют собой кортежи, элементы которых берутся из каждого итерируемого объекта, переданного данной функции в качестве аргумента, в том порядке, в котором они передавались. Первый кортеж включает первое значение из каждого итерируемого объекта. *Последний* итерируемый объект, переданный функции `product()`, обрабатывает-

ся первым, затем обрабатывается предпоследний итерируемый объект и т.д. В результате этого порядок расположения возвращаемых значений основывается сначала на первом итерируемом объекте, затем на следующем и т.д.

В этом примере карты упорядочиваются сначала по старшинству, а затем по масти.

```
$ python3 itertools_product.py

 2H  2D  2C  2S
 3H  3D  3C  3S
 4H  4D  4C  4S
 5H  5D  5C  5S
 6H  6D  6C  6S
 7H  7D  7C  7S
 8H  8D  8C  8S
 9H  9D  9C  9S
10H 10D 10C 10S
 JH  JD  JC  JS
 QH  QD  QC  QS
 KH  KD  KC  KS
 AH  AD  AC  AS
```

Порядок следования карт можно изменить, поменяв порядок передачи аргументов функции `product()`.

Листинг 3.40. itertools_product_ordering.py

```
from itertools import *
import pprint

FACE_CARDS = ('J', 'Q', 'K', 'A')
SUITS = ('H', 'D', 'C', 'S')

DECK = list(
    product(
        SUITS,
        chain(range(2, 11), FACE_CARDS),
    )
)

for card in DECK:
    print('{:>2}{}'.format(card[1], card[0]), end=' ')
    if card[1] == FACE_CARDS[-1]:
        print()
```

В этом примере ожидается появление туза, а не пиковой масти, после чего добавляется символ перевода строки и выводится следующая строка.

```
$ python3 itertools_product_ordering.py

 2H  3H  4H  5H  6H  7H  8H  9H 10H  JH  QH  KH  AH
 2D  3D  4D  5D  6D  7D  8D  9D 10D  JD  QD  KD  AD
```

```
 2C  3C  4C  5C  6C  7C  8C  9C 10C  JC  QC  KC  AC
 2S  3S  4S  5S  6S  7S  8S  9S 10S  JS  QS  KS  AS
```

Чтобы вычислить произведение последовательности на саму себя, следует указать количество повторений.

Листинг 3.41. `itertools_product_repeat.py`

```
from itertools import *


def show(iterable):
    for i, item in enumerate(iterable, 1):
        print(item, end=' ')
        if (i % 3) == 0:
            print()
    print()


print('Repeat 2:\n')
show(list(product(range(3), repeat=2)))

print('Repeat 3:\n')
show(list(product(range(3), repeat=3)))
```

Поскольку повторение одиночного итерируемого объекта определенное количество раз равносильно передаче того же количества одинаковых итерируемых объектов, количество элементов в каждом кортеже, создаваемом функцией `product()`, будет равно счетчику повторений.

```
$ python3 itertools_product_repeat.py

Repeat 2:

(0, 0) (0, 1) (0, 2)
(1, 0) (1, 1) (1, 2)
(2, 0) (2, 1) (2, 2)

Repeat 3:

(0, 0, 0) (0, 0, 1) (0, 0, 2)
(0, 1, 0) (0, 1, 1) (0, 1, 2)
(0, 2, 0) (0, 2, 1) (0, 2, 2)
(1, 0, 0) (1, 0, 1) (1, 0, 2)
(1, 1, 0) (1, 1, 1) (1, 1, 2)
(1, 2, 0) (1, 2, 1) (1, 2, 2)
(2, 0, 0) (2, 0, 1) (2, 0, 2)
(2, 1, 0) (2, 1, 1) (2, 1, 2)
(2, 2, 0) (2, 2, 1) (2, 2, 2)
```

Функция `permutations()` создает итератор, который возвращает все возможные перестановки заданной длины, образуемые из элементов входного итерируемого объекта.

Листинг 3.42. itertools_permutations.py

```
from itertools import *


def show(iterable):
    first = None
    for i, item in enumerate(iterable, 1):
        if first != item[0]:
            if first is not None:
                print()
            first = item[0]
        print(''.join(item), end=' ')
    print()


print('All permutations:\n')
show(permutations('abcd'))

print('\nPairs:\n')
show(permutations('abcd', r=2))
```

Для изменения длины возвращаемых перестановок и их количества используйте аргумент `r`.

```
$ python3 itertools_permutations.py

All permutations:

abcd abdc acbd acdb adbc adcb
bacd badc bcad bcda bdac bdca
cabd cadb cbad cbda cdab cdba
dabc dacb dbac dbca dcab dcba

Pairs:

ab ac ad
ba bc bd
ca cb cd
da db dc
```

Чтобы ограничить круг значений уникальными сочетаниями, а не перестановками, используйте функцию `combinations()`. Коль скоро все элементы входной последовательности являются уникальными, выходные последовательности не будут включать повторяющихся значений.

Листинг 3.43. itertools_combinations.py

```
from itertools import *


def show(iterable):
    first = None
    for i, item in enumerate(iterable, 1):
```

```
        if first != item[0]:
            if first is not None:
                print()
            first = item[0]
        print(''.join(item), end=' ')
    print()


print('Unique pairs:\n')
show(combinations('abcd', r=2))
```

В отличие от функции `permutations()`, аргумент `r` функции `combinations()` является обязательным.

```
$ python3 itertools_combinations.py

Unique pairs:

ab ac ad
bc bd
cd
```

В то время как функция `combinations()` не повторяет индивидуальные элементы входной последовательности, иногда полезно рассматривать сочетания, допускающие повторение элементов. В подобных случаях следует использовать функцию `combinations_with_replacement()`.

Листинг 3.44. `itertools_combinations_with_replacement.py`

```
from itertools import *


def show(iterable):
    first = None
    for i, item in enumerate(iterable, 1):
        if first != item[0]:
            if first is not None:
                print()
            first = item[0]
        print(''.join(item), end=' ')
    print()


print('Unique pairs:\n')
show(combinations_with_replacement('abcd', r=2))
```

На этот раз каждый входной элемент образует сочетание с самим собой, а также с другими элементами входной последовательности.

```
$ python3 itertools_combinations_with_replacement.py

Unique pairs:
```

```
aa ab ac ad
bb bc bd
cc cd
dd
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `itertools`[5].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `itertools` (раздел A.6.22).
- *The Standard ML Basis Library*[6]. Описание библиотеки SML.
- *Definition of Haskell and the Standard Libraries*[7]. Спецификация стандартной библиотеки языка функционального программирования Haskell.
- *Clojure*[8]. Clojure — динамический язык функционального программирования, выполняющийся в среде виртуальной машины Java.
- **tee**[9]. Утилита командной строки Unix, предназначенная для разветвления одного входного потока на несколько идентичных выходных потоков.
- Википедия: *Декартово произведение*[10]. Математическое определение прямого (декартового) произведения двух последовательностей.

3.3. `operator`: функциональный интерфейс встроенных операторов

Время от времени в процессе программирования с использованием итераторов возникает необходимость в создании небольших функций для вычисления простых выражений. Иногда это можно реализовать с помощью лямбда-функций, но для некоторых операций новые функции вообще не нужны. Модуль `operator` содержит функции, которые соответствуют встроенным арифметическим операторам, операторам сравнения и другим стандартным операторам.

3.3.1. Логические операции

Модуль `operator` предоставляет функции для определения булевых эквивалентов значений, отрицания значений для получения противоположного булевого значения, а также сравнения объектов с целью проверки их идентичности.

Листинг 3.45. `operator_boolean.py`

```
from operator import *

a = -1
b = 5
```

[5] https://docs.python.org/3.5/library/itertools.html
[6] www.standardml.org/Basis/
[7] www.haskell.org/definition/
[8] http://clojure.org
[9] http://man7.org/linux/man-pages/man1/tee.1.html
[10] https://ru.wikipedia.org/wiki/Прямое_произведение

```
print('a =', a)
print('b =', b)
print()

print('not_(a)     :', not_(a))
print('truth(a)    :', truth(a))
print('is_(a, b)   :', is_(a, b))
print('is_not(a, b):', is_not(a, b))
```

В название функции `not_()` включен символ подчеркивания, поскольку в Python существует ключевое слово `not`. Функция `truth()` возвращает значение `True`, если аргумент имеет истинное значение, и `False` — в противном случае. Функция `is_()` реализует ту же проверку, что и ключевое слово `is`, а функция `is_not()` выполняет ту же проверку, но возвращает противоположный результат.

```
$ python3 operator_boolean.py

a = -1
b = 5

not_(a)     : False
truth(a)    : True
is_(a, b)   : False
is_not(a, b): True
```

3.3.2. Операторы сравнения

Поддерживаются все операторы расширенного сравнения.

Листинг 3.46. `operator_comparisons.py`

```
from operator import *

a = 1
b = 5.0

print('a =', a)
print('b =', b)
for func in (lt, le, eq, ne, ge, gt):
    print('{}(a, b): {}'.format(func.__name__, func(a, b)))
```

Соответствующие функции эквивалентны использованию синтаксиса выражений с операторами <, <=, ==, >= и >.

```
$ python3 operator_comparisons.py

a = 1
b = 5.0
lt(a, b): True
le(a, b): True
eq(a, b): False
ne(a, b): True
ge(a, b): False
gt(a, b): False
```

3.3.3. Арифметические операторы

Поддерживаются также арифметические операторы для манипулирования числовыми значениями.

Листинг 3.47. `operator_math.py`

```
from operator import *

a = -1
b = 5.0
c = 2
d = 6

print('a =', a)
print('b =', b)
print('c =', c)
print('d =', d)

print('\nPositive/Negative:')
print('abs(a):', abs(a))
print('neg(a):', neg(a))
print('neg(b):', neg(b))
print('pos(a):', pos(a))
print('pos(b):', pos(b))

print('\nArithmetic:')
print('add(a, b) :', add(a, b))
print('floordiv(a, b):', floordiv(a, b))
print('floordiv(d, c):', floordiv(d, c))
print('mod(a, b) :', mod(a, b))
print('mul(a, b) :', mul(a, b))
print('pow(c, d) :', pow(c, d))
print('sub(b, a) :', sub(b, a))
print('truediv(a, b) :', truediv(a, b))
print('truediv(d, c) :', truediv(d, c))

print('\nBitwise:')
print('and_(c, d) :', and_(c, d))
print('invert(c) :', invert(c))
print('lshift(c, d):', lshift(c, d))
print('or_(c, d) :', or_(c, d))
print('rshift(d, c):', rshift(d, c))
print('xor(c, d) :', xor(c, d))
```

Предоставляются две различные операции деления: `floordiv()` (целочисленное деление в том виде, как оно реализовано в версиях Python, предшествующих версии 3.0) и `truediv()` (деление чисел с плавающей точкой).

```
$ python3 operator_math.py

a = -1
b = 5.0
c = 2
```

```
d = 6

Positive/Negative:
abs(a): 1
neg(a): 1
neg(b): -5.0
pos(a): -1
pos(b): 5.0

Arithmetic:
add(a, b)     : 4.0
floordiv(a, b): -1.0
floordiv(d, c): 3
mod(a, b)     : 4.0
mul(a, b)     : -5.0
pow(c, d)     : 64
sub(b, a)     : 6.0
truediv(a, b) : -0.2
truediv(d, c) : 3.0

Bitwise:
and_(c, d)  : 2
invert(c)   : -3
lshift(c, d): 128
or_(c, d)   : 6
rshift(d, c): 1
xor(c, d)   : 4
```

3.3.4. Операторы для работы с последовательностями

Операторы для работы с последовательностями можно разбить на четыре группы: создание последовательностей, поиск элементов, доступ к содержимому и удаление элементов последовательности.

Листинг 3.48. `operator_sequences.py`

```
from operator import *

a = [1, 2, 3]
b = ['a', 'b', 'c']

print('a =', a)
print('b =', b)

print('\nConstructive:')
print('  concat(a, b):', concat(a, b))

print('\nSearching:')
print('  contains(a, 1)  :', contains(a, 1))
print('  contains(b, "d"):', contains(b, "d"))
print('  countOf(a, 1)   :', countOf(a, 1))
print('  countOf(b, "d") :', countOf(b, "d"))
print('  indexOf(a, 5)   :', indexOf(a, 1))
```

```
print('\nAccess Items:')
print('  getitem(b, 1)                  :',
      getitem(b, 1))
print('  getitem(b, slice(1, 3))        :',
      getitem(b, slice(1, 3)))
print('  setitem(b, 1, "d")             :', end=' ')
setitem(b, 1, "d")
print(b)
print('  setitem(a, slice(1, 3), [4, 5]):', end=' ')
setitem(a, slice(1, 3), [4, 5])
print(a)

print('\nDestructive:')
print('  delitem(b, 1)          :', end=' ')
delitem(b, 1)
print(b)
print('  delitem(a, slice(1, 3)):', end=' ')
delitem(a, slice(1, 3))
print(a)
```

Некоторые из этих операций, такие как `setitem()` и `delitem()`, изменяют последовательность на месте и не возвращают никаких значений.

```
$ python3 operator_sequences.py

a = [1, 2, 3]
b = ['a', 'b', 'c']

Constructive:
  concat(a, b): [1, 2, 3, 'a', 'b', 'c']

Searching:
  contains(a, 1)  : True
  contains(b, "d"): False
  countOf(a, 1)   : 1
  countOf(b, "d") : 0
  indexOf(a, 5)   : 0

Access Items:
  getitem(b, 1)                  : b
  getitem(b, slice(1, 3))        : ['b', 'c']
  setitem(b, 1, "d")             : ['a', 'd', 'c']
  setitem(a, slice(1, 3), [4, 5]): [1, 4, 5]

Destructive:
  delitem(b, 1)          : ['a', 'c']
  delitem(a, slice(1, 3)): [1]
```

3.3.5. Операторы, изменяющие операнды

В дополнение к стандартным операторам многие типы объектов поддерживают также операции, выполняемые "на месте", т.е. с изменением самих операндов, как операция +=. Эквивалентные функции предусмотрены и для таких операторов.

Листинг 3.49. `operator_inplace.py`

```
from operator import *

a = -1
b = 5.0
c = [1, 2, 3]
d = ['a', 'b', 'c']
print('a =', a)
print('b =', b)
print('c =', c)
print('d =', d)
print()

a = iadd(a, b)
print('a = iadd(a, b) =>', a)
print()

c = iconcat(c, d)
print('c = iconcat(c, d) =>', c)
```

В приведенных примерах продемонстрирована работа лишь некоторых из этих функций. Для получения более подробных сведений о соответствующих функциях обратитесь к документации стандартной библиотеки.

```
$ python3 operator_inplace.py

a = -1
b = 5.0
c = [1, 2, 3]
d = ['a', 'b', 'c']

a = iadd(a, b) => 4.0

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

3.3.6. Функции доступа к элементам и атрибутам

Одна из наиболее необычных возможностей, предлагаемых модулем `operator`, связана с понятием "получателей свойств" (getters). Это понятие относится к вызываемым объектам, которые создаются во время выполнения программы и предназначены для получения атрибутов объектов или содержимого последовательностей. Получатели свойств особенно полезны при работе с итераторами или генераторами последовательностей, поскольку работают быстрее и потребляют меньше памяти, чем лямбда-функции и функции Python.

Листинг 3.50. `operator_attrgetter.py`

```
from operator import *


class MyObj:
    """Образец класса для attrgetter"""
```

```
    def __init__(self, arg):
        super().__init__()
        self.arg = arg

    def __repr__(self):
        return 'MyObj({})'.format(self.arg)


l = [MyObj(i) for i in range(5)]
print('objects :', l)

# Извлечение значения 'arg' из каждого объекта
g = attrgetter('arg')
vals = [g(i) for i in l]
print('arg values:', vals)

# Сортировка с использованием arg
l.reverse()
print('reversed :', l)
print('sorted :', sorted(l, key=g))
```

Получатели атрибутов работают аналогично лямбда-функции вида `lambda x,n='attrname': getattr(x,n)`.

```
$ python3 operator_attrgetter.py

objects   : [MyObj(0), MyObj(1), MyObj(2), MyObj(3), MyObj(4)]
arg values: [0, 1, 2, 3, 4]
reversed  : [MyObj(4), MyObj(3), MyObj(2), MyObj(1), MyObj(0)]
sorted    : [MyObj(0), MyObj(1), MyObj(2), MyObj(3), MyObj(4)]
```

Получатели элементов работают аналогично лямбда-функции вида `lambda x,y=5: x[y]`.

Листинг 3.51. `operator_itemgetter.py`

```
from operator import *

l = [dict(val=-1 * i) for i in range(4)]
print('Dictionaries:')
print(' original:', l)
g = itemgetter('val')
vals = [g(i) for i in l]
print(' values:', vals)
print(' sorted:', sorted(l, key=g))

print
l = [(i, i * -2) for i in range(4)]
print('\nTuples:')
print(' original:', l)
g = itemgetter(1)
vals = [g(i) for i in l]
print('   values:', vals)
print('   sorted:', sorted(l, key=g))
```

Получатели элементов работают как с последовательностями, так и с отображениями.

```
$ python3 operator_itemgetter.py

Dictionaries:
 original: [{'val': 0}, {'val': -1}, {'val': -2}, {'val': -3}]
  values: [0, -1, -2, -3]
  sorted: [{'val': -3}, {'val': -2}, {'val': -1}, {'val': 0}]

Tuples:
 original: [(0, 0), (1, -2), (2, -4), (3, -6)]
   values: [0, -2, -4, -6]
   sorted: [(3, -6), (2, -4), (1, -2), (0, 0)]
```

3.3.7. Сочетание операторов с пользовательскими классами

Функции, содержащиеся в модуле `operator`, выполняют свойственные им операции с использованием стандартных интерфейсов Python. Таким образом, они способны работать с пользовательскими классами так же, как и со встроенными типами.

Листинг 3.52. `operator_classes.py`

```
from operator import *


class MyObj:
    """Пример перегрузки операторов"""

    def __init__(self, val):
        super(MyObj, self).__init__()
        self.val = val

    def __str__(self):
        return 'MyObj({})'.format(self.val)

    def __lt__(self, other):
        """Сравнение меньше чем"""
        print('Testing {} < {}'.format(self, other))
        return self.val < other.val

    def __add__(self, other):
        """Суммирование значений"""
        print('Adding {} + {}'.format(self, other))
        return MyObj(self.val + other.val)

a = MyObj(1)
b = MyObj(2)

print('Comparison:')
print(lt(a, b))
```

```
print('\nArithmetic:')
print(add(a, b))
```

Для ознакомления с полным списком специальных методов, используемых каждым оператором, обратитесь к справочному руководству Python.

```
$ python3 operator_classes.py

Comparison:
Testing MyObj(1) < MyObj(2)
True

Arithmetic:
Adding MyObj(1) + MyObj(2)
MyObj(3)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `operator`[11].
- `functools` (раздел 3.1). Содержит инструменты функционального программирования, включая декоратор `total_ordering()`, обеспечивающий добавление методов расширенного сравнения в класс.
- `itertools` (раздел 3.2). Операции, выполняемые с помощью итераторов.
- `collections` (раздел 2.2). Абстрактные типы коллекций.
- `Numbers`. Абстрактный тип для числовых значений.

3.4. `contextlib`: утилиты менеджеров контекста

Модуль `contextlib` содержит вспомогательные функции для работы с менеджерами контекста и инструкцией `with`.

3.4.1. API менеджера контекста

Менеджер контекста отвечает за использование ресурсов в пределах блока кода, возможно, с созданием менеджера контекста при входе в блок и его последующим уничтожением с очисткой ресурсов при выходе из блока. Например, файлы поддерживают API менеджера контекста, гарантирующего закрытие файла по завершении чтения или записи данных.

Листинг 3.53. `contextlib_file.py`

```
with open('/tmp/pymotw.txt', 'wt') as f:
    f.write('contents go here')
# Файл автоматически закрывается
```

Менеджер контекста активизируется инструкцией `with`, а соответствующий API включает два метода. Метод `__enter__()` выполняется при входе в блок кода инструкции `with`. Он возвращает объект, который будет использоваться в пределах

[11] https://docs.python.org/3.5/library/operator.html

данного контекста. Когда поток управления покидает блок with, вызывается метод __exit__() менеджера контекста, освобождающий использованные ресурсы.

Листинг 3.54. contextlib_api.py

```
class Context:

    def __init__(self):
        print('__init__()')

    def __enter__(self):
        print('__enter__()')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__()')

with Context():
    print('Doing work in the context')
```

В сочетании с инструкцией with менеджер контекста обеспечивает наиболее компактный способ записи блока try:finally, поскольку метод __exit__() менеджера контекста вызывается в любом случае, даже если возникло исключение.

```
$ python3 contextlib_api.py

__init__()

__enter__()
Doing work in the context
__exit__()
```

Метод __enter__() может вернуть любой объект, который будет связан с именем, указанным в предложении as инструкции with. В следующем примере метод Context() возвращает объект, который использует открытый контекст.

Листинг 3.55. contextlib_api_other_object.py

```
class WithinContext:

    def __init__(self, context):
        print('WithinContext.__init__({})'.format(context))

    def do_something(self):
        print('WithinContext.do_something()')

    def __del__(self):
        print('WithinContext.__del__')


class Context:

    def __init__(self):
        print('Context.__init__()')
```

```
    def __enter__(self):
        print('Context.__enter__()')
        return WithinContext(self)

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('Context.__exit__()')

with Context() as c:
    c.do_something()
```

Значение, связываемое с объектом c, — это объект, возвращенный методом __enter__(), который не обязательно должен быть экземпляром Context, созданным в пределах инструкции with.

```
$ python3 contextlib_api_other_object.py

Context.__init__()
Context.__enter__()
WithinContext.__init__(<__main__.Context object at 0x1007b1c50>)
WithinContext.do_something()
Context.__exit__()
WithinContext.__del__
```

Метод __exit__() получает аргументы, содержащие подробную информацию о любом исключении, возникающем в пределах блока with.

Листинг 3.56. contextlib_api_error.py

```
class Context:


    def __init__(self, handle_error):
        print('__init__({})'.format(handle_error))
        self.handle_error = handle_error

    def __enter__(self):
        print('__enter__()')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__()')
        print(' exc_type =', exc_type)
        print(' exc_val =', exc_val)
        print(' exc_tb =', exc_tb)
        return self.handle_error

with Context(True):
    raise RuntimeError('error message handled')

print()

with Context(False):
    raise RuntimeError('error message propagated')
```

Если менеджер контекста может обрабатывать исключения, то метод `__exit__()` должен возвращать истинное значение, указывающее на то, что исключение не должно распространяться. Возврат ложного значения приводит к повторному возбуждению исключения после выхода из метода `__exit__()`.

```
$ python3 contextlib_api_error.py

__init__(True)
__enter__()
__exit__()
  exc_type = <class 'RuntimeError'>
  exc_val = error message handled
  exc_tb = <traceback object at 0x10115cc88>
__init__(False)
__enter__()
__exit__()
  exc_type = <class 'RuntimeError'>
  exc_val = error message propagated
  exc_tb = <traceback object at 0x10115cc88>
Traceback (most recent call last):
  File "contextlib_api_error.py", line 33, in <module>
  raise RuntimeError('error message propagated')
RuntimeError: error message propagated
```

3.4.2. Менеджеры контекста как декораторы функций

Класс `ContextDecorator` добавляет в классы контекстных менеджеров поддержку, позволяющую использовать их не только в качестве менеджеров контекста, но и в качестве декораторов функций.

Листинг 3.57. `contextlib_decorator.py`

```
import contextlib


class Context(contextlib.ContextDecorator):

    def __init__(self, how_used):
        self.how_used = how_used
        print('__init__({})'.format(how_used))

    def __enter__(self):
        print('__enter__({})'.format(self.how_used))
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__({})'.format(self.how_used))


@Context('as decorator')
def func(message):
    print(message)
```

```
print()
with Context('as context manager'):
    print('Doing work in the context')

print()
func('Doing work in the wrapped function')
```

Использование менеджера контекста в качестве декоратора отличается от его использования в инструкции with с предложением as, в частности, тем, что значение, возвращаемое методом __enter__(), недоступно в декорируемой функции. Аргументы, передаваемые декорированной функции, остаются доступными, как обычно.

```
$ python3 contextlib_decorator.py

__init__(as decorator)

__init__(as context manager)
__enter__(as context manager)
Doing work in the context
__exit__(as context manager)

__enter__(as decorator)
Doing work in the wrapped function
__exit__(as decorator)
```

3.4.3. От генератора к менеджеру контекста

Создание менеджеров контекста традиционным способом, т.е. посредством написания класса, содержащего методы __enter__() и __exit__(), не составляет труда. Однако в случае тривиальных контекстов написание полностью всего необходимого кода приводит к неоправданному увеличению накладных расходов. В ситуациях подобного рода наилучшим выходом является преобразование функции-генератора в менеджер контекста с помощью декоратора contextmanager().

Листинг 3.58. contextlib_contextmanager.py

```
import contextlib


@contextlib.contextmanager
def make_context():
    print(' entering')
    try:
        yield {}
    except RuntimeError as err:
        print(' ERROR:', err)
    finally:
        print(' exiting')

print('Normal:')
with make_context() as value:
```

```
    print(' inside with statement:', value)

print('\nHandled error:')
with make_context() as value:
    raise RuntimeError('showing example of handling an error')

print('\nUnhandled error:')
with make_context() as value:
    raise ValueError('this exception is not handled')
```

Генератор должен инициализировать контекст, использовать инструкцию `yield` ровно один раз, а затем очистить контекст. Выработанное значение, если таковое имеется, связывается с переменной, указанной в предложении `as` инструкции `with`. Возникшие в блоке `with` исключения повторно возбуждаются в генераторе, где их и можно обрабатывать.

```
$ python3 contextlib_contextmanager.py

Normal:
  entering
  inside with statement: {}
  exiting

Handled error:
  entering
  ERROR: showing example of handling an error
  exiting

Unhandled error:
  entering
  exiting
Traceback (most recent call last):
  File "contextlib_contextmanager.py", line 32, in <module>
    raise ValueError('this exception is not handled')
ValueError: this exception is not handled
```

Менеджер контекста, возвращаемый функцией `contextmanager()`, наследуется от класса `ContextDecorator`, и поэтому он работает также как декоратор функций.

Листинг 3.59. `contextlib_contextmanager_decorator.py`

```
import contextlib


@contextlib.contextmanager
def make_context():
    print(' entering')
    try:
        # Передается управление, а не значение, поскольку в случае
        # использования менеджера контекста в качестве декоратора
        # любое возвращенное значение остается недоступным
        yield
```

```
    except RuntimeError as err:
        print(' ERROR:', err)
    finally:
        print(' exiting')


@make_context()
def normal():
    print(' inside with statement')


@make_context()
def throw_error(err):
    raise err


print('Normal:')
normal()

print('\nHandled error:')
throw_error(RuntimeError('showing example of handling an error'))

print('\nUnhandled error:')
throw_error(ValueError('this exception is not handled'))
```

Как было показано в предыдущем примере с классом `ContextDecorator`, в случае использования менеджера контекста в качестве декоратора значение, выработанное генератором, остается недоступным в декорируемой функции. Аргументы, переданные декорированной функции, по-прежнему остаются доступными, что подтверждается следующим примером.

```
$ python3 contextlib_contextmanager_decorator.py

Normal:
  entering
  inside with statement
  exiting

Handled error:
  entering
  ERROR: showing example of handling an error
  exiting

Unhandled error:
  entering
  exiting
Traceback (most recent call last):
  File "contextlib_contextmanager_decorator.py", line 43, in
<module>
    throw_error(ValueError('this exception is not handled'))
  File ".../lib/python3.5/contextlib.py", line 30, in inner
    return func(*args, **kwds)
  File "contextlib_contextmanager_decorator.py", line 33, in
```

```
throw_error
    raise err
ValueError: this exception is not handled
```

3.4.4. Закрытие открытых дескрипторов

Класс `file` обеспечивает непосредственную поддержку API менеджера контекста, однако в случае других объектов, представляющих открытые дескрипторы, это не так. В разделе документации стандартной библиотеки, посвященном модулю `contextlib`, в качестве примера представлен объект, возвращаемый вызовом `urllib.urlopen()`. Некоторые другие устаревшие классы используют метод `close()`, но не поддерживают API менеджера контекста. Гарантированное закрытие дескриптора обеспечивается созданием для него менеджера контекста с помощью функции `closing()`.

Листинг 3.60. `contextlib_closing.py`

```
import contextlib


class Door:

    def __init__(self):
        print(' __init__()')
        self.status = 'open'

    def close(self):
        print(' close()')
        self.status = 'closed'

print('Normal Example:')
with contextlib.closing(Door()) as door:
    print(' inside with statement: {}'.format(door.status))
print(' outside with statement: {}'.format(door.status))

print('\nError handling example:')
try:
    with contextlib.closing(Door()) as door:
        print(' raising from inside with statement')
        raise RuntimeError('error message')
except Exception as err:
    print(' Had an error:', err)
```

Дескриптор закрывается независимо от того, возникла или не возникла ошибка в блоке `with`.

```
$ python3 contextlib_closing.py

Normal Example:
  __init__()
  inside with statement: open
  close()
```

```
  outside with statement: closed

Error handling example:
  __init__()
  raising from inside with statement
  close()
  Had an error: error message
```

3.4.5. Игнорирование исключений

Во многих случаях исключения, возбуждаемые библиотеками, удобно игнорировать, если ошибка указывает на достижение определенного состояния или может быть проигнорирована по другим причинам. Наиболее распространенным способом игнорирования исключений является использование инструкции `try:except`, содержащей в блоке `except` только инструкцию `pass`.

Листинг 3.61. contextlib_ignore_error.py

```
import contextlib


class NonFatalError(Exception):
    pass

def non_idempotent_operation():
    raise NonFatalError(
        'The operation failed because of existing state'
    )


try:
    print('trying non-idempotent operation')
    non_idempotent_operation()
    print('succeeded!')
except NonFatalError:
    pass

print('done')
```

В данном случае операцию выполнить не удалось, и ошибка была проигнорирована.

```
$ python3 contextlib_ignore_error.py

trying non-idempotent operation
done
```

Форму `try:except` можно заменить формой `contextlib.suppress()` для более явного подавления класса исключений, возникающих в пределах блока `with`.

Листинг 3.62. contextlib_suppress.py

```
import contextlib


class NonFatalError(Exception):
    pass


def non_idempotent_operation():
    raise NonFatalError(
        'The operation failed because of existing state'
    )


with contextlib.suppress(NonFatalError):
    print('trying non-idempotent operation')
    non_idempotent_operation()
    print('succeeded!')

print('done')
```

В этой обновленной версии исключение полностью игнорируется.

```
$ python3 contextlib_suppress.py

trying non-idempotent operation
done
```

3.4.6. Перенаправление выходных потоков

Плохо спроектированный библиотечный код может осуществлять запись данных непосредственно в поток `sys.stdout` или `sys.stderr`, не предоставляя аргументов, позволяющих конфигурировать другие варианты вывода. В подобных случаях, когда возможность доступа к исходному коду для внесения в него изменений, позволяющих управлять выводом с помощью аргумента, отсутствует, можно перехватывать вывод с помощью менеджеров контекста `redirect_stdout()` и `redirect_stderr()`.

Листинг 3.63. contextlib_redirect.py

```
from contextlib import redirect_stdout, redirect_stderr
import io
import sys


def misbehaving_function(a):
    sys.stdout.write('(stdout) A: {!r}\n'.format(a))
    sys.stderr.write('(stderr) A: {!r}\n'.format(a))


capture = io.StringIO()
with redirect_stdout(capture), redirect_stderr(capture):
```

```
    misbehaving_function(5)

print(capture.getvalue())
```

В этом примере функция `misbehaving_function()` осуществляет запись в оба потока вывода, `stdout` и `stderr`, но два контекстных менеджера перенаправляют вывод в один и тот же экземпляр `io.StringIO`, где он сохраняется для последующего использования.

```
$ python3 contextlib_redirect.py

(stdout) A: 5
(stderr) A: 5
```

Примечание

Оба менеджера контекста, `redirect_stdout()` и `redirect_stderr()`, изменяют глобальное состояние, подменяя объекты, содержащиеся в модуле `sys` (раздел 17.2); по этой причине их следует использовать с осторожностью. Фактически эти функции не являются потокобезопасными, поэтому их вызовы в многопоточном приложении могут приводить к неопределенным результатам. Кроме того, они могут взаимодействовать с другими операциями, которые рассчитывают на подключение стандартных потоков вывода к терминальным устройствам.

3.4.7. Стеки динамических менеджеров контекста

Большинство менеджеров контекста работает каждый раз с одним объектом, таким как дескриптор одиночного файла или базы данных. В подобных случаях объект известен заранее, и код, использующий менеджер контекста, может центрироваться вокруг этого объекта. Однако в других случаях программа может нуждаться в создании неизвестного количества объектов, для которых необходимо предусмотреть освобождение неиспользуемых ресурсов при покидании контекста. Именно для таких динамических случаев и создавался стек `ExitStack`.

Экземпляр `ExitStack` поддерживает структуру данных для хранения функций обратного вызова, обеспечивающих очистку ресурсов. Эти функции помещаются в стек явным образом внутри контекста и при покидании контекста потоком управления вызываются в обратном порядке. Результат выглядит так, как если бы использовались динамически устанавливаемые вложенные инструкции `with`.

3.4.7.1. Создание стека менеджеров контекста

Для заполнения экземпляра `ExitStack` существует несколько способов. В следующем примере для добавления нового менеджера контекста в стек используется метод `enter_context()`.

Листинг 3.64. `contextlib_exitstack_enter_context.py`

```
import contextlib


@contextlib.contextmanager
def make_context(i):
```

```
    print('{} entering'.format(i))
    yield {}
    print('{} exiting'.format(i))


def variable_stack(n, msg):
    with contextlib.ExitStack() as stack:
        for i in range(n):
            stack.enter_context(make_context(i))
        print(msg)


variable_stack(2, 'inside context')
```

Прежде всего метод `enter_context()` вызывает метод `__enter__()` для менеджера контекста. Затем он регистрирует его метод `__exit__()` в качестве функции обратного вызова.

```
$ python3 contextlib_exitstack_enter_context.py

0 entering
1 entering
inside context
1 exiting
0 exiting
```

Предоставленные экземпляру `ExitStack` менеджеры контекста обрабатываются так, как если бы они встречались в серии вложенных инструкций `with`. Ошибки, возникающие в пределах контекста, распространяются в соответствии с обычной процедурой обработки ошибок в менеджерах контекста.

Листинг 3.65. `contextlib_context_managers.py`

```
import contextlib


class Tracker:
    "Базовые класс для менеджеров контекста, генерирующих ошибки."

    def __init__(self, i):
        self.i = i

    def msg(self, s):
        print('  {}({}): {}'.format(
            self.__class__.__name__, self.i, s))

    def __enter__(self):
        self.msg('entering')


class HandleError(Tracker):
    "Если получено исключение, считать его обработанным."
```

```
    def __exit__(self, *exc_details):
        received_exc = exc_details[1] is not None
        if received_exc:
            self.msg('handling exception {!r}'.format(
                exc_details[1]))
        self.msg('exiting {}'.format(received_exc))
        # Возврат булевого значения, указывающего на то,
        # было ли обработано исключение
        return received_exc


class PassError(Tracker):
    "Если получено исключение, передать его дальше."

    def __exit__(self, *exc_details):
        received_exc = exc_details[1] is not None
        if received_exc:
            self.msg('passing exception {!r}'.format(
                exc_details[1]))
        self.msg('exiting')
        # Возврат значения False, указывающего на то, что
        # исключение не было обработано
        return False


class ErrorOnExit(Tracker):
    "Сгенерировать исключение."

    def __exit__(self, *exc_details):
        self.msg('throwing error')
        raise RuntimeError('from {}'.format(self.i))


class ErrorOnEnter(Tracker):
    "Сгенерировать исключение."

    def __enter__(self):
        self.msg('throwing error on enter')
        raise RuntimeError('from {}'.format(self.i))

    def __exit__(self, *exc_info):
        self.msg('exiting')
```

Применение этих классов проиллюстрировано ниже на примере функции `variable_stack()`, которая использует передаваемые ей менеджеры контекста для конструирования стека `ExitStack`, пошагово создавая общий контекст. Передача этой функции различных менеджеров контекста позволяет исследовать различные сценарии обработки ошибок. Первый пример представляет обычный случай, когда выполнение программы не сопровождается возбуждением исключений.

```
print('No errors:')
variable_stack([
    HandleError(1),
```

```
    PassError(2),
])
```

Следующий пример иллюстрирует обработку исключений, которые возникают в менеджерах контекста, находящихся в конце стека, когда при развертывании стека закрываются все открытые менеджеры контекста.

```
print('\nError at the end of the context stack:')
variable_stack([
    HandleError(1),
    HandleError(2),
    ErrorOnExit(3),
])
```

В следующем примере обрабатываются исключения, которые возникают в менеджерах контекста, находящихся в середине стека. Ошибка возникает тогда, когда некоторые контексты уже успели закрыться и поэтому не видят эту ошибку.

```
print('\nError in the middle of the context stack:')
variable_stack([
    HandleError(1),
    PassError(2),
    ErrorOnExit(3),
    HandleError(4),
])
```

Последний пример иллюстрирует случай, когда исключение остается необработанным и распространяется до вызывающего кода.

```
try:
    print('\nError ignored:')
    variable_stack([
        PassError(1),
        ErrorOnExit(2),
    ])
except RuntimeError:
    print('error handled outside of context')
```

Если какой-либо менеджер контекста в стеке получает исключение и возвращает значение `True`, то это предотвращает распространение исключения в другие менеджеры контекста.

```
$ python3 contextlib_exitstack_enter_context_errors.py

No errors:
  HandleError(1): entering
  PassError(2): entering
  PassError(2): exiting
  HandleError(1): exiting False
  outside of stack, any errors were handled

Error at the end of the context stack:
  HandleError(1): entering
```

```
  HandleError(2): entering
  ErrorOnExit(3): entering
  ErrorOnExit(3): throwing error
  HandleError(2): handling exception RuntimeError('from 3',)
  HandleError(2): exiting True
  HandleError(1): exiting False
  outside of stack, any errors were handled

Error in the middle of the context stack:
  HandleError(1): entering
  PassError(2): entering
  ErrorOnExit(3): entering
  HandleError(4): entering
  HandleError(4): exiting False
  ErrorOnExit(3): throwing error
  PassError(2): passing exception RuntimeError('from 3',)
  PassError(2): exiting
  HandleError(1): handling exception RuntimeError('from 3',)
  HandleError(1): exiting True
  outside of stack, any errors were handled

Error ignored:
  PassError(1): entering
  ErrorOnExit(2): entering
  ErrorOnExit(2): throwing error
  PassError(1): passing exception RuntimeError('from 2',)
  PassError(1): exiting
error handled outside of context
```

3.4.7.2. Использование произвольных функций обратного вызова в контекстах

Класс `ExitStack` также поддерживает использование произвольных функций обратного вызова для закрытия контекста, что упрощает выполнение завершающих операций по освобождению ресурсов, которые не контролируются менеджером контекста.

Листинг 3.66. `contextlib_exitstack_callbacks.py`

```
import contextlib


def callback(*args, **kwds):
    print('closing callback({}, {})'.format(args, kwds))


with contextlib.ExitStack() as stack:
    stack.callback(callback, 'arg1', 'arg2')
    stack.callback(callback, arg3='val3')
```

Как и в случае методов `__exit__()` полных менеджеров контекста, функции обратного вызова вызываются в обратном порядке по отношению к порядку их регистрации.

```
$ python3 contextlib_exitstack_callbacks.py

closing callback((), {'arg3': 'val3'})
closing callback(('arg1', 'arg2'), {})
```

Функции обратного вызова вызываются независимо от того, возникла ли ошибка, и не получают никакой информации об ошибке, если она возникает. Возвращаемое ими значение игнорируется.

Листинг 3.67. contextlib_exitstack_callbacks_error.py

```
import contextlib


def callback(*args, **kwds):
    print('closing callback({}, {})'.format(args, kwds))

try:
    with contextlib.ExitStack() as stack:
        stack.callback(callback, 'arg1', 'arg2')
        stack.callback(callback, arg3='val3')
        raise RuntimeError('thrown error')
except RuntimeError as err:
    print('ERROR: {}'.format(err))
```

Поскольку функции обратного вызова не имеют доступа к ошибке, они не могут предотвращать распространение исключений вдоль оставшейся части стека менеджеров контекста.

```
$ python3 contextlib_exitstack_callbacks_error.py

closing callback((), {'arg3': 'val3'})
closing callback(('arg1', 'arg2'), {})
ERROR: thrown error
```

Функции обратного вызова обеспечивают удобный способ четкого определения логики освобождения ресурсов без дополнительных накладных расходов, связанных с созданием нового класса менеджеров контекста. Для улучшения читаемости кода эту логику можно инкапсулировать во встраиваемую функцию с использованием функции `callback()` в качестве декоратора.

Листинг 3.68. contextlib_exitstack_callbacks_decorator.py

```
import contextlib


with contextlib.ExitStack() as stack:

    @stack.callback
    def inline_cleanup():
        print('inline_cleanup()')
        print('local_resource = {!r}'.format(local_resource))
```

```
    local_resource = 'resource created in context'
    print('within the context')
```

Не существует способов передачи аргументов функциям, зарегистрированным с помощью функции `callback()` в форме декоратора. Однако в случае использования встраиваемой функции правила областей видимости предоставляют ей доступ к переменным, определенным в вызывающем коде.

```
$ python3 contextlib_exitstack_callbacks_decorator.py

within the context
inline_cleanup()
local_resource = 'resource created in context'
```

3.4.7.3. Частичные стеки

Иногда при создании сложных контекстов полезно иметь возможность прекратить выполнение операции, если контекст не может быть создан, но при этом отложить выполнение завершающих операций по освобождению всех ресурсов до более позднего момента времени, если все они могут быть настроены надлежащим образом. Например, если для операции требуется несколько долгосрочных сетевых соединений, то, возможно, лучше всего вообще не начинать выполнение операции, если одно из соединений не удается установить. Однако, если могут быть открыты все соединения, то все они должны оставаться открытыми в течение более длительного периода времени, чем время жизни одного менеджера контекста. В подобных сценариях можно использовать метод `pop_all()` класса `ExitStack`.

Метод `pop_all()` удаляет из стека, для которого он вызван, все менеджеры контекста и функции обратного вызова и возвращает новый стек, заполненный этими же менеджерами контекста и функциями обратного вызова. Метод `close()` нового стека может быть вызван для освобождения ресурсов позже, когда будет удален исходный стек.

Листинг 3.69. `contextlib_exitstack_pop_all.py`

```
import contextlib

from contextlib_context_managers import *


def variable_stack(contexts):
    with contextlib.ExitStack() as stack:
        for c in contexts:
            stack.enter_context(c)
        # Вернуть метод close() нового стека в качестве
        # функции, освобождающей ресурсы
        return stack.pop_all().close
    # Явно вернуть значение None, указывающее на то, что
    # инициалиировать объект ExitStack невозможно, но операции
    # по освобождению ресурсов уже выполнены
    return None
```

```
print('No errors:')
cleaner = variable_stack([
    HandleError(1),
    HandleError(2),
])
cleaner()

print('\nHandled error building context manager stack:')
try:
    cleaner = variable_stack([
        HandleError(1),
        ErrorOnEnter(2),
    ])
except RuntimeError as err:
    print('caught error {}'.format(err))
else:
    if cleaner is not None:
        cleaner()
    else:
        print('no cleaner returned')

print('\nUnhandled error building context manager stack:')
try:
    cleaner = variable_stack([
        PassError(1),
        ErrorOnEnter(2),
    ])
except RuntimeError as err:
    print('caught error {}'.format(err))
else:
    if cleaner is not None:
        cleaner()
    else:
        print('no cleaner returned')
```

В этом примере используются те же классы менеджеров контекста, которые были определены ранее, но менеджер контекста `ErrorOnEnter` возбуждает исключение не в методе `__exit__()`, а в методе `__enter__()`. Если в теле функции `variable_stack()` все контексты создаются без возникновения ошибок, то она возвращает метод `close()` нового экземпляра `ExitStack`. Если возникает обрабатываемая ошибка, функция `variable_stack()` возвращает значение `None`, указывающее на то, что очистка ресурсов уже выполнена. Если же возникает необрабатываемая ошибка, то частичный стек очищается, а ошибка распространяется.

```
$ python3 contextlib_exitstack_pop_all.py

No errors:
  HandleError(1): entering
  HandleError(2): entering
  HandleError(2): exiting False
  HandleError(1): exiting False
```

```
Handled error building context manager stack:
  HandleError(1): entering
  ErrorOnEnter(2): throwing error on enter
  HandleError(1): handling exception RuntimeError('from 2',)
  HandleError(1): exiting True
no cleaner returned

Unhandled error building context manager stack:
  PassError(1): entering
  ErrorOnEnter(2): throwing error on enter
  PassError(1): passing exception RuntimeError('from 2',)
  PassError(1): exiting
caught error from 2
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `contextlib`[12].
- **PEP 343**[13]. Инструкция `with`.
- *Context Manager Types*[14]. Описание API менеджера контекста в документации стандартной библиотеки.
- *with Statement Context Managers*[15]. Описание API менеджера контекста в справочном руководстве Python.
- *Resource management in Python 3.3, or contextlib.ExitStack FTW!*[16] (Barry Warsaw). Примеры использования класса `ExitStack`.

[12] https://docs.python.org/3.5/library/contextlib.html
[13] www.python.org/dev/peps/pep-0343
[14] https://docs.python.org/library/stdtypes.html#typecontextmanager
[15] https://docs.python.org/reference/datamodel.html#context-managers
[16] www.wefearchange.org/2013/05/resource-management-in-python-33-or.html

Глава 4

Дата и время

В языке Python нет собственных типов значений даты и времени, как это, скажем, предусмотрено для целых чисел (`int`), чисел с плавающей точкой (`float`) или строк (`str`), но он содержит три модуля, позволяющие манипулировать значениями даты и времени в нескольких представлениях.

Модуль `time` (раздел 4.1) обеспечивает доступ к функциям, предназначенным для работы со временем, которые содержатся в базовой библиотеке C. Он предоставляет функции, предназначенные для извлечения значений системного и процессорного времени, а также базовые инструменты синтаксического анализа и форматирования строк.

Модуль `datetime` (раздел 4.2) предоставляет высокоуровневый интерфейс для работы с датами, временем и их комбинированными значениями. Содержащиеся в модуле `datetime` классы поддерживают арифметические операции, операции сравнения и настройку часовых поясов.

Модуль `calendar` (раздел 4.3) создает форматированное представление недель, месяцев и лет. Кроме того, его можно использовать для обработки повторяющихся событий и других календарных значений.

4.1. `time`: системное время

Модуль `time` обеспечивает доступ к нескольким типам часов, каждый из которых предназначен для использования в различных целях. Стандартные системные вызовы, такие как `time()`, возвращают системное время. Вызовы функции `monotonic()` можно использовать для измерения истекшего времени в случае длительно выполняющихся процессов, поскольку они гарантируют постоянное увеличение возвращаемых значений, даже если системное время было изменено. Для тестирования характеристик производительности следует использовать функцию `perf_counter()`, предоставляющую доступ к часам с наилучшей разрешающей способностью, что обеспечивает наибольшую точность измерения временных промежутков. Значения процессорного времени доступны через функцию `clock()`, тогда как функция `process_time()` возвращает комбинированное процессорное и системное время.

Примечание

Реализации Python обеспечивают доступ к функциям библиотеки C, позволяющим манипулировать значениями дат и времени. Поскольку они привязаны к лежащим в их основе реализациям C, некоторые детали (такие, как начало эпохи или максимальное поддерживаемое значение даты) оказываются платформозависимыми. Для получения более подробных сведений обратитесь к документации библиотеки.

4.1.1. Сравнительные характеристики часов

Детали реализации часов меняются от платформы к платформе. Для получения доступа к основной информации о текущей реализации, включая разрешающую способность часов, используйте функцию `get_clock_info()`.

Листинг 4.1. `time_get_clock_info.py`

```
import textwrap
import time

available_clocks = [
    ('clock', time.clock),
    ('monotonic', time.monotonic),
    ('perf_counter', time.perf_counter),
    ('process_time', time.process_time),
    ('time', time.time),
]

for clock_name, func in available_clocks:
    print(textwrap.dedent('''\
    {name}:
        adjustable    : {info.adjustable}
        implementation: {info.implementation}
        monotonic     : {info.monotonic}
        resolution    : {info.resolution}
        current       : {current}
    ''').format(
        name=clock_name,
        info=time.get_clock_info(clock_name),
        current=func())
    )
```

Как следует из приведенного ниже вывода результатов для Mac OS X, часы `monotonic` и `perf_counter` реализованы с использованием одного и того же базового системного вызова.

```
$ python3 time_get_clock_info.py

clock:
    adjustable    : False
    implementation: clock()
    monotonic     : True
    resolution    : 1e-06
    current       : 0.028399

monotonic:
    adjustable    : False
    implementation: mach_absolute_time()
    monotonic     : True
    resolution    : 1e-09
    current       : 172336.002232467

perf_counter:
```

```
    adjustable    : False
    implementation: mach_absolute_time()
    monotonic     : True
    resolution    : 1e-09
    current       : 172336.002280763

process_time:
    adjustable    : False
    implementation: getrusage(RUSAGE_SELF)
    monotonic     : True
    resolution    : 1e-06
    current       : 0.028593

time:
    adjustable    : True
    implementation: gettimeofday()
    monotonic     : False
    resolution    : 1e-06
    current       : 1471198232.045526
```

4.1.2. Часы текущего времени

Одной из основных функций модуля `time` является функция `time()`, которая возвращает количество секунд, истекших с момента начала "эпохи", в виде значения с плавающей точкой.

Листинг 4.2. `time_time.py`

```
import time

print('The time is:', time.time())
```

В качестве момента времени, соответствующего началу эпохи, от которого ведется отсчет времени, в системах Unix используется полночь 1 января 1970 года. Несмотря на то что значение времени всегда выражается числом с плавающей точкой, фактическая точность зависит от платформы.

```
$ python3 time_time.py

The time is: 1471198232.091589
```

Представление времени в виде значения с плавающей точкой весьма полезно при сохранении или сравнении дат, но неудобно для восприятия человеком. Для записи значений времени в журнал или вывода на консоль лучше использовать функцию `ctime()`.

Листинг 4.3. `time_ctime.py`

```
import time

print('The time is      :', time.ctime())
later = time.time() + 15
print('15 secs from now :', time.ctime(later))
```

В этом примере второй вызов функции демонстрирует, как использовать функцию `ctime()` для форматирования значения времени, отличного от текущего.

```
$ python3 time_ctime.py

The time is      : Sun Aug 14 14:10:32 2016
15 secs from now : Sun Aug 14 14:10:47 2016
```

4.1.3. Монотонные часы

Поскольку значения, возвращаемые функцией `time()`, основаны на показаниях системных часов, которые могут быть изменены пользователем или системными службами для синхронизации часов на нескольких компьютерах, результат каждого из повторных измерений, получаемых с ее помощью, может отличаться от предыдущего как в одну, так и в другую сторону. Это может приводить к неожиданному поведению результатов при попытках измерения длительностей временных промежутков или использования их для других целей. Этого можно избежать, используя функцию `monotonic()`, последовательные вызовы которой возвращают только возрастающие значения.

Листинг 4.4. `time_monotonic.py`

```
import time

start = time.monotonic()
time.sleep(0.1)
end = time.monotonic()
print('start : {:>9.2f}'.format(start))
print('end   : {:>9.2f}'.format(end))
print('span  : {:>9.2f}'.format(end - start))
```

Для монотонных часов начало отсчета не определено, поэтому возвращаемые ими значения полезны лишь для выполнения вычислений совместно со значениями других часов. В данном примере функция `monotonic()` используется для измерения длительности паузы.

```
$ python3 time_monotonic.py

start : 172336.14
end   : 172336.24
span  :      0.10
```

4.1.4. Процессорное время

В то время как функция `time()` возвращает значение текущего времени, функция `clock()` возвращает значения процессорного времени, которые отражают время, фактически затраченное процессором на обработку выполняющейся программы.

Листинг 4.5. time_clock.py

```
import hashlib
import time

# Данные, используемые для расчета контрольных сумм md5
data = open(__file__, 'rb').read()

for i in range(5):
    h = hashlib.sha1()
    print(time.ctime(), ': {:0.3f} {:0.3f}'.format(
        time.time(), time.clock()))
    for i in range(300000):
        h.update(data)
    cksum = h.digest()
```

В этом примере форматированные значения `ctime()` выводятся на каждой итерации цикла вместе со значениями с плавающей точкой, возвращаемыми функциями `time()` и `clock()`.

Примечание

Если захотите выполнить данный пример на своем компьютере, вам, возможно, придется увеличить количество итераций внутреннего цикла или использовать более крупный набор данных, чтобы заметить разницу в выводимых значениях.

```
$ python3 time_clock.py

Sun Aug 14 14:10:32 2016 : 1471198232.327 0.033
Sun Aug 14 14:10:32 2016 : 1471198232.705 0.409
Sun Aug 14 14:10:33 2016 : 1471198233.086 0.787
Sun Aug 14 14:10:33 2016 : 1471198233.466 1.166
Sun Aug 14 14:10:33 2016 : 1471198233.842 1.540
```

Как правило, в периоды бездействия программы отсчет процессорного времени не производится.

Листинг 4.6. time_clock_sleep.py

```
import time

template = '{} - {:0.2f} - {:0.2f}'

print(template.format(
    time.ctime(), time.time(), time.clock())
)

for i in range(3, 0, -1):
    print('Sleeping', i)
    time.sleep(i)
    print(template.format(
        time.ctime(), time.time(), time.clock())
    )
```

В этом примере цикл выполняет совсем небольшой объем работы, организуя короткие паузы после каждой итерации. Значение, возвращаемое функцией `time()`, увеличивается даже во время бездействия приложения, чего нельзя сказать о значениях, возвращаемых функцией `clock()`.

```
$ python3 -u time_clock_sleep.py

Sun Aug 14 14:10:34 2016 - 1471198234.28 - 0.03
Sleeping 3
Sun Aug 14 14:10:37 2016 - 1471198237.28 - 0.03
Sleeping 2
Sun Aug 14 14:10:39 2016 - 1471198239.29 - 0.03
Sleeping 1
Sun Aug 14 14:10:40 2016 - 1471198240.29 - 0.03
```

Вызов функции `sleep()` приостанавливает выполнение текущего потока и запрашивает для него ожидание до тех пор, пока он не будет пробужден системой. Если программа выполняется только в одном потоке, то вызов этой функции фактически блокирует приложение, так что в период паузы оно вообще не работает.

4.1.5. Измерение производительности

Для измерения характеристик производительности очень важно использовать монотонные часы с высокой разрешающей способностью. Определение наилучшего источника данных о времени требует учета особенностей платформы, что обеспечивается функцией `perf_counter()`.

Листинг 4.7. `time_perf_counter.py`

```
import hashlib
import time

# Данные, используемые для вычисления контрольной суммы md5
data = open(__file__, 'rb').read()

loop_start = time.perf_counter()

for i in range(5):
    iter_start = time.perf_counter()
    h = hashlib.sha1()
    for i in range(300000):
        h.update(data)
    cksum = h.digest()
    now = time.perf_counter()
    loop_elapsed = now - loop_start
    iter_elapsed = now - iter_start
    print(time.ctime(), ': {:0.3f} {:0.3f}'.format(
        iter_elapsed, loop_elapsed))
```

Как и в случае функции `monotonic()`, начало эпохи для функции `perf_counter()` не определено, в связи с чем предполагается, что ее результаты будут использоваться для сравнения и вычисления значений, а не в качестве абсолютных значений времени.

```
$ python3 time_perf_counter.py

Sun Aug 14 14:10:40 2016 : 0.487 0.487
Sun Aug 14 14:10:41 2016 : 0.485 0.973
Sun Aug 14 14:10:41 2016 : 0.494 1.466
Sun Aug 14 14:10:42 2016 : 0.487 1.953
Sun Aug 14 14:10:42 2016 : 0.480 2.434
```

4.1.6. Компоненты времени

Сохранение значений времени в виде количества истекших секунд полезно в некоторых ситуациях, но иногда программе может требоваться доступ к отдельным полям даты (например, год или месяц). Модуль `time` определяет класс `struct_time`, предназначенный для хранения значений даты и времени с разбивкой на отдельные компоненты для упрощения доступа к ним. Некоторые функции работают именно с этими значениями, а не со значениями в виде чисел с плавающей точкой.

Листинг 4.8. `time_struct.py`

```
import time


def show_struct(s):
    print('  tm_year :', s.tm_year)
    print('  tm_mon  :', s.tm_mon)
    print('  tm_mday :', s.tm_mday)
    print('  tm_hour :', s.tm_hour)
    print('  tm_min  :', s.tm_min)
    print('  tm_sec  :', s.tm_sec)
    print('  tm_wday :', s.tm_wday)
    print('  tm_yday :', s.tm_yday)
    print('  tm_isdst:', s.tm_isdst)


print('gmtime:')
show_struct(time.gmtime())
print('\nlocaltime:')
show_struct(time.localtime())
print('\nmktime:', time.mktime(time.localtime()))
```

Функция `gmtime()` возвращает текущее время в соответствии с UTC (всемирное координированное время). Функция `localtime()` возвращает текущее время, соответствующее установленному текущему часовому поясу. Функция `mktime()` получает в качестве аргумента объект `struct_time` и преобразует его в представление числа с плавающей точкой.

```
$ python3 time_struct.py

gmtime:
  tm_year : 2016
```

```
  tm_mon : 8
  tm_mday : 14
  tm_hour : 18
  tm_min : 10
  tm_sec : 42
  tm_wday : 6
  tm_yday : 227
  tm_isdst: 0

localtime:
  tm_year : 2016
  tm_mon : 8
  tm_mday : 14
  tm_hour : 14
  tm_min : 10
  tm_sec : 42
  tm_wday : 6
  tm_yday : 227
  tm_isdst: 1

mktime: 1471198242.0
```

4.1.7. Работа с часовыми поясами

Результаты работы функций, определяющие текущее время, зависят от того, какой часовой пояс установлен программой или задан по умолчанию для операционной системы. Изменение часового пояса изменяет не фактическое время, а лишь способ его представления.

Чтобы изменить часовой пояс, требуется установить значение переменной среды `TZ`, а затем вызвать функцию `tzset()`. При задании часового пояса можно указать множество деталей, вплоть до моментов начала и окончания летнего времени. Однако обычно проще использовать название часового пояса и позволить базовым библиотекам самостоятельно извлечь недостающую информацию.

В следующем примере устанавливаются несколько значений, определяющих часовой пояс, и демонстрируется, как эти изменения влияют на другие настройки модуля `time`.

Листинг 4.9. `time_timezone.py`

```
import time
import os


def show_zone_info():
    print('  TZ    :', os.environ.get('TZ', '(not set)'))
    print('  tzname:', time.tzname)
    print('  Zone  : {} ({})'.format(
        time.timezone, (time.timezone / 3600)))
    print('  DST   :', time.daylight)
    print('  Time  :', time.ctime())
    print()
```

```
print('Default :')
show_zone_info()

ZONES = [
    'GMT',
    'Europe/Amsterdam',
]

for zone in ZONES:
    os.environ['TZ'] = zone
    time.tzset()
    print(zone, ':')
    show_zone_info()
```

При подготовке примеров был использован часовой пояс U.S./Eastern, установленный в системе по умолчанию. Другие часовые пояса в примере изменяют значения атрибутов `tzname` и `timezone` и флага летнего времени DST.

```
$ python3 time_timezone.py

Default :
  TZ    : (not set)
  tzname: ('EST', 'EDT')
  Zone  : 18000 (5.0)
  DST   : 1
  Time  : Sun Aug 14 14:10:42 2016

GMT :
  TZ    : GMT
  tzname: ('GMT', 'GMT')
  Zone  : 0 (0.0)
  DST   : 0
  Time  : Sun Aug 14 18:10:42 2016

Europe/Amsterdam :
  TZ    : Europe/Amsterdam
  tzname: ('CET', 'CEST')
  Zone  : -3600 (-1.0)
  DST   : 1
  Time  : Sun Aug 14 20:10:42 2016
```

4.1.8. Разбор и форматирование значений времени

Функции `strptime()` и `strftime()` обеспечивают преобразование объекта `struct_time` в строковое представление значений времени и наоборот. Длинный список директив форматирования, поддерживаемых обеими функциями, обеспечивает различные стили представления входной и выходной информации. Их полный список можно найти в разделе документации библиотеки, посвященном модулю `time`.

Следующий пример иллюстрирует преобразование текущего времени из строкового представления в экземпляр `struct_time` и обратно в строку.

Листинг 4.10. `time_strptime.py`

```
import time


def show_struct(s):
    print('  tm_year :', s.tm_year)
    print('  tm_mon  :', s.tm_mon)
    print('  tm_mday :', s.tm_mday)
    print('  tm_hour :', s.tm_hour)
    print('  tm_min  :', s.tm_min)
    print('  tm_sec  :', s.tm_sec)
    print('  tm_wday :', s.tm_wday)
    print('  tm_yday :', s.tm_yday)
    print('  tm_isdst:', s.tm_isdst)


now = time.ctime(1483391847.433716)
print('Now:', now)

parsed = time.strptime(now)
print('\nParsed:')
show_struct(parsed)

print('\nFormatted:',
      time.strftime("%a %b %d %H:%M:%S %Y", parsed))
```

Выходная строка не повторяет в точности входную, отличаясь от нее добавленным ведущим нулем в номере месяца.

```
$ python3 time_strptime.py

Now: Mon Jan 2 16:17:27 2017

Parsed:
  tm_year : 2017
  tm_mon : 1
  tm_mday : 2
  tm_hour : 16
  tm_min : 17
  tm_sec : 27
  tm_wday : 0
  tm_yday : 2
  tm_isdst: -1

Formatted: Mon Jan 02 16:17:27 2017
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `time`[1].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `time` (раздел A.6.46).
- `datetime` (раздел 4.2). Модуль `datetime` включает другие классы, предназначенные для выполнения вычислений с использованием значений даты и времени.
- `calendar` (раздел 4.3). Предоставляет высокоуровневые функции для получения календарных значений и работы с повторяющимися событиями.

4.2. `datetime`: манипулирование значениями даты и времени

Модуль `datetime` содержит функции и классы, предназначенные для синтаксического анализа значений даты и времени, их форматирования и выполнения арифметических операций над ними.

4.2.1. Время

Для представления значений времени используется класс `time`. Экземпляр `time` имеет атрибуты `hour`, `minute`, `second` и `microsecond`; он также может включать информацию о часовом поясе.

Листинг 4.11. `datetime_time.py`

```
import datetime

t = datetime.time(1, 2, 3)
print(t)
print('hour       :', t.hour)
print('minute     :', t.minute)
print('second     :', t.second)
print('microsecond:', t.microsecond)
print('tzinfo     :', t.tzinfo)
```

Задавать аргументы, инициализирующие экземпляр `time`, необязательно, но, вероятно, используемые по умолчанию значения `0` не всегда будут корректными.

```
$ python3 datetime_time.py

01:02:03
hour       : 1
minute     : 2
second     : 3
microsecond: 0
tzinfo     : None
```

Экземпляр `time` хранит лишь значения времени и не содержит информацию о дате, связанной с данным значением времени.

[1] https://docs.python.org/3.5/library/time.html

Листинг 4.12. datetime_time_minmax.py

```
import datetime

print('Earliest  :', datetime.time.min)
print('Latest    :', datetime.time.max)
print('Resolution:', datetime.time.resolution)
```

Атрибуты класса `min` и `max` отражают диапазон допустимых значений времени суток.

```
$ python3 datetime_time_minmax.py

Earliest  : 00:00:00
Latest    : 23:59:59.999999
Resolution: 0:00:00.000001
```

Разрешающая способность экземпляров `time` ограничена целыми микросекундами.

Листинг 4.13. datetime_time_resolution.py

```
import datetime

for m in [1, 0, 0.1, 0.6]:
    try:
        print('{:02.1f} :'.format(m),
              datetime.time(0, 0, 0, microsecond=m))
    except TypeError as err:
        print('ERROR:', err)
```

Попытка задания для микросекунд значения в виде числа с плавающей точкой вызывает ошибку `TypeError`.

```
$ python3 datetime_time_resolution.py

1.0 : 00:00:00.000001
0.0 : 00:00:00
ERROR: integer argument expected, got float
ERROR: integer argument expected, got float
```

4.2.2. Даты

Для представления значений календарных дат используется класс `date`. Экземпляры этого класса имеют атрибуты, представляющие значения года, месяца и дня. Метод `today()` класса обеспечивает простой способ создания экземпляра, представляющего текущую дату.

Листинг 4.14. datetime_date.py

```
import datetime

today = datetime.date.today()
print(today)
```

```
print('ctime  :', today.ctime())
tt = today.timetuple()
print('tuple  : tm_year  =', tt.tm_year)
print('         tm_mon   =', tt.tm_mon)
print('         tm_mday  =', tt.tm_mday)
print('         tm_hour  =', tt.tm_hour)
print('         tm_min   =', tt.tm_min)
print('         tm_sec   =', tt.tm_sec)
print('         tm_wday  =', tt.tm_wday)
print('         tm_yday  =', tt.tm_yday)
print('         tm_isdst =', tt.tm_isdst)
print('ordinal:', today.toordinal())
print('Year   :', today.year)
print('Mon    :', today.month)
print('Day    :', today.day)
```

В этом примере текущее время выводится в нескольких форматах.

```
$ python3 datetime_date.py

2016-07-10
ctime  : Sun Jul 10 00:00:00 2016
tuple  : tm_year  = 2016
         tm_mon   = 7
         tm_mday  = 10
         tm_hour  = 0
         tm_min   = 0
         tm_sec   = 0
         tm_wday  = 6
         tm_yday  = 192
         tm_isdst = -1
ordinal: 736155
Year   : 2016
Mon    : 7
Day    : 10
```

Существуют также методы класса, предназначенные для создания экземпляров из временных меток POSIX или целых чисел, представляющих значения дат по григорианскому календарю, причем 1 января 1 года соответствует значению 1, которое увеличивается на 1 для каждых последующих суток.

Листинг 4.15. `datetime_date_fromordinal.py`

```
import datetime
import time

o = 733114
print('o               :', o)
print('fromordinal(o)  :', datetime.date.fromordinal(o))

t = time.time()
print('t               :', t)
print('fromtimestamp(t):', datetime.date.fromtimestamp(t))
```

Этот пример иллюстрирует различные типы значений, используемых функциями `fromordinal()` и `fromtimestamp()`.

```
$ python3 datetime_date_fromordinal.py

o               : 733114
fromordinal(o)  : 2008-03-13
t               : 1468161894.788508
fromtimestamp(t): 2016-07-10
```

Как и в случае класса `time`, диапазон допустимых значений даты можно определить с помощью атрибутов `min` и `max`.

Листинг 4.16. `datetime_date_minmax.py`

```
import datetime

print('Earliest  :', datetime.date.min)
print('Latest    :', datetime.date.max)
print('Resolution:', datetime.date.resolution)
```

Разрешающая способность для дат ограничена целыми сутками.

```
$ python3 datetime_date_minmax.py

Earliest  : 0001-01-01
Latest    : 9999-12-31
Resolution: 1 day, 0:00:00
```

Другим способом создания экземпляров даты является использование метода `replace()` существующего экземпляра `date`.

Листинг 4.17. `datetime_date_replace.py`

```
import datetime

d1 = datetime.date(2008, 3, 29)
print('d1:', d1.ctime())

d2 = d1.replace(year=2009)
print('d2:', d2.ctime())
```

В этом примере изменяется год, тогда как день и месяц не изменяются.

```
$ python3 datetime_date_replace.py

d1: Sat Mar 29 00:00:00 2008
d2: Sun Mar 29 00:00:00 2009
```

4.2.3. Промежутки времени

Будущие и прошедшие даты можно рассчитывать, используя базовые арифметические операции с участием двух объектов `datetime` или объекта `datetime`

и объекта timedelta, представляющего длительность временного промежутка. Вычитание дат дает длительность промежутка, а промежутки можно складывать и вычитать из дат для получения другой даты. Внутренние значения объектов timedelta хранятся в виде количества дней, секунд и микросекунд.

Листинг 4.18. datetime_timedelta.py

```
import datetime

print('microseconds:', datetime.timedelta(microseconds=1))
print('milliseconds:', datetime.timedelta(milliseconds=1))
print('seconds     :', datetime.timedelta(seconds=1))
print('minutes     :', datetime.timedelta(minutes=1))
print('hours       :', datetime.timedelta(hours=1))
print('days        :', datetime.timedelta(days=1))
print('weeks       :', datetime.timedelta(weeks=1))
```

Значения промежуточных уровней, передаваемые конструктору, преобразуется в количество дней, секунд и микросекунд.

```
$ python3 datetime_timedelta.py

microseconds: 0:00:00.000001
milliseconds: 0:00:00.001000
seconds     : 0:00:01
minutes     : 0:01:00
hours       : 1:00:00
days        : 1 day, 0:00:00
weeks       : 7 days, 0:00:00
```

Полную длительность временного промежутка, выраженную в секундах, можно получить с помощью функции total_seconds().

Листинг 4.19. datetime_timedelta_total_seconds.py

```
import datetime

for delta in [datetime.timedelta(microseconds=1),
              datetime.timedelta(milliseconds=1),
              datetime.timedelta(seconds=1),
              datetime.timedelta(minutes=1),
              datetime.timedelta(hours=1),
              datetime.timedelta(days=1),
              datetime.timedelta(weeks=1),
              ]:
    print('{:15} = {:8} seconds'.format(
        str(delta), delta.total_seconds())
    )
```

Чтобы обеспечить возможность работы с длительностями менее 1 секунды, функция возвращает значение в виде числа с плавающей точкой.

```
$ python3 datetime_timedelta_total_seconds.py

0:00:00.000001  =    1e-06 seconds
0:00:00.001000  =    0.001 seconds
0:00:01         =      1.0 seconds
0:01:00         =     60.0 seconds
1:00:00         =   3600.0 seconds
1 day, 0:00:00  =  86400.0 seconds
7 days, 0:00:00 = 604800.0 seconds
```

4.2.4. Арифметика дат

Выполнение операций над датами осуществляется с помощью стандартных арифметических операторов.

Листинг 4.20. datetime_date_math.py

```
import datetime

today = datetime.date.today()
print('Today    :', today)

one_day = datetime.timedelta(days=1)
print('One day  :', one_day)

yesterday = today - one_day
print('Yesterday:', yesterday)

tomorrow = today + one_day
print('Tomorrow :', tomorrow)

print()
print('tomorrow - yesterday:', tomorrow - yesterday)
print('yesterday - tomorrow:', yesterday - tomorrow)
```

В этом примере с объектами даты иллюстрируется использование объектов `timedelta` для вычисления новых дат. Кроме того, показано, как можно получить объект `timedelta` путем взаимного вычитания объектов даты (что может приводить к отрицательным значениям временны́х промежутков).

```
$ python3 datetime_date_math.py

Today : 2016-07-10
One day : 1 day, 0:00:00
Yesterday: 2016-07-09
Tomorrow : 2016-07-11

tomorrow - yesterday: 2 days, 0:00:00
yesterday - tomorrow: -2 days, 0:00:00
```

Объект `timedelta` также поддерживает арифметические операции с участием целых чисел, чисел с плавающей точкой и других экземпляров `timedelta`.

Листинг 4.21. `datetime_timedelta_math.py`

```
import datetime

one_day = datetime.timedelta(days=1)
print('1 day    :', one_day)
print('5 days   :', one_day * 5)
print('1.5 days :', one_day * 1.5)
print('1/4 day  :', one_day / 4)

# Отвести один час для ланча.
work_day = datetime.timedelta(hours=7)
meeting_length = datetime.timedelta(hours=1)
print('meetings per day :', work_day / meeting_length)
```

В этом примере вычисляются несколько различных длительностей, кратных целым суткам, с выражением результатов в виде количества дней или часов.

Последний пример демонстрирует вычисление значений с использованием двух объектов `timedelta`. В данном случае результат выражается числом с плавающей точкой.

```
$ python3 datetime_timedelta_math.py

1 day    : 1 day, 0:00:00
5 days   : 5 days, 0:00:00
1.5 days : 1 day, 12:00:00
1/4 day  : 6:00:00
meetings per day : 7.0
```

4.2.5. Сравнение значений

Значения как даты, так и времени можно сравнивать между собой, используя стандартные операторы сравнения для определения того, какое из них соответствует более раннему или более позднему моменту времени.

Листинг 4.22. `datetime_comparing.py`

```
import datetime
import time
print('Times:')
t1 = datetime.time(12, 55, 0)
print(' t1:', t1)
t2 = datetime.time(13, 5, 0)
print(' t2:', t2)
print(' t1 < t2:', t1 < t2)
print
print('Dates:')
d1 = datetime.date.today()
print(' d1:', d1)
d2 = datetime.date.today() + datetime.timedelta(days=1)
print(' d2:', d2)
print(' d1 > d2:', d1 > d2)
```

Поддерживаются все операторы сравнения.

```
$ python3 datetime_comparing.py

Times:
  t1: 12:55:00
  t2: 13:05:00
  t1 < t2: True
Dates:
  d1: 2016-07-10
  d2: 2016-07-11
  d1 > d2: False
```

4.2.6. Объединение значений даты и времени

Для хранения значений, включающих оба компонента, дату и время, используйте класс `datetime`. Как и в случае класса `date`, для создания экземпляров `datetime` из других распространенных значений предусмотрено несколько удобных методов класса.

Листинг 4.23. `datetime_datetime.py`

```
import datetime

print('Now    :', datetime.datetime.now())
print('Today  :', datetime.datetime.today())
print('UTC Now:', datetime.datetime.utcnow())
print

FIELDS = [
    'year', 'month', 'day',
    'hour', 'minute', 'second',
    'microsecond',
]

d = datetime.datetime.now()
for attr in FIELDS:
    print('{:15}: {}'.format(attr, getattr(d, attr)))
```

Как и можно было ожидать, экземпляр `datetime` имеет все атрибуты, свойственные как экземплярам `date`, так и экземплярам `time`.

```
$ python3 datetime_datetime.py

Now    : 2016-07-10 10:44:55.215677
Today  : 2016-07-10 10:44:55.215719
UTC Now: 2016-07-10 14:44:55.215732
year           : 2016
month          : 7
day            : 10
hour           : 10
minute         : 44
```

```
second         : 55
microsecond    : 216198
```

Как и класс date, класс datetime предоставляет удобные методы класса для создания новых экземпляров. В их число входят методы fromordinal() и fromtimestamp().

Листинг 4.24. datetime_datetime_combine.py

```
import datetime

t = datetime.time(1, 2, 3)
print('t :', t)

d = datetime.date.today()
print('d :', d)

dt = datetime.datetime.combine(d, t)
print('dt:', dt)
```

Метод combine() создает экземпляр datetime из одного экземпляра date и одного экземпляра time.

```
$ python3 datetime_datetime_combine.py

t : 01:02:03
d : 2016-07-10
dt: 2016-07-10 01:02:03
```

4.2.7. Форматирование и анализ значений

По умолчанию в качестве строкового представления объекта datetime используется формат ISO-8601 (ГГГГ-ММДДТЧЧ:ММ:СС.мммммм). Функция strftime() позволяет генерировать другие форматы.

Листинг 4.25. datetime_datetime_strptime.py

```
import datetime

format = "%a %b %d %H:%M:%S %Y"

today = datetime.datetime.today()
print('ISO :', today)

s = today.strftime(format)
print('strftime:', s)

d = datetime.datetime.strptime(s, format)
print('strptime:', d.strftime(format))
```

Для преобразования форматированных строк в экземпляры datetime используйте функцию datetime.strptime().

```
$ python3 datetime_datetime_strptime.py

ISO     : 2016-07-10 10:44:55.325247
strftime: Sun Jul 10 10:44:55 2016
strptime: Sun Jul 10 10:44:55 2016
```

Те же коды форматирования можно использовать в мини-языке форматирования строк Python[2], помещая их после двоеточия (:) в поле спецификации формата.

Листинг 4.26. datetime_format.py

```
import datetime

today = datetime.datetime.today()
print('ISO :', today)
print('format(): {:%a %b %d %H:%M:%S %Y}'.format(today))
```

Таблица 4.1. Коды формата функций strptime()/strftime()

Символ	Назначение	Пример
%a	Сокращенное название дня недели	'Wed'
%A	Полное название дня недели	'Wednesday'
%w	Десятичное представление дня недели: от 0 (воскресенье) до 6 (суббота)	'3'
%d	Десятичное представление числа месяца (дополнение ведущим нулем)	'13'
%b	Сокращенное название месяца	'Jan'
%B	Полное название месяца	'January'
%m	Десятичное представление месяца	'01'
%y	Десятичное представление года без указания века	'16'
%Y	Десятичное представление года с указанием века	'2016'
%H	Десятичное представление часа (24-часовая шкала)	'17'
%I	Десятичное представление часа (12-часовая шкала)	'05'
%p	AM/PM (до полудня/после полудня)	'PM'
%M	Десятичное представление минут	'00'
%S	Десятичное представление секунд	'00'
%f	Десятичное представление микросекунд	'000000'
%z	Смещение относительно времени UTC для объектов, получающих информацию о часовом поясе	'-0500'
%Z	Название часового пояса	'EST'
%j	Десятичное представление дня года	'013'
%W	Номер недели в году	'02'
%c	Дата и время в представлении текущей локали	'Wed Jan 13 17:00:00 2016'
%x	Дата в представлении текущей локали	'01/13/16'

[2] https://docs.python.org/3.5/library/string.html#formatspec

Окончание табл. 4.1

Символ	Назначение	Пример
%X	Время в представлении текущей локали	'17:00:00'
%%	Литеральный символ %	'%'

Каждый код формата представления даты и времени должен начинаться с префикса %. Все последующие двоеточия интерпретируются как литеральные символы и включаются в вывод.

```
$ python3 datetime_format.py

ISO : 2016-07-10 10:44:55.389239
format(): Sun Jul 10 10:44:55 2016
```

Представленные в табл. 4.1 коды форматирования соответствуют моменту времени 17:00 13 января 2016 года для часового пояса U.S./Eastern.

4.2.8. Часовые пояса

В классе datetime часовые пояса представлены подклассами tzinfo. Поскольку tzinfo — абстрактный базовый класс, приложение должно определить собственный подкласс и предоставить соответствующие реализации нескольких методов.

Модуль datetime включает тривиальную реализацию в классе timezone, в которой используется фиксированное смещение относительно UTC. Эта реализация не поддерживает различные значения смещения в различные дни года, что, например, требуется для учета перехода на летнее время или в тех случаях, когда смещение относительно UTC может меняться со временем.

Листинг 4.27. datetime_timezone.py

```
import datetime

min6 = datetime.timezone(datetime.timedelta(hours=-6))
plus6 = datetime.timezone(datetime.timedelta(hours=6))
d = datetime.datetime.now(min6)

print(min6, ':', d)
print(datetime.timezone.utc, ':', d.astimezone(datetime.timezone.utc))
print(plus6, ':', d.astimezone(plus6))

# Преобразовать в текущий системный часовой пояс
d_system = d.astimezone()
print(d_system.tzinfo, '      :', d_system)
```

Преобразование значения datetime одного часового пояса в другой осуществляется с помощью метода astimezone(). В предыдущем примере представлены два разных часовых пояса, смещенных на 6 часов по обе стороны от времени UTC, которое представлено атрибутом utc класса datetime.timezone. В последней строке вывода отображается значение системного часового пояса, полученное посредством вызова метода astimezone() без аргумента.

```
$ python3 datetime_timezone.py

UTC-06:00 : 2016-07-10 08:44:55.495995-06:00
UTC+00:00 : 2016-07-10 14:44:55.495995+00:00
UTC+06:00 : 2016-07-10 20:44:55.495995+06:00
EDT       : 2016-07-10 10:44:55.495995-04:00
```

Примечание

Сторонний модуль `pytz`[3] предлагает лучшую реализацию для работы с часовыми поясами. Он поддерживает именованные часовые пояса и обновляет базу данных смещений относительно UTC в случае принятия теми или иными государствами решений об их изменении.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `datetime`[4].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `datetime` (раздел A.6.13).
- `calendar` (раздел 4.3). Описание модуля `calendar`.
- `time` (раздел 4.1). Описание модуля `time`.
- `dateutil`[5]. Утилита `dateutil`, разработанная компанией Labix, расширяет модуль `datetime`, добавляя в него новые возможности.
- `pytz`[6]. База данных и классы для создания объектов `datetime`, имеющих доступ к актуальной информации о часовых поясах.
- Википедия: *Пролептический григорианский календарь*[7]. Описание григорианской календарной системы.
- Википедия: *ISO 8601*[8]. Стандарт, определяющий правила числового представления значений даты и времени.

4.3. `calendar`: работа с датами

Модуль `calendar` определяет класс `Calendar`, инкапсулирующий вычисление таких значений, как даты дней недели в заданном месяце или году. В дополнение к этому классы `TextCalendar` и `HTMLCalendar` позволяют получать предварительно отформатированный вывод.

4.3.1. Примеры форматирования

Метод `prmonth()` — это простая функция, обеспечивающая форматированный текстовый вывод для указанного месяца.

[3] `http://pytz.sourceforge.net/`
[4] `https://docs.python.org/3.5/library/datetime.html`
[5] `http://labix.org/python-dateutil`
[6] `http://pytz.sourceforge.net/`
[7] `https://ru.wikipedia.org/wiki/Пролептический_григорианский_календарь`
[8] `https://ru.wikipedia.org/wiki/ISO_8601`

Листинг 4.28. calendar_textcalendar.py

```
import calendar

c = calendar.TextCalendar(calendar.SUNDAY)
c.prmonth(2017, 7)
```

В следующем примере для класса TextCalendar в соответствии с принятым в США соглашением в качестве начального дня недели установлено воскресенье (Sunday). По умолчанию используется принятое в Европе соглашение, в соответствии с которым началом недели считается понедельник. Ниже приведен результат выполнения этого примера.

```
$ python3 calendar_textcalendar.py

     July 2017
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Аналогичную HTML-таблицу можно получить с помощью класса HTMLCalendar и метода formatmonth(). В этом случае визуализируемый вывод выглядит в основном так же, как и его текстовая версия, но обернут тегами HTML. Каждая ячейка таблицы имеет атрибут класса, соответствующий дню недели, поэтому внешним видом HTML-версии можно управлять, используя стили CSS.

Чтобы вывести результаты в формате, отличающемся от принятого по умолчанию, используйте класс calendar для вычисления дат и сгруппируйте значения по неделям и месяцам, а затем выполните итерации по полученному результату. Для этих целей особенно удобно использовать методы weekheader(), monthcalendar() и yeardays2calendar() класса Calendar.

Вызов метода yeardays2calendar() создает последовательность списков "строк месяцев". Каждый список месяцев содержит каждый месяц в виде списка недель. Недели представляются списками кортежей, включающих числа месяца (1–31) и номера дней недели (0–6). Дни, выходящие за пределы месяца, имеют нулевой номер.

Листинг 4.29. calendar_yeardays2calendar.py

```
import calendar
import pprint

cal = calendar.Calendar(calendar.SUNDAY)

cal_data = cal.yeardays2calendar(2017, 3)
print('len(cal_data)      :', len(cal_data))
```

```
top_months = cal_data[0]
print('len(top_months)    :', len(top_months))

first_month = top_months[0]
print('len(first_month)   :', len(first_month))

print('first_month:')
pprint.pprint(first_month, width=65)
```

Вызов метода `yeardays2calendar(2017, 3)` возвращает даты для 2017 года, сгруппированные по три месяца в строке.

```
$ python3 calendar_yeardays2calendar.py

len(cal_data) : 4
len(top_months) : 3
len(first_month) : 5
first_month:
[[(1, 6), (2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5)],
 [(8, 6), (9, 0), (10, 1), (11, 2), (12, 3), (13, 4), (14, 5)],
 [(15, 6), (16, 0), (17, 1), (18, 2), (19, 3), (20, 4), (21,
5)],
 [(22, 6), (23, 0), (24, 1), (25, 2), (26, 3), (27, 4), (28,
5)],
 [(29, 6), (30, 0), (31, 1), (0, 2), (0, 3), (0, 4), (0, 5)]]
```

Ниже приведен эквивалент этих дат, используемый методом `formatyear()`.

Листинг 4.30. `calendar_formatyear.py`

```
import calendar

cal = calendar.TextCalendar(calendar.SUNDAY)
print(cal.formatyear(2017, 2, 1, 1, 3))
```

При указанных аргументах метод `formatyear()` выводит следующий результат.

```
$ python3 calendar_formatyear.py

                                  2017

      January                   February                  March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7               1  2  3  4            1  2  3  4
 8  9 10 11 12 13 14   5  6  7  8  9 10 11   5  6  7  8  9 10 11
15 16 17 18 19 20 21  12 13 14 15 16 17 18  12 13 14 15 16 17 18
22 23 24 25 26 27 28  19 20 21 22 23 24 25  19 20 21 22 23 24 25
29 30 31              26 27 28              26 27 28 29 30 31

       April                     May                      June
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
                   1      1  2  3  4  5  6                  1  2  3
 2  3  4  5  6  7  8   7  8  9 10 11 12 13   4  5  6  7  8  9 10
 9 10 11 12 13 14 15  14 15 16 17 18 19 20  11 12 13 14 15 16 17
```

```
16 17 18 19 20 21 22  21 22 23 24 25 26 27  18 19 20 21 22 23 24
23 24 25 26 27 28 29  28 29 30 31            25 26 27 28 29 30
30

        July                    August                  September
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
                   1            1  2  3  4  5                  1  2
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   3  4  5  6  7  8  9
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  10 11 12 13 14 15 16
16 17 18 19 20 21 22  20 21 22 23 24 25 26  17 18 19 20 21 22 23
23 24 25 26 27 28 29  27 28 29 30 31        24 25 26 27 28 29 30
30 31

       October                 November                December
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7               1  2  3  4                  1  2
 8  9 10 11 12 13 14   5  6  7  8  9 10 11   3  4  5  6  7  8  9
15 16 17 18 19 20 21  12 13 14 15 16 17 18  10 11 12 13 14 15 16
22 23 24 25 26 27 28  19 20 21 22 23 24 25  17 18 19 20 21 22 23
29 30 31              26 27 28 29 30        24 25 26 27 28 29 30
                                            31
```

Атрибуты модуля `day_name`, `day_abbr`, `month_name` и `month_abbr` полезны для получения вывода в пользовательском формате (например, включающем ссылки в HTML-вывод). Эти атрибуты автоматически конфигурируются подходящим образом для текущей локали.

4.3.2. Локали

Для получения календаря, отформатированного в соответствии с локалью, отличной от локали по умолчанию, используйте класс `LocaleTextCalendar` или `LocaleHTMLCalendar`.

Листинг 4.31. `calendar_locale.py`

```
import calendar

c = calendar.LocaleTextCalendar(locale='en_US')
c.prmonth(2017, 7)

print()

c = calendar.LocaleTextCalendar(locale='fr_FR')
c.prmonth(2017, 7)
```

Первый день недели не является частью настроек локали. Его значение берется из аргумента, переданного классу `calendar`, как это происходит и в случае обычного класса `TextCalendar`.

```
$ python3 calendar_locale.py

     July 2017
Mo Tu We Th Fr Sa Su
                1  2
```

```
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

    juillet 2017
Lu Ma Me Je Ve Sa Di
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

4.3.3. Вычисление дат

Несмотря на то что модуль `calendar` предназначен главным образом для вывода полных календарей в различных форматах, он также предоставляет функции, которые могут быть полезными при других способах использования дат, скажем, для вычисления дат повторяющихся событий. Например, группа пользователей Python в Атланте проводит собрания каждый второй четверг каждого месяца. Для получения графика собраний на протяжении года можно использовать значение, возвращаемое функцией `monthcalendar()`.

Листинг 4.32. `calendar_monthcalendar.py`

```
import calendar

import pprint
pprint.pprint(calendar.monthcalendar(2017, 7))
```

Некоторые дни имеют номер `0`. Это те дни недели, которые перекрываются с заданным месяцем, но относятся к другому месяцу.

```
$ python3 calendar_monthcalendar.py

[[0, 0, 0, 0, 0, 1, 2],
 [3, 4, 5, 6, 7, 8, 9],
 [10, 11, 12, 13, 14, 15, 16],
 [17, 18, 19, 20, 21, 22, 23],
 [24, 25, 26, 27, 28, 29, 30],
 [31, 0, 0, 0, 0, 0, 0]]
```

По умолчанию первым днем недели считается понедельник. Это значение можно изменить, вызвав функцию `setfirstweekday()`. В данном случае еще более удобный подход заключается в том, чтобы опустить этот шаг, поскольку модуль `calendar` включает константы для индексирования диапазонов дат, возвращаемых функцией `monthcalendar()`.

Чтобы вычислить график собраний группы на протяжении года в предположении, что они созываются каждый второй четверг каждого месяца, следует

определить даты, на которые приходятся четверги, найдя их среди результатов, возвращаемых функцией `monthcalendar()`. Первая и последняя недели месяца заполняются нулевыми значениями, относящими эти недели соответственно к предыдущему и последующему месяцам. Например, если первым днем месяца является пятница, то значением для первой недели в позиции четверга (Thursday) будет 0.

Листинг 4.33. `calendar_secondthursday.py`

```
import calendar
import sys

year = int(sys.argv[1])

# Отобразить каждый месяц
for month in range(1, 13):

    # Вычислить даты для каждой недели, перекрывающейся
    # с данным месяцем
    c = calendar.monthcalendar(year, month)
    first_week = c[0]
    second_week = c[1]
    third_week = c[2]

    # Если первая неделя содержит четверг,
    # то второй четверг приходится на вторую неделю.
    # В противном случае второй четверг должен
    # приходиться на третью неделю.
    if first_week[calendar.THURSDAY]:
        meeting_date = second_week[calendar.THURSDAY]
    else:
        meeting_date = third_week[calendar.THURSDAY]

    print('{:>3}: {:>2}'.format(calendar.month_abbr[month],
                                meeting_date))
```

Таким образом, расписание встреч для заданного года имеет следующий вид.

```
$ python3 calendar_secondthursday.py 2017

Jan: 12
Feb:  9
Mar:  9
Apr: 13
May: 11
Jun:  8
Jul: 13
Aug: 10
Sep: 14
Oct: 12
Nov:  9
Dec: 14
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `calendar`[9].
- `time` (раздел 4.1). Низкоуровневые функции для работы со значениями времени.
- `datetime` (раздел 4.2). Манипулирование значениями даты, включая временные метки и часовые пояса.
- `locale` (раздел А.15.2). Настройки локали.

[9] https://docs.python.org/3.5/library/calendar.html

Глава 5

Математика

Python — универсальный язык программирования, и поэтому его часто используют для решения математических задач. Он включает встроенные типы для работы с целыми числами и числами с плавающей точкой, пригодные для выполнения базовых математических вычислений, с которыми приходится сталкиваться в типичных приложениях. Стандартная библиотека включает модули, предназначенные для проведения более сложных математических расчетов.

В основе встроенного типа чисел с плавающей точкой лежит представление чисел двойной точности (`double`). Для большинства программ этой точности вполне достаточно, но в тех случаях, когда к точности нецелочисленных значений предъявляются повышенные требования, могут оказаться полезными модуль `decimal`, позволяющий работать с дробными десятичными числами фиксированной точности (раздел 5.1), и модуль `fractions`, позволяющий работать с рациональными числами (раздел 5.2). Арифметические операции над числами этого типа обеспечивают сохранение необходимой точности и выполняются медленнее, чем операции над числами с плавающей точкой.

Модуль `random` (раздел 5.3) включает генератор равномерно распределенных псевдослучайных чисел, а также функции, имитирующие многие неравномерные распределения.

Модуль `math` (раздел 5.4) содержит высокопроизводительные реализации более сложных математических функций, таких как логарифмы и тригонометрические функции. В этом модуле доступен полный комплект функций, которые обычно входят в состав библиотек C, удовлетворяющих требованиям стандарта IEEE.

5.1. `decimal`: математика чисел с фиксированной точностью и чисел с плавающей точкой

Модуль `decimal` реализует арифметику чисел с фиксированной точностью и чисел с плавающей точкой с использованием модели, знакомой большинству людей, а не с использованием версии чисел с плавающей точкой, определенной стандартом IEEE, которая реализована на большинстве аппаратных платформ и знакома программистам. Экземпляр `Decimal` обеспечивает точное представление любого числа, с округлением как вниз, так и вверх, используя заданное предельное значение количества значащих цифр.

5.1.1. Класс `Decimal`

Дробные десятичные числа представляются экземплярами класса `Decimal`. Его конструктору передается в качестве аргумента одно целое число или строка. Числа с плавающей точкой могут предварительно преобразовываться в строку, прежде чем будут использованы для создания экземпляра `Decimal`, тем самым

предоставляя вызывающему коду возможность явно устанавливать количество значащих цифр в значениях, которые не могут быть точно выражены с использованием аппаратного представления чисел с плавающей точкой.

Другой возможный вариант заключается в использовании метода `from_float()` для преобразования числа с плавающей точкой в его точное десятичное представление.

Листинг 5.1. `decimal_create.py`

```
import decimal

fmt = '{0:<25} {1:<25}'

print(fmt.format('Input', 'Output'))
print(fmt.format('-' * 25, '-' * 25))

# Целое число
print(fmt.format(5, decimal.Decimal(5)))

# Строка
print(fmt.format('3.14', decimal.Decimal('3.14')))

# Число с плавающей точкой
f = 0.1
print(fmt.format(repr(f), decimal.Decimal(str(f))))
print('{:<0.23g} {:<25}'.format(
    f,
    str(decimal.Decimal.from_float(f))[:25])
)
```

Значение с плавающей точкой `0.1` невозможно точно представить в двоичной форме, поэтому его представление в виде числа с плавающей точкой отличается от дробного десятичного значения. В последней строке выведенных результатов полное строковое представление усечено до 25 символов.

```
Input                     Output
------------------------- -------------------------
5                         5
3.14                      3.14
0.1                       0.1
0.10000000000000000555112 0.1000000000000000055511
```

Дробные десятичные числа также можно создавать на основе кортежей, содержащих флаг знака (`0` для положительных чисел и `1` для отрицательных), кортеж цифр и целочисленную экспоненту.

Листинг 5.2. `decimal_tuple.py`

```
import decimal

# Кортеж
t = (1, (1, 1), -2)
print('Input  :', t)
print('Decimal:', decimal.Decimal(t))
```

Представление чисел, основанное на кортежах, менее удобно для создания экземпляров Decimal, но обеспечивает портируемость дробных десятичных значений при их экспорте без потери точности. Число в форме кортежа может быть передано по сети или сохранено в базе данных, которая не поддерживает точные десятичные значения, а затем преобразовано обратно в экземпляр Decimal.

```
$ python3 decimal_tuple.py

Input  : (1, (1, 1), -2)
Decimal: -0.11
```

5.1.2. Форматирование

Класс Decimal соответствует протоколу форматирования строк Python[1] и использует те же синтаксис и опции, что и другие числовые типы.

Листинг 5.3. decimal_format.py

```
import decimal

d = decimal.Decimal(1.1)
print('Precision:')
print('{:.1}'.format(d))
print('{:.2}'.format(d))
print('{:.3}'.format(d))
print('{:.18}'.format(d))

print('\nWidth and precision combined:')
print('{:5.1f} {:5.1g}'.format(d, d))
print('{:5.2f} {:5.2g}'.format(d, d))
print('{:5.2f} {:5.2g}'.format(d, d))

print('\nZero padding:')
print('{:05.1}'.format(d))
print('{:05.2}'.format(d))
print('{:05.3}'.format(d))
```

Строки формата могут управлять шириной вывода, точностью (т.е. количеством значащих цифр) и средствами выравнивания значений по заданной ширине поля.

```
$ python3 decimal_format.py

Precision:
1
1.1
1.10
1.10000000000000009
```

[1] https://docs.python.org/3.5/library/string.html#formatspec

```
Width and precision combined:
  1.1     1
 1.10   1.1
 1.10   1.1

Zero padding:
00001
001.1
01.10
```

5.1.3. Арифметика

Класс Decimal перегружает простые арифметические операторы, поэтому его экземплярами можно манипулировать в основном так же, как и встроенными числовыми типами.

Листинг 5.4. decimal_operators.py

```
import decimal

a = decimal.Decimal('5.1')
b = decimal.Decimal('3.14')
c = 4
d = 3.14

print('a =', repr(a))
print('b =', repr(b))
print('c =', repr(c))
print('d =', repr(d))
print()

print('a + b =', a + b)
print('a - b =', a - b)
print('a * b =', a * b)
print('a / b =', a / b)
print()

print('a + c =', a + c)
print('a - c =', a - c)
print('a * c =', a * c)
print('a / c =', a / c)
print()

print('a + d =', end=' ')
try:
    print(a + d)
except TypeError as e:
    print(e)
```

Операторы класса Decimal также поддерживают целочисленные аргументы. В противоположность этому значения с плавающей точкой должны преобразовываться в экземпляры Decimal, прежде чем их можно будет использовать совместно с этими операторами.

```
$ python3 decimal_operators.py

a = Decimal('5.1')
b = Decimal('3.14')
c = 4
d = 3.14

a + b = 8.24
a - b = 1.96
a * b = 16.014
a / b = 1.624203821656050955414012739

a + c = 9.1
a - c = 1.1
a * c = 20.4
a / c = 1.275

a + d = unsupported operand type(s) for +: 'decimal.Decimal' and
 'float'
```

Помимо базовых арифметических операторов класс `Decimal` включает методы для вычисления десятичных и натуральных логарифмов. Возвращаемые методами `log10()` и `ln()` значения являются экземплярами `Decimal`, которые можно непосредственно использовать в формулах наряду с другими значениями.

5.1.4. Специальные значения

Кроме ожидаемых числовых значений, класс `Decimal` может представлять несколько специальных значений, в том числе положительную и отрицательную бесконечности (`+Infinity` и `-Infinity`), "не число" (`NaN`) и нуль.

Листинг 5.5. `decimal_special.py`

```
import decimal

for value in ['Infinity', 'NaN', '0']:
    print(decimal.Decimal(value), decimal.Decimal('-' + value))
print()

# Операции с Infinity
print('Infinity + 1:', (decimal.Decimal('Infinity') + 1))
print('-Infinity + 1:', (decimal.Decimal('-Infinity') + 1))

# Вывод результатов сравнения с NaN
print(decimal.Decimal('NaN') == decimal.Decimal('Infinity'))
print(decimal.Decimal('NaN') != decimal.Decimal(1))
```

Результатом сложения любого числа с `Infinity` (`-Infinity`) является `Infinity` (`-Infinity`). Сравнение на равенство со значением `NaN` всегда возвращает значение `False`, тогда как сравнение на неравенство — значение `True`. Сравнение со значением `NaN` в целях сортировки не определено и приводит к ошибке.

```
$ python3 decimal_special.py

Infinity -Infinity
NaN -NaN
0 -0
Infinity + 1: Infinity
-Infinity + 1: -Infinity
False
True
```

5.1.5. Контекст

До сих пор во всех примерах использовалось поведение модуля decimal, заданное по умолчанию. Существует возможность изменить это поведение, используя *контекст* для настройки таких параметров, как поддерживаемая точность, способ округления и обработка ошибок. Контексты могут применяться ко всем экземплярам Decimal как в потоке, так и локально, в пределах небольшого участка кода.

5.1.5.1. Текущий контекст

Для извлечения текущего глобального контекста используется функция getcontext().

Листинг 5.6. decimal_getcontext.py

```
import decimal

context = decimal.getcontext()

print('Emax     =', context.Emax)
print('Emin     =', context.Emin)
print('capitals =', context.capitals)
print('prec =', context.prec)
print('rounding =', context.rounding)
print('flags    =')
for f, v in context.flags.items():
    print(' {}: {}'.format(f, v))
print('traps    =')
for t, v in context.traps.items():
    print(' {}: {}'.format(t, v))
```

В этом примере сценарий отображает общедоступные свойства объекта Context.

```
$ python3 decimal_getcontext.py

Emax     = 999999
Emin     = -999999
capitals = 1
prec     = 28
rounding = ROUND_HALF_EVEN
```

```
flags    =
  <class 'decimal.InvalidOperation'>: False
  <class 'decimal.FloatOperation'>: False
  <class 'decimal.DivisionByZero'>: False
  <class 'decimal.Overflow'>: False
  <class 'decimal.Underflow'>: False
  <class 'decimal.Subnormal'>: False
  <class 'decimal.Inexact'>: False
  <class 'decimal.Rounded'>: False
  <class 'decimal.Clamped'>: False
traps    =
  <class 'decimal.InvalidOperation'>: True
  <class 'decimal.FloatOperation'>: False
  <class 'decimal.DivisionByZero'>: True
  <class 'decimal.Overflow'>: True
  <class 'decimal.Underflow'>: False
  <class 'decimal.Subnormal'>: False
  <class 'decimal.Inexact'>: False
  <class 'decimal.Rounded'>: False
  <class 'decimal.Clamped'>: False
```

5.1.5.2. Точность

Атрибут `prec` контекста управляет точностью, поддерживаемой для новых значений, которые создаются в результате выполнения математических операций. Литеральные значения поддерживаются в соответствии с приведенным описанием.

Листинг 5.7. `decimal_precision.py`

```
import decimal

d = decimal.Decimal('0.123456')

for i in range(1, 5):
    decimal.getcontext().prec = i
    print(i, ':', d, d * 1)
```

Точность можно изменить, присвоив новое значение из диапазона от 1 до `decimal.MAX_PREC` непосредственно данному атрибуту.

```
$ python3 decimal_precision.py

1 : 0.123456 0.1
2 : 0.123456 0.12
3 : 0.123456 0.123
4 : 0.123456 0.1235
```

5.1.5.3. Округление

Существует несколько опций, управляющих округлением значений в пределах заданной точности.

- ROUND_CEILING. Округление в направлении Infinity.
- ROUND_DOWN. Округление в направлении нуля.
- ROUND_FLOOR. Округление в направлении -Infinity.
- ROUND_HALF_DOWN. Округление в направлении от нуля, если последняя цифра равна или больше 5; в противном случае — округление в направлении нуля.
- ROUND_HALF_EVEN. Аналогична ROUND_HALF_DOWN, за исключением того, что в случаях, когда последняя цифра равна 5, проверяется предпоследняя цифра. Четные цифры приводят к округлению результата вниз, нечетные — к округлению вверх.
- ROUND_HALF_UP. Аналогична ROUND_HALF_DOWN, за исключением того, что в тех случаях, когда последняя цифра равна 5, значение округляется в направлении от нуля.
- ROUND_UP. Округление в направлении от нуля.
- ROUND_05UP. Округление в направлении от нуля, если последняя цифра равна 0 или 5; в противном случае — округление в направлении нуля.

Листинг 5.8. decimal_rounding.py

```
import decimal

context = decimal.getcontext()

ROUNDING_MODES = [
    'ROUND_CEILING',
    'ROUND_DOWN',
    'ROUND_FLOOR',
    'ROUND_HALF_DOWN',
    'ROUND_HALF_EVEN',
    'ROUND_HALF_UP',
    'ROUND_UP',
    'ROUND_05UP',
]

header_fmt = '{:10} ' + ' '.join(['{:^8}'] * 6)

print(header_fmt.format(
    ' ',
    '1/8 (1)', '-1/8 (1)',
    '1/8 (2)', '-1/8 (2)',
    '1/8 (3)', '-1/8 (3)',
))
for rounding_mode in ROUNDING_MODES:
    print('{0:10}'.format(rounding_mode.partition('_')[-1]),
          end=' ')
    for precision in [1, 2, 3]:
        context.prec = precision
        context.rounding = getattr(decimal, rounding_mode)
        value = decimal.Decimal(1) / decimal.Decimal(8)
        print('{0:^8}'.format(value), end=' ')
```

```
        value = decimal.Decimal(-1) / decimal.Decimal(8)
        print('{0:^8}'.format(value), end=' ')
    print()
```

Эта программа отображает результаты округления одного и того же значения до различных уровней точности с использованием разных алгоритмов.

```
$ python3 decimal_rounding.py

           1/8 (1)  -1/8 (1) 1/8 (2)  -1/8 (2) 1/8 (3)  -1/8 (3)
CEILING      0.2      -0.1     0.13     -0.12    0.125    -0.125
DOWN         0.1      -0.1     0.12     -0.12    0.125    -0.125
FLOOR        0.1      -0.2     0.12     -0.13    0.125    -0.125
HALF_DOWN    0.1      -0.1     0.12     -0.12    0.125    -0.125
HALF_EVEN    0.1      -0.1     0.12     -0.12    0.125    -0.125
HALF_UP      0.1      -0.1     0.13     -0.13    0.125    -0.125
UP           0.2      -0.2     0.13     -0.13    0.125    -0.125
05UP         0.1      -0.1     0.12     -0.12    0.125    -0.125
```

5.1.5.4. Локальный контекст

Контекст может применяться к блоку кода с использованием инструкции `with`.

Листинг 5.9. `decimal_context_manager.py`

```
import decimal

with decimal.localcontext() as c:
    c.prec = 2
    print('Local precision:', c.prec)
    print('3.14 / 3 =', (decimal.Decimal('3.14') / 3))

print()
print('Default precision:', decimal.getcontext().prec)
print('3.14 / 3 =', (decimal.Decimal('3.14') / 3))
```

Контекст поддерживает API менеджеров контекста, используемых с инструкцией `with`, поэтому соответствующие настройки применяются только в пределах блока.

```
$ python3 decimal_context_manager.py

Local precision: 2
3.14 / 3 = 1.0

Default precision: 28
3.14 / 3 = 1.046666666666666666666666667
```

5.1.5.5. Контекст экземпляра

Контексты также можно использовать для создания экземпляров `Decimal`, которые затем наследуют аргументы точности и округления, взятые из контекста.

Листинг 5.10. decimal_instance_context.py

```
import decimal

# Установка контекста с ограниченной точностью
c = decimal.getcontext().copy()
c.prec = 3

# Создание собственной константы
pi = c.create_decimal('3.1415')

# Округление постоянного значения
print('PI :', pi)

# Результат использования константы с учетом
# глобального контекста
print('RESULT:', decimal.Decimal('2.01') * pi)
```

Этот подход позволяет приложению выбирать точность постоянных значений независимо, например, от точности пользовательских данных.

```
$ python3 decimal_instance_context.py

PI : 3.14
RESULT: 6.3114
```

5.1.5.6. Потоки

"Глобальный" контекст в действительности является локальным контекстом потока, поэтому каждый поток может конфигурироваться с использованием различных значений.

Листинг 5.11. decimal_thread_context.py

```
import decimal
import threading
from queue import PriorityQueue


class Multiplier(threading.Thread):
    def __init__(self, a, b, prec, q):
        self.a = a
        self.b = b
        self.prec = prec
        self.q = q
        threading.Thread.__init__(self)

    def run(self):
        c = decimal.getcontext().copy()
        c.prec = self.prec
        decimal.setcontext(c)
        self.q.put((self.prec, a * b))


a = decimal.Decimal('3.14')
```

```
b = decimal.Decimal('1.234')
# Объект PriorityQueue вернет значения, сортированные по
# точности, независимо от очередности завершения потоков
q = PriorityQueue()
threads = [Multiplier(a, b, i, q) for i in range(1, 6)]
for t in threads:
    t.start()

for t in threads:
    t.join()

for i in range(5):
    prec, value = q.get()
    print('{}  {}'.format(prec, value))
```

Этот пример создает новый контекст с использованием заданных значений, а затем устанавливает его в каждом потоке.

```
$ python3 decimal_thread_context.py

1  4
2  3.9
3  3.87
4  3.875
5  3.8748
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `decimal`[2].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `decimal` (раздел A.6.14).
- Википедия: *Число с плавающей запятой*[3]. Статья, касающаяся представления и арифметики чисел с плавающей точкой (запятой).
- *Floating Point Arithmetic. Issues and Limitations*[4]. Статья из руководства Python, в которой описываются проблемы, связанные с выполнением математических операций над числами с плавающей точкой.

5.2. `fractions`: рациональные числа

Класс `Fraction` реализует арифметические операции над рациональными числами, опираясь на API, определяемый классом `Rational` в модуле `numbers`.

5.2.1. Создание экземпляров `Fraction`

Как и в случае модуля `decimal` (раздел 3.1), новые значения можно создавать несколькими способами. Один из простых способов сделать это заключается в использовании отдельных значений числителя и знаменателя.

[2] https://docs.python.org/3.5/library/decimal.html

[3] https://ru.wikipedia.org/wiki/Число_с_плавающей_запятой

[4] https://docs.python.org/tutorial/floatingpoint.html

Листинг 5.12. fractions_create_integers.py

```
import fractions

for n, d in [(1, 2), (2, 4), (3, 6)]:
    f = fractions.Fraction(n, d)
    print('{}/{} = {}'.format(n, d, f))
```

При вычислении новых значений поддерживается наименьший знаменатель дроби.

```
$ python3 fractions_create_integers.py

1/2 = 1/2
2/4 = 1/2
3/6 = 1/2
```

Другой способ создания объектов Fraction заключается в использовании строкового представления *<числитель>/<знаменатель>*.

Листинг 5.13. fractions_create_strings.py

```
import fractions

for s in ['1/2', '2/4', '3/6']:
    f = fractions.Fraction(s)
    print('{} = {}'.format(s, f))
```

Значения числителя и знаменателя определяются в результате анализа строки.

```
$ python3 fractions_create_strings.py

1/2 = 1/2
2/4 = 1/2
3/6 = 1/2
```

Кроме того, в строках можно использовать более привычную запись десятичных чисел и чисел с плавающей точкой в виде последовательности цифр, содержащей точку. Поддерживаются любые строки, которые могут быть преобразованы в число с помощью функции float() и не представляют значение NaN или бесконечность.

Листинг 5.14. fractions_create_strings_floats.py

```
import fractions

for s in ['0.5', '1.5', '2.0', '5e-1']:
    f = fractions.Fraction(s)
    print('{0:>4} = {1}'.format(s, f))
```

Значения числителя и знаменателя, представленные числами с плавающей точкой, обрабатываются автоматически.

```
$ python3 fractions_create_strings_floats.py

 0.5 = 1/2
 1.5 = 3/2
 2.0 = 2
5e-1 = 1/2
```

Также возможно создание экземпляров Fraction непосредственно из других представлений рациональных чисел, таких как float или Decimal.

Листинг 5.15. fractions_from_float.py

```
import fractions

for v in [0.1, 0.5, 1.5, 2.0]:
    print('{} = {}'.format(v, fractions.Fraction(v)))
```

Значения с плавающей точкой, которые не могут быть выражены точно, могут приводить к неожиданным результатам.

```
$ python3 fractions_from_float.py

0.1 = 3602879701896397/36028797018963968
0.5 = 1/2
1.5 = 3/2
2.0 = 2
```

Использование значений в представлении Decimal приводит к ожидаемым результатам.

Листинг 5.16. fractions_from_decimal.py

```
import decimal
import fractions

values = [
    decimal.Decimal('0.1'),
    decimal.Decimal('0.5'),
    decimal.Decimal('1.5'),
    decimal.Decimal('2.0'),
]

for v in values:
    print('{} = {}'.format(v, fractions.Fraction(v)))
```

Внутренняя реализация типа Decimal не страдает недостатками стандартного представления чисел с плавающей точкой, связанными с потерей точности.

```
$ python3 fractions_from_decimal.py

0.1 = 1/10
0.5 = 1/2
1.5 = 3/2
2.0 = 2
```

5.2.2. Арифметика

Как только получены экземпляры рациональных чисел, их можно использовать в математических выражениях.

Листинг 5.17. `fractions_arithmetic.py`

```
import fractions

f1 = fractions.Fraction(1, 2)
f2 = fractions.Fraction(3, 4)

print('{} + {} = {}'.format(f1, f2, f1 + f2))
print('{} - {} = {}'.format(f1, f2, f1 - f2))
print('{} * {} = {}'.format(f1, f2, f1 * f2))
print('{} / {} = {}'.format(f1, f2, f1 / f2))
```

Поддерживаются все стандартные операторы.

```
$ python3 fractions_arithmetic.py

1/2 + 3/4 = 5/4
1/2 - 3/4 = -1/4
1/2 * 3/4 = 3/8
1/2 / 3/4 = 2/3
```

5.2.3. Аппроксимация значений

Полезным свойством экземпляров `Fraction` является возможность преобразования чисел с плавающей точкой в приближенную рациональную дробь.

Листинг 5.18. `fractions_limit_denominator.py`

```
import fractions
import math

print('PI       =', math.pi)

f_pi = fractions.Fraction(str(math.pi))
print('No limit =', f_pi)

for i in [1, 6, 11, 60, 70, 90, 100]:
    limited = f_pi.limit_denominator(i)
    print('{0:8} = {1}'.format(i, limited))
```

Значением дроби можно управлять, ограничивая величину знаменателя.

```
$ python3 fractions_limit_denominator.py

PI       = 3.141592653589793
No limit = 3141592653589793/1000000000000000
       1 = 3
       6 = 19/6
      11 = 22/7
```

```
 60 = 179/57
 70 = 201/64
 90 = 267/85
100 = 311/99
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `fractions`[5].
- `decimal` (раздел 5.1). Модуль `decimal` предоставляет API для выполнения математических операций над числами с фиксированной и плавающей точкой.
- `numbers`. Числовые абстрактные базовые классы.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `fractions` (раздел А.6.15).

5.3. `random`: генератор псевдослучайных чисел

Модуль `random` предоставляет быстрый генератор псевдослучайных чисел, основанный на алгоритме под названием *вихрь Мерсенна*. Этот алгоритм, первоначально разработанный для расчетов методом Монте-Карло, генерирует числа, подчиняющиеся равномерному распределению с большим периодом повторения, что делает его весьма удобным для широкого ряда приложений.

5.3.1. Генерация случайных чисел

Функция `random()` возвращает следующее случайное число с плавающей точкой из генерируемой последовательности. Все генерируемые значения принадлежат диапазону чисел `0 <= n < 1.0`.

Листинг 5.19. `random_random.py`

```
import random

for i in range(5):
    print('%04.3f' % random.random(), end=' ')
print()
```

Повторные запуски программы приводят к различным последовательностям чисел.

```
$ python3 random_random.py

0.859 0.297 0.554 0.985 0.452

$ python3 random_random.py

0.797 0.658 0.170 0.297 0.593
```

Функция `uniform()` обеспечивает генерацию чисел в пределах заданного числового диапазона.

[5] https://docs.python.org/3.5/library/fractions.html

Листинг 5.20. random_uniform.py

```
import random

for i in range(5):
    print('{:04.3f}'.format(random.uniform(1, 100)), end=' ')
print()
```

Функция uniform() получает минимальное и максимальное значения диапазона и корректирует возвращаемые функцией random() значения с помощью формулы min + (max - min) * random().

```
$ python3 random_uniform.py

12.428 93.766 95.359 39.649 88.983
```

5.3.2. Инициализация

Функция random() генерирует при каждом запуске различные значения, характеризующиеся большим периодом повторения чисел. Это полезно для получения уникальных значений или варьируемого ряда значений, но иногда желательно иметь один и тот же набор данных, подвергаемый обработке различными способами. Вообще говоря, программа может сгенерировать последовательность случайных чисел и сохранить ее для последующей обработки в качестве отдельного шага. Однако в случае больших наборов данных такой подход не является оптимальным, поэтому модуль random включает функцию seed(), позволяющую инициализировать генератор псевдослучайных чисел затравочным значением таким образом, чтобы он создавал один и тот же набор ожидаемых значений при каждом запуске.

Листинг 5.21. random_seed.py

```
import random

random.seed(1)

for i in range(5):
    print('{:04.3f}'.format(random.random()), end=' ')
print()
```

Затравочное значение управляет первым значением, получаемым с помощью формулы, которая используется для генерации псевдослучайных чисел. Учитывая детерминистический характер этой формулы, затравочное значение определяет последовательность генерируемых чисел в целом. Аргументом функции seed() может служить любой хешируемый объект.

По умолчанию используется платформозависимый фактор случайности, если это возможно. В противном случае в качестве затравочного значения используется системное время.

```
$ python3 random_seed.py

0.134 0.847 0.764 0.255 0.495
```

```
$ python3 random_seed.py

0.134 0.847 0.764 0.255 0.495
```

5.3.3. Сохранение состояния

Внутреннее состояние алгоритма генерации псевдослучайных чисел, используемого функцией `random()`, можно сохранить и использовать для управления генерацией чисел при последующих запусках. Восстановление предыдущего состояния, выступающего в качестве начальной точки для продолжения процесса, снижает вероятность повторения ранее полученных значений или последовательностей значений. Функция `getstate()` возвращает данные, которые впоследствии могут быть использованы для повторной инициализации генератора случайных чисел с помощью функции `setstate()`.

Листинг 5.22. `random_state.py`

```
import random
import os
import pickle

import random
import os
import pickle

if os.path.exists('state.dat'):
    # Восстановление ранее сохраненного состояния
    print('Found state.dat, initializing random module')
    with open('state.dat', 'rb') as f:
        state = pickle.load(f)
    random.setstate(state)
else:
    # Использование известного начального состояния
    print('No state.dat, seeding')
    random.seed(1)

# Создание случайных значений
for i in range(3):
    print('{:04.3f}'.format(random.random()), end=' ')
print()

# Сохранение состояния для следующего запуска
with open('state.dat', 'wb') as f:
    pickle.dump(random.getstate(), f)

# Получение дополнительных случайных значений
print('\nAfter saving state:')
for i in range(3):
    print('{:04.3f}'.format(random.random()), end=' ')
print()
```

Характер данных, возвращаемых функцией `getstate()`, зависит от реализации, поэтому в данном примере данные сохраняются в файле с помощью модуля

pickle (раздел 7.1); во всем остальном генератор псевдослучайных чисел рассматривается как черный ящик. Если во время запуска программы обнаруживается, что файл существует, она загружает прежнее состояние и продолжает вычисления. С целью демонстрации того, что восстановление состояния приводит к повторной генерации тех же значений, программа генерирует при каждом запуске несколько значений как до, так и после сохранения состояния.

```
$ python3 random_state.py

No state.dat, seeding
0.134 0.847 0.764

After saving state:
0.255 0.495 0.449

$ python3 random_state.py

Found state.dat, initializing random module
0.255 0.495 0.449

After saving state:
0.652 0.789 0.094
```

5.3.4. Случайные целые числа

Функция random() генерирует числа с плавающей точкой. Эти числа можно преобразовать в целые, однако удобнее использовать функцию randint(),непосредственно генерирующую целые числа.

Листинг 5.23. random_randint.py

```
import random

print('[1, 100]:', end=' ')

for i in range(3):
    print(random.randint(1, 100), end=' ')

print('\n[-5, 5]:', end=' ')
for i in range(3):
    print(random.randint(-5, 5), end=' ')
print()
```

Аргументами функции randint() являются границы интервала значений. Ими могут быть положительные и отрицательные числа, но первое значение должно быть меньше второго.

```
$ python3 random_randint.py

[1, 100]: 98 75 34
[-5, 5]: 4 0 5
```

Функция randrange() обеспечивает более общую форму выбора значений из диапазона.

Листинг 5.24. random_randrange.py

```
import random

for i in range(3):
    print(random.randrange(0, 101, 5), end=' ')
print()
```

В дополнение к значениям start и stop функция randrange() поддерживает аргумент step, поэтому она полностью эквивалентна выбору случайного значения из диапазона range(start,stop,step). В то же время она более эффективна, поскольку диапазон фактически не конструируется.

```
$ python3 random_randrange.py

15 20 85
```

5.3.5. Выбор случайных элементов

Генераторы случайных чисел часто применяются для выбора случайного элемента из последовательности перечислимых значений, даже если эти значения не являются числами. Модуль random включает функцию choice(), обеспечивающую случайный выбор элемента последовательности. Следующий пример имитирует подбрасывание монеты 10000 раз и подсчет количества выпавших "орлов" или "решек".

Листинг 5.25. random_choice.py

```
import random
import itertools

outcomes = {
    'heads': 0,
    'tails': 0,
}
sides = list(outcomes.keys())

for i in range(10000):
    outcomes[random.choice(sides)] += 1

print('Heads:', outcomes['heads'])
print('Tails:', outcomes['tails'])
```

В данном случае существуют лишь два возможных исхода. Таким образом, вместо того чтобы использовать числа и преобразовывать их, в примере применяется функция choice(), которой в качестве аргумента передается словарь с ключами "heads" ("орлы") и "tails" ("решки"). Результаты сохраняются в этом же словаре в виде его значений.

```
$ python3 random_choice.py

Heads: 5091
Tails: 4909
```

5.3.6. Перестановки

Имитация карточной игры предполагает перемешивание колоды карт и раздачу их игрокам, причем ни одна карта не должна встречаться более одного раза. Использование функции `choice()` может привести к тому, что одна и та же карта встретится более одного раза. В подобных случаях для перемешивания колоды карт можно использовать функцию `shuffle()`, а затем удалять карты из полученного набора по мере их "раздачи" игрокам.

Листинг 5.26. `random_shuffle.py`

```
import random
import itertools

import random
import itertools

FACE_CARDS = ('J', 'Q', 'K', 'A')
SUITS = ('H', 'D', 'C', 'S')


def new_deck():
    return [
        # Всегда использовать для значения две позиции, чтобы
        # обеспечить одинаковую ширину строк
        '{:>2}{}'.format(*c)
        for c in itertools.product(
            itertools.chain(range(2, 11), FACE_CARDS),
            SUITS,
        )
    ]


def show_deck(deck):
    p_deck = deck[:]
    while p_deck:
        row = p_deck[:13]
        p_deck = p_deck[13:]
        for j in row:
            print(j, end=' ')
        print()


# Создать новую колоду с упорядоченным расположением карт
deck = new_deck()
print('Initial deck:')
show_deck(deck)
```

```
# Перетасовать колоду с помощью функции shuffle()
random.shuffle(deck)
print('\nShuffled deck:')
show_deck(deck)

# Раздать карты на 4 руки по 5 карт на каждую руку
hands = [[], [], [], []]

for i in range(5):
    for h in hands:
        h.append(deck.pop())

# Показать раздачу
print('\nHands:')
for n, h in enumerate(hands):
    print('{}:'.format(n + 1), end=' ')
    for c in h:
        print(c, end=' ')
    print()

# Показать карты в оставшейся части колоды
print('\nRemaining deck:')
show_deck(deck)
```

Каждая карта представлена строкой, указывающей значимость и масть карты. Раздача карт имитируется добавлением по одной карте в каждый из четырех списков, представляющих "руки", с одновременным удалением этой карты из колоды, чтобы предотвратить ее повторную раздачу.

```
$ python3 random_shuffle.py

Initial deck:
 2H  2D  2C  2S  3H  3D  3C  3S  4H  4D  4C  4S  5H
 5D  5C  5S  6H  6D  6C  6S  7H  7D  7C  7S  8H  8D
 8C  8S  9H  9D  9C  9S 10H 10D 10C 10S  JH  JD  JC
 JS  QH  QD  QC  QS  KH  KD  KC  KS  AH  AD  AC  AS

Shuffled deck:
 7H  4D  AD  QC  AS  2H  4H  4C  6S  2D  KC  9S  JH
 6H  9D  6D  3H  3S 10S  QH  8S  5H  7D  KD  QS  8H
 9C 10C 10H  7S  4S  JS  2C  8C  3C  5S 10D  KS  JD
 5D  5C  QD  KH  JC  6C  2S  7C  3D  9H  8D  AC  AH

Hands:
1:  AH  3D  JC  5D  5S
2:  AC  7C  KH  JD  3C
3:  8D  2S  QD  KS  8C
4:  9H  6C  5C 10D  2C

Remaining deck:
 7H  4D  AD  QC  AS  2H  4H  4C  6S  2D  KC  9S  JH
 6H  9D  6D  3H  3S 10S  QH  8S  5H  7D  KD  QS  8H
 9C 10C 10H  7S  4S  JS
```

5.3.7. Выборки

Во многих имитациях требуется извлекать выборки из популяции (генеральной совокупности) входных значений. Функция `sample()` генерирует выборки, не содержащие повторяющихся значений, без изменения входной последовательности. В следующем примере выводится случайная выборка слов, извлекаемых из системного словаря.

Листинг 5.27. `random_sample.py`

```
import random

with open('/usr/share/dict/words', 'rt') as f:
    words = f.readlines()
words = [w.rstrip() for w in words]

for w in random.sample(words, 5):
    print(w)
```

Учет размеров входной последовательности и требуемой выборки алгоритмом, используемым для создания результирующего набора, обеспечивает максимально возможную эффективность получаемых результатов.

```
$ python3 random_sample.py

streamlet
impestation
violaquercitrin
mycetoid
plethoretical

$ python3 random_sample.py

nonseditious
empyemic
ultrasonic
Kyurinish
amphide
```

5.3.8. Одновременное использование нескольких генераторов

В дополнение к функциям уровня модуля модуль `random` предлагает класс `Random`, обеспечивающий управление внутренним состоянием генераторов чисел. Все ранее описанные функции доступны в виде методов экземпляров `Random`, и каждый экземпляр может инициализироваться и использоваться по отдельности, не взаимодействуя со значениями, возвращенными другими экземплярами.

Листинг 5.28. `random_random_class.py`

```
import random
import time
```

```
print('Default initializiation:\n')

r1 = random.Random()
r2 = random.Random()

for i in range(3):
    print('{:04.3f} {:04.3f}'.format(r1.random(), r2.random()))

print('\nSame seed:\n')

seed = time.time()
r1 = random.Random(seed)
r2 = random.Random(seed)

for i in range(3):
    print('{:04.3f} {:04.3f}'.format(r1.random(), r2.random()))
```

В случае систем с хорошими возможностями предоставления затравочных значений, базирующихся на аппаратных особенностях платформы, каждый экземпляр запускается в собственном уникальном состоянии. В отсутствие надежного аппаратного источника случайных значений для этой цели используется текущее системное время, что, вероятнее всего, приведет к тому, что все генераторы будут генерировать одни и те же значения.

```
$ python3 random_random_class.py

Default initializiation:

0.862 0.390
0.833 0.624
0.252 0.080

Same seed:

0.466 0.466
0.682 0.682
0.407 0.407
```

5.3.9. Класс `SystemRandom`

Некоторые операционные системы предоставляют генератор случайных чисел, имеющий доступ к дополнительным источникам энтропии, которые могут вводиться в генератор. Модуль `random` предоставляет такую возможность посредством класса `SystemRandom`. Этот класс имеет тот же API, что и класс `Random`, но использует функцию `os.urandom()` для генерации значений, которые служат базисом для всех других алгоритмов.

Листинг 5.29. `random_system_random.py`

```
import random
import time
```

```
print('Default initializiation:\n')

r1 = random.SystemRandom()
r2 = random.SystemRandom()

for i in range(3):
    print('{:04.3f} {:04.3f}'.format(r1.random(), r2.random()))

print('\nSame seed:\n')

seed = time.time()
r1 = random.SystemRandom(seed)
r2 = random.SystemRandom(seed)

for i in range(3):
    print('{:04.3f} {:04.3f}'.format(r1.random(), r2.random()))
```

Последовательности, получаемые с помощью класса `SystemRandom`, не являются воспроизводимыми, поскольку фактор случайности порождается системой, а не состоянием программного обеспечения. (Фактически в данном случае от функций `seed()` и `setstate()` вообще ничего не зависит.)

```
$ python3 random_system_random.py

Default initializiation:
0.110 0.481
0.624 0.350
0.378 0.056

Same seed:
0.634 0.731
0.893 0.843
0.065 0.177
```

5.3.10. Неоднородные распределения

В то время как однородное распределение значений, получаемое с помощью функции `random()`, полезно для многих целей, существуют другие распределения, которые обеспечивают более точное моделирование тех или иных ситуаций. Модуль `random` включает ряд функций для генерации значений, подчиняющихся таким законам распределения. Ниже эти функции лишь перечислены и не рассматриваются более подробно, поскольку они имеют специальные области применения и для их иллюстрации требуются более сложные примеры.

5.3.10.1. Нормальное распределение

Нормальное распределение широко применяется для исследования закономерностей поведения таких неоднородно распределенных непрерывных значений, как рост или вес. Кривая этого распределения имеет характерную форму, которая и обусловила ее распространенное название: *колоколообразная кривая*. Модуль `random` включает две функции, которые генерируют значения, подчиняющиеся

нормальному распределению: `normalvariate()` и работающая несколько быстрее `gauss()` (нормальное распределение также называют *гауссовым*).

Родственная им функция `lognormvariate()` позволяет получать псевдослучайные значения, логарифмы которых распределены по нормальному закону. Такие *логарифмически нормальные* (*логнормальные*) распределения удобны для исследования поведения чисел, представляющих собой произведения нескольких случайных переменных, которые не взаимодействуют между собой.

5.3.10.2. Аппроксимирующее распределение

Треугольное распределение, генерируемое функцией `triangular()`, используется в качестве приближенного распределения при небольших размерах выборок. "Кривая" плотности треугольного распределения определяется двумя точками основания треугольника, соответствующими известным минимальному и максимальному значениям (аргументы `low` и `high` функции `triangular()`), и точкой вершины, соответствующей *моде* – "наиболее вероятному" значению (аргумент `mode`).

5.3.10.3. Экспоненциальное распределение

Функция `expovariate()` генерирует *экспоненциальное* (*показательное*) распределение, используемое для имитации совершения событий или интервалов времени между совершениями событий в однородных процессах Пуассона, таких как процесс радиоактивного распада или поток запросов, поступающих на сервер.

Закон Парето, согласующийся со многими наблюдаемыми явлениями, был популяризирован Крисом Андерсоном в статье *The Long Tail* ("Длинный хвост технологий"). Функция `paretovariate()` полезна для имитации неравномерности распределения причин и следствий (например, распределение доходов среди населения, спроса на музыкантов, посещаемости блогов и т.п.).

5.3.10.4. Угловое распределение

Распределение фон Мизеса, или *круговое нормальное распределение*, генерируемое функцией `vonmisesvariate()`, используется для вычисления вероятностей циклических значений, таких как углы, календарные дни и время.

5.3.10.5. Размерные распределения

Функция `betavariate()` генерирует значения, соответствующие бета-распределению, которое обычно используется в байесовской статистике и таких приложениях, как моделирование длительности выполнения задач.

Гамма-распределение, генерируемое функцией `gammavariate()`, используется для моделирования размера различных величин, таких как время ожидания, количество осадков или количество вычислительных ошибок.

Распределение Вейбулла, генерируемое с помощью функции `weibullvariate()`, используется при анализе отказов, в промышленной инженерии и при составлении прогнозов погоды. Оно описывает распределение размеров частиц или других дискретных объектов.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `random`[6].
- *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator* (M. Matsumoto, T. Nishimura). Статья в журнале *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, January 1998, pp. 3–30.
- Википедия: *Вихрь Мерсенна*[7]. Описание алгоритма генератора псевдослучайных чисел, используемого в Python.
- Википедия: *Непрерывное равномерное распределение*[8]. Статья о применении непрерывного равномерного распределения в статистике.

5.4. `math`: математические функции

Модуль `math` реализует многие из специальных функций, специфицированных стандартом IEEE, которые обычно содержатся в платформозависимых библиотеках C и предназначены для выполнения сложных математических операций с использованием значений с плавающей точкой, включая логарифмические и тригонометрические операции.

5.4.1. Специальные константы

Во многих математических операциях используются специальные константы. Модуль `math` включает значения π (число "пи"), **e** (основание натуральных логарифмов), **nan** ("не число") и **Infinity** (бесконечность).

Листинг 5.30. `math_constants.py`

```
import math

print('  π: {:.30f}'.format(math.pi))
print('  e: {:.30f}'.format(math.e))
print('nan: {:.30f}'.format(math.nan))
print('inf: {:.30f}'.format(math.inf))
```

Точность как π, так и **e** ограничена только точностью чисел с плавающей точкой в соответствии с установленной на данной платформе библиотекой C.

```
$ python3 math_constants.pyп

  π: 3.141592653589793115997963468544
  e: 2.718281828459045090795598298428
nan: nan
inf: inf
```

[6] https://docs.python.org/3.5/library/random.html

[7] https://ru.wikipedia.org/wiki/Вихрь_Мерсенна

[8] https://ru.wikipedia.org/wiki/Непрерывное_равномерное_распределение

5.4.2. Тестирование исключительных значений

Операции над числами с плавающей точкой могут приводить к двум типам специальных значений. Первое из них – это `inf` (бесконечность), которое возникает в результате переполнения разрядов, отведенных для хранения чисел с двойной точностью, в случае значений с плавающей точкой, имеющих большую абсолютную величину.

Листинг 5.31. `math_isinf.py`

```
import math

import math

print('{:^3} {:6} {:6} {:6}'.format(
    'e', 'x', 'x**2', 'isinf'))
print('{:-^3} {:-^6} {:-^6} {:-^6}'.format(
    '', '', '', ''))

for e in range(0, 201, 20):
    x = 10.0 ** e
    y = x * x
    print('{:3d} {:<6g} {:<6g} {!s:6}'.format(
        e, x, y, math.isinf(y),
    ))
```

Если значение показателя степени (`e`) в этом примере становится достаточно большим, то значение, получаемое для `x**2`, уже не умещается в том количестве разрядов, которое отведено для чисел двойной точности, и выводится как бесконечное.

```
$ python3 math_isinf.py

 e  x      x**2   isinf
--- ------ ------ ------
  0 1      1      False
 20 1e+20  1e+40  False
 40 1e+40  1e+80  False
 60 1e+60  1e+120 False
 80 1e+80  1e+160 False
100 1e+100 1e+200 False
120 1e+120 1e+240 False
140 1e+140 1e+280 False
160 1e+160 inf    True
180 1e+180 inf    True
200 1e+200 inf    True
```

Однако не все переполнения для чисел с плавающей точкой приводят к бесконечному результату. В частности, попытка возведения в степень больших чисел с плавающей точкой приводит не к сохранению значения `inf` в качестве результата, а к возбуждению исключения `OverflowError`.

Листинг 5.32. `math_overflow.py`

```
x = 10.0 ** 200

print('x    =', x)
print('x*x  =', x * x)
print('x**2 =', end=' ')
try:
    print(x ** 2)
except OverflowError as err:
    print(err)
```

Это расхождение обусловлено различиями в реализации библиотеки, используемой в CPython.

```
$ python3 math_overflow.py

x    = 1e+200
x*x  = inf
x**2 = (34, 'Result too large')
```

Операции деления с участием бесконечных значений не определены. Результатом деления числа на бесконечность является `nan` ("не число").

Листинг 5.33. `math_isnan.py`

```
import math

x = (10.0 ** 200) * (10.0 ** 200)
y = x / x

print('x =', x)
print('isnan(x) =', math.isnan(x))
print('y = x / x =', x / x)
print('y == nan =', y == float('nan'))
print('isnan(y) =', math.isnan(y))
```

Значение `nan` не равно ни одному другому значению, в том числе и самому себе. Таким образом, самый простой способ проверки на `nan` — это использовать функцию `isnan()`.

```
$ python3 math_isnan.py

x = inf
isnan(x) = False
y = x / x = nan
y == nan = False
isnan(y) = True
```

Чтобы выяснить, является ли значение обычным числом или одним из значений `inf` или `nan`, используйте функцию `isfinite()`.

Листинг 5.34. math_isfinite.py

```
import math

for f in [0.0, 1.0, math.pi, math.e, math.inf, math.nan]:
    print('{:5.2f} {!s}'.format(f, math.isfinite(f)))
```

Функция `isfinite()` возвращает значение `False` для любого из исключительных случаев и значение `True` в противном случае.

```
$ python3 math_isfinite.py

 0.00 True
 1.00 True
 3.14 True
 2.72 True
  inf False
  nan False
```

5.4.3. Сравнение

Операции сравнения с участием значений с плавающей точкой подвержены ошибкам, причем на каждом этапе вычислений существует риск возникновения ошибок, обусловленных представлением числа. Функция `isclose()` использует устойчивый алгоритм минимизации подобных ошибок и обеспечивает сравнение с использованием как абсолютного, так и относительного допуска. Соответствующая формула эквивалентна следующему выражению:

```
abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)
```

По умолчанию функция `isclose()` использует относительное сравнение с допуском `1e-09`, означающим, что различие между значениями не должно превышать значения `1e-09`, умноженного на наибольшее из абсолютных значений `a` и `b`. Величину допуска, определяющего близость значений, можно изменить с помощью именованного аргумента `rel_tol` функции `isclose()`. В следующем примере значения должны различаться не более чем на 10%.

Листинг 5.35. math_isclose.py

```
import math

import math

INPUTS = [
    (1000, 900, 0.1),
    (100, 90, 0.1),
    (10, 9, 0.1),
    (1, 0.9, 0.1),
    (0.1, 0.09, 0.1),
]

print('{:^8} {:^8} {:^8} {:^8} {:^8} {:^8}'.format(
    'a', 'b', 'rel_tol', 'abs(a-b)', 'tolerance', 'close')
```

```
)
print('{:-^8} {:-^8} {:-^8} {:-^8} {:-^8} {:-^8}'.format(
    '-', '-', '-', '-', '-', '-'),
)

fmt = '{:8.2f} {:8.2f} {:8.2f} {:8.2f} {:8.2f} {!s:>8}'

for a, b, rel_tol in INPUTS:
    close = math.isclose(a, b, rel_tol=rel_tol)
    tolerance = rel_tol * max(abs(a), abs(b))
    abs_diff = abs(a - b)
    print(fmt.format(a, b, rel_tol, abs_diff, tolerance, close))
```

Результат сравнения значений `0.1` и `0.09` оказывается отрицательным из-за ошибки, связанной с представлением значения `0.1`.

```
$ python3 math_isclose.py

   a        b      rel_tol  abs(a-b) tolerance  close
-------- -------- -------- -------- -------- --------
 1000.00   900.00     0.10   100.00   100.00     True
  100.00    90.00     0.10    10.00    10.00     True
   10.00     9.00     0.10     1.00     1.00     True
    1.00     0.90     0.10     0.10     0.10     True
    0.10     0.09     0.10     0.01     0.01    False
```

Чтобы использовать фиксированный или "абсолютный" допуск, следует передать функции `isclose()` аргумент `abs_tol` вместо аргумента `rel_tol`.

Листинг 5.36. `math_isclose_abs_tol.py`

```
import math

INPUTS = [
    (1.0, 1.0 + 1e-07, 1e-08),
    (1.0, 1.0 + 1e-08, 1e-08),
    (1.0, 1.0 + 1e-09, 1e-08),
]

print('{:^8} {:^11} {:^8} {:^10} {:^8}'.format(
    'a', 'b', 'abs_tol', 'abs(a-b)', 'close')
)
print('{:-^8} {:-^11} {:-^8} {:-^10} {:-^8}'.format(
    '-', '-', '-', '-', '-'),
)

for a, b, abs_tol in INPUTS:
    close = math.isclose(a, b, abs_tol=abs_tol)
    abs_diff = abs(a - b)
    print('{:8.2f} {:11} {:8} {:0.9f} {!s:>8}'.format(
        a, b, abs_tol, abs_diff, close))
```

В случае абсолютного допуска разница между входными значениями должна быть меньше заданного значения.

```
$ python3 math_isclose_abs_tol.py

   a          b       abs_tol   abs(a-b)   close
-------- ---------- -------- ---------- --------
    1.00  1.0000001    1e-08 0.000000100    False
    1.00 1.00000001    1e-08 0.000000010     True
    1.00 1.000000001    1e-08 0.000000001     True
```

Значения nan и inf представляют специальные случаи.

Листинг 5.37. math_isclose_inf.py

```
import math

print('nan, nan:', math.isclose(math.nan, math.nan))
print('nan, 1.0:', math.isclose(math.nan, 1.0))
print('inf, inf:', math.isclose(math.inf, math.inf))
print('inf, 1.0:', math.isclose(math.inf, 1.0))
```

Значение nan не может быть близким ни к какому другому значению, включая самого себя. Значение inf близко только к самому себе.

```
$ python3 math_isclose_inf.py

nan, nan: False
nan, 1.0: False
inf, inf: True
inf, 1.0: False
```

5.4.4. Преобразование значений с плавающей точкой в целые числа

Модуль math включает три функции, предназначенные для преобразования значений с плавающей точкой в целые числа. В этих функциях используются разные подходы к выполнению данного преобразования, и каждая из них оказывается удобной в тех или иных обстоятельствах.

Самая простая из них — функция trunc(), которая отбрасывает дробную часть, оставляя только целочисленную часть значения. Функция floor() округляет значение до наибольшего из предшествующих целых чисел, а функция ceil() — до наименьшего из следующих целых чисел.

Листинг 5.38. math_integers.py

```
import math

HEADINGS = ('i', 'int', 'trunk', 'floor', 'ceil')
print('{:^5} {:^5} {:^5} {:^5} {:^5}'.format(*HEADINGS))
print('{:-^5} {:-^5} {:-^5} {:-^5} {:-^5}'.format(
    '', '', '', '', '',
))

fmt = '{:5.1f} {:5.1f} {:5.1f} {:5.1f} {:5.1f}'
```

```
TEST_VALUES = [
    -1.5,
    -0.8,
    -0.5,
    -0.2,
    0,
    0.2,
    0.5,
    0.8,
    1,
]

for i in TEST_VALUES:
    print(fmt.format(
        i,
        int(i),
        math.trunc(i),
        math.floor(i),
        math.ceil(i),
    ))
```

Функция `trunc()` эквивалентна непосредственному преобразованию в тип `int`.

```
$ python3 math_integers.py

  i     int   trunk  floor  ceil
----- ----- ----- ----- -----
 -1.5  -1.0  -1.0  -2.0  -1.0
 -0.8   0.0   0.0  -1.0   0.0
 -0.5   0.0   0.0  -1.0   0.0
 -0.2   0.0   0.0  -1.0   0.0
  0.0   0.0   0.0   0.0   0.0
  0.2   0.0   0.0   0.0   1.0
  0.5   0.0   0.0   0.0   1.0
  0.8   0.0   0.0   0.0   1.0
  1.0   1.0   1.0   1.0   1.0
```

5.4.5. Альтернативные представления значений с плавающей точкой

Функция `modf()` получает единственный аргумент в виде числа с плавающей точкой и возвращает кортеж, содержащий дробную и целую части входного значения.

Листинг 5.39. `math_modf.py`

```
import math

for i in range(6):
    print('{}/2 = {}'.format(i, math.modf(i / 2.0)))
```

Оба числа в возвращаемом значении являются числами с плавающей точкой.

```
$ python3 math_modf.py

0/2 = (0.0, 0.0)
1/2 = (0.5, 0.0)
2/2 = (0.0, 1.0)
3/2 = (0.5, 1.0)
4/2 = (0.0, 2.0)
5/2 = (0.5, 2.0)
```

Функция `frexp()` возвращает мантиссу и экспоненту числа с плавающей точкой. Эту функцию можно использовать для создания представления значения в более переносимой форме.

Листинг 5.40. `math_frexp.py`

```
import math

print('{:^7} {:^7} {:^7}'.format('x', 'm', 'e'))
print('{:-^7} {:-^7} {:-^7}'.format('', '', ''))

for x in [0.1, 0.5, 4.0]:
    m, e = math.frexp(x)
    print('{:7.2f} {:7.2f} {:7d}'.format(x, m, e))
```

Функция `frexp()` использует формулу `x = m * 2**e` и возвращает значения `m` и `e`.

```
$ python3 math_frexp.py

   x       m       e
------- ------- -------
   0.10    0.80      -3
   0.50    0.50       0
   4.00    0.50       3
```

Функция `ldexp()` — обратная по отношению к функции `frexp()`.

Листинг 5.41. `math_ldexp.py`

```
import math

print('{:^7} {:^7} {:^7}'.format('m', 'e', 'x'))
print('{:-^7} {:-^7} {:-^7}'.format('', '', ''))

INPUTS = [
    (0.8, -3),
    (0.5, 0),
    (0.5, 3),
]

for m, e in INPUTS:
    x = math.ldexp(m, e)
    print('{:7.2f} {:7d} {:7.2f}'.format(m, e, x))
```

Используя ту же формулу, что и функция `frexp()`, функция `ldexp()` получает значения мантиссы и экспоненты и возвращает число с плавающей точкой.

```
$ python3 math_ldexp.py

   m       e       x
------- ------- -------
   0.80      -3    0.10
   0.50       0    0.50
   0.50       3    4.00
```

5.4.6. Знак числа

Абсолютная величина числа — это значение числа с отброшенным знаком. Для вычисления абсолютной величины числа с плавающей точкой используется функция `fabs()`.

Листинг 5.42. `math_fabs.py`

```
import math

print(math.fabs(-1.1))
print(math.fabs(-0.0))
print(math.fabs(0.0))
print(math.fabs(1.1))
```

С практической точки зрения абсолютная величина числа с плавающей точкой представляется положительным значением.

```
$ python3 math_fabs.py

1.1
0.0
0.0
1.1
```

Если требуется определить знак числа, будь то для присвоения его набору значений или для сравнения с другими значениями, используйте функцию `copysign()`, позволяющую установить знак для любого корректного значения.

Листинг 5.43. `math_copysign.py`

```
import math

HEADINGS = ('f', 's', '< 0', '> 0', '= 0')
print('{:^5} {:^5} {:^5} {:^5} {:^5}'.format(*HEADINGS))
print('{:-^5} {:-^5} {:-^5} {:-^5} {:-^5}'.format(
    '', '', '', '', '',
))

VALUES = [
    -1.0,
    0.0,
    1.0,
```

```
    float('-inf'),
    float('inf'),
    float('-nan'),
    float('nan'),
]

for f in VALUES:
    s = int(math.copysign(1, f))
    print('{:5.1f} {:5d} {!s:5} {!s:5} {!s:5}'.format(
        f, s, f < 0, f > 0, f == 0,
    ))
```

Дополнительная функция наподобие `copysign()` нужна по той причине, что непосредственное сравнение `nan` и `-nan` с другими значениями не работает.

```
$ python3 math_copysign.py

  f     s    < 0   > 0   = 0
----- ----- ----- ----- -----
 -1.0    -1 True  False False
  0.0     1 False False True
  1.0     1 False True  False
 -inf    -1 True  False False
  inf     1 False True  False
  nan    -1 False False False
  nan     1 False False False
```

5.4.7. Распространенные виды вычислений

Точное представление значений с плавающей точкой в двоичном виде в памяти машины довольно проблематично. Некоторые значения вообще не могут быть представлены точно. Кроме того, чем чаще такое значение используется в расчетах, тем выше вероятность того, что влияние ошибки представления будет только возрастать. Модуль `math` включает функцию, предназначенную для вычисления суммы последовательности чисел с плавающей точкой с использованием эффективного алгоритма, который минимизирует подобные ошибки.

Листинг 5.44. `math_fsum.py`

```
import math

values = [0.1] * 10

print('Input values:', values)

print('sum()       : {:.20f}'.format(sum(values)))

s = 0.0
for i in values:
    s += i
print('for-loop    : {:.20f}'.format(s))

print('math.fsum() : {:.20f}'.format(math.fsum(values)))
```

Для последовательности 10 значений, каждое из которых равно `0.1`, ожидаемое значение суммы равно `1.0`. Однако ввиду того, что значение `0.1` невозможно представить точно в машинной памяти, эти ошибки, если не использовать функцию `fsum()`, накапливаются в сумме.

```
$ python3 math_fsum.py

Input values: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
sum()       : 0.9999999999999988898
for-loop    : 0.9999999999999988898
math.fsum() : 1.0000000000000000000
```

Функцию `factorial()` обычно используют для расчета количества перестановок или сочетаний элементов последовательности. Факториал положительного числа n, обозначаемый как $n!$, равен произведению всех натуральных чисел от 1 до n включительно и рассчитывается по рекуррентной формуле `(n-1) ! * n`, в которой, в соответствии с общепринятым соглашением, принимается, что `0! == 1`.

Листинг 5.45. `math_factorial.py`

```
import math

for i in [0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.1]:
    try:
        print('{:2.0f} {:6.0f}'.format(i, math.factorial(i)))
    except ValueError as err:
        print('Error computing factorial({}): {}'.format(i, err))
```

Функция `factorial()` работает только с целыми числами, но может получать также аргументы в виде чисел с плавающей точкой, коль скоро они могут быть преобразованы в целые значения без потери значения.

```
$ python3 math_factorial.py

0      1
 1      1
 2      2
 3      6
 4     24
 5    120
Error computing factorial(6.1): factorial() only accepts integral
values
```

Функция `gamma()` похожа на функцию `factorial()`, но работает с вещественными числами, а факториал вычисляется для значения аргумента, уменьшенного на 1 (гамма-функция равна `(n - 1)!`).

Листинг 5.46. `math_gamma.py`

```
import math

for i in [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6]:
    try:
```

```
        print('{:2.1f} {:6.2f}'.format(i, math.gamma(i)))
    except ValueError as err:
        print('Error computing gamma({}): {}'.format(i, err))
```

Нулевое значение аргумента недопустимо, поскольку оно приводит к отрицательному начальному значению факториала.

```
$ python3 math_gamma.py

Error computing gamma(0): math domain error
1.1   0.95
2.2   1.10
3.3   2.68
4.4  10.14
5.5  52.34
6.6 344.70
```

Функция `lgamma()` возвращает натуральный логарифм абсолютной величины гамма-функции для входного значения.

Листинг 5.47. `math_lgamma.py`

```
import math

for i in [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6]:
    try:
        print('{:2.1f} {:.20f} {:.20f}'.format(
            i,
            math.lgamma(i),
            math.log(math.gamma(i)),
        ))
    except ValueError as err:
        print('Error computing lgamma({}): {}'.format(i, err))
```

Использование функции `lgamma()` позволяет увеличить точность конечного результата по сравнению с независимым вычислением гамма-функции и логарифма.

```
$ python3 math_lgamma.py

Error computing lgamma(0): math domain error
1.1 -0.04987244125984036103 -0.04987244125983997245
2.2 0.09694746679063825923 0.09694746679063866168
3.3 0.98709857789473387513 0.98709857789473409717
4.4 2.31610349142485727469 2.31610349142485727469
5.5 3.95781396761871651080 3.95781396761871606671
6.6 5.84268005527463252236 5.84268005527463252236
```

Оператор деления по модулю (`%`) вычисляет остаток от деления (например, `5 % 2 = 1`). Встроенный оператор языка хорошо работает с целыми числами, но, как и в случае многих других операций, выполняемых над числами с плавающей точкой, с промежуточными вычислениями связана проблема точности представления чисел этого типа в машинной памяти, что может приводить к потере дан-

ных. Функция fmod() предоставляет более точную реализацию деления по модулю для значений с плавающей точкой.

Листинг 5.48. math_fmod.py

```
import math

print('{:^4} {:^4} {:^5} {:^5}'.format(
    'x', 'y', '%', 'fmod'))
print('{:-^4} {:-^4} {:-^5} {:-^5}'.format(
    '-', '-', '-', '-'))

INPUTS = [
    (5, 2),
    (5, -2),
    (-5, 2),
]

for x, y in INPUTS:
    print('{:4.1f} {:4.1f} {:5.2f} {:5.2f}'.format(
        x,
        y,
        x % y,
        math.fmod(x, y),
    ))
```

Потенциальным источником недоразумений является тот факт, что алгоритм, используемый функцией fmod() для вычисления остатка, отличается от алгоритма, используемого оператором %, в результате чего получаемые с их помощью результаты могут отличаться знаками.

```
$ python3 math_fmod.py

 x    y     %   fmod
---- ---- ----- -----
 5.0  2.0  1.00  1.00
 5.0 -2.0 -1.00  1.00
-5.0  2.0  1.00 -1.00
```

Функция gcd() позволяет находить наибольший общий делитель двух чисел.

Листинг 5.49. math_gcd.py

```
import math

print(math.gcd(10, 8))
print(math.gcd(10, 0))
print(math.gcd(50, 225))
print(math.gcd(11, 9))
print(math.gcd(0, 0))
```

Если оба значения равны 0, результат равен 0.

```
$ python3 math_gcd.py

2
10
25
1
0
```

5.4.8. Экспоненты и логарифмы

Экспоненциальные кривые встречаются в экономике, физике и других областях науки. В Python имеется встроенный оператор возведения в степень (**), но в тех случаях, когда другой функции необходимо передать аргумент в виде вызываемого объекта, может оказаться полезной функция pow().

Листинг 5.50. math_pow.py

```
import math

INPUTS = [
    # Типичное использование
    (2, 3),
    (2.1, 3.2),

    # Всегда 1
    (1.0, 5),
    (2.0, 0),

    # Не число
    (2, float('nan')),

    # Корни
    (9.0, 0.5),
    (27.0, 1.0 / 3),
]

for x, y in INPUTS:
    print('{:5.1f} ** {:5.3f} = {:6.3f}'.format(
        x, y, math.pow(x, y)))
```

Результатом возведения 1 в любую степень, как и результатом возведения любого числа в степень 0.0, всегда является 1.0.

В большинстве случаев операции с nan ("не число") возвращают значение nan. Если показатель степени меньше 1, функция pow() вычисляет корень.

```
$ python3 math_pow.py

  2.0 ** 3.000 =  8.000
  2.1 ** 3.200 = 10.742
  1.0 ** 5.000 =  1.000
  2.0 ** 0.000 =  1.000
  2.0 **   nan =    nan
```

```
  9.0 ** 0.500 =  3.000
 27.0 ** 0.333 =  3.000
```

Ввиду того что необходимость в вычислении квадратных корней (показатель степени 1/2) возникает довольно часто, для этой операции предусмотрена отдельная функция.

Листинг 5.51. math_sqrt.py

```
import math

print(math.sqrt(9.0))
print(math.sqrt(3))
try:
    print(math.sqrt(-1))
except ValueError as err:
    print('Cannot compute sqrt(-1):', err)
```

Вычисление квадратных корней из отрицательных значений требует использования комплексных чисел, с которыми модуль math не работает. Любая попытка вычислить квадратный корень из отрицательного числа приводит к ошибке ValueError.

```
$ python3 math_sqrt.py

3.0
1.7320508075688772
Cannot compute sqrt(-1): math domain error
```

Логарифмическая функция находит такое значение y, при котором x = b ** y. По умолчанию функция log() вычисляет натуральный логарифм (с основанием e). Если используется второй аргумент, то он задает основание логарифма.

Листинг 5.52. math_log.py

```
import math

print(math.log(8))
print(math.log(8, 2))
print(math.log(0.5, 2))
```

Логарифмы для значений x меньше 1 являются отрицательными значениями.

```
$ python3 math_log.py

2.0794415416798357
3.0
-1.0
```

Существуют три разновидности функции log(). При заданном представлении числа с плавающей точкой и ошибках округления значение, получаемое с помощью функции log(x,b), имеет ограниченную точность, особенно для некоторых оснований логарифмов. Функция log10() вычисляет результат вызова функции log(x,10), используя более точный алгоритм, чем функция log().

Листинг 5.53. math_log10.py

```
import math

print('{:2} {:^12} {:^10} {:^20} {:8}'.format(
    'i', 'x', 'accurate', 'inaccurate', 'mismatch',
))
print('{:-^2} {:-^12} {:-^10} {:-^20} {:-^8}'.format(
    '', '', '', '', '',
))

for i in range(0, 10):
    x = math.pow(10, i)
    accurate = math.log10(x)
    inaccurate = math.log(x, 10)
    match = '' if int(inaccurate) == i else '*'
    print('{:2d} {:12.1f} {:10.8f} {:20.18f} {:^5}'.format(
        i, x, accurate, inaccurate, match,
    ))
```

Замыкающими символами * в строках вывода обозначены неточные результаты.

```
$ python3 math_log10.py

i       x         accurate       inaccurate       mismatch
-- ------------ ---------- -------------------- --------
 0          1.0 0.00000000 0.000000000000000000
 1         10.0 1.00000000 1.000000000000000000
 2        100.0 2.00000000 2.000000000000000000
 3       1000.0 3.00000000 2.999999999999999556   *
 4      10000.0 4.00000000 4.000000000000000000
 5     100000.0 5.00000000 5.000000000000000000
 6    1000000.0 6.00000000 5.999999999999999112   *
 7   10000000.0 7.00000000 7.000000000000000000
 8  100000000.0 8.00000000 8.000000000000000000
 9 1000000000.0 9.00000000 8.999999999999998224   *
```

Аналогично функции `log10()`, функция `log2()` вычисляет эквивалент результата вызова функции `math.log(x,2)`.

Листинг 5.54. math_log2.py

```
import math

print('{:>2} {:^5} {:^5}'.format(
    'i', 'x', 'log2',
))
print('{:-^2} {:-^5} {:-^5}'.format(
    '', '', '',
))

for i in range(0, 10):
    x = math.pow(2, i)
```

```
    result = math.log2(x)
    print('{:2d} {:5.1f} {:5.1f}'.format(
        i, x, result,
    ))
```

В зависимости от платформы, встроенные и специальные функции могут обеспечивать более высокие показатели производительности и точности, используя преимущества специальных алгоритмов для основания 2, чем более общие функции.

```
$ python3 math_log2.py

 i    x   log2
-- ----- -----
 0   1.0   0.0
 1   2.0   1.0
 2   4.0   2.0
 3   8.0   3.0
 4  16.0   4.0
 5  32.0   5.0
 6  64.0   6.0
 7 128.0   7.0
 8 256.0   8.0
 9 512.0   9.0
```

Функция `log1p()` используется для вычисления рядов Ньютона–Меркатора (натуральный логарифм `1 + x`).

Листинг 5.55. `math_log1p.py`

```
import math

x = 0.00000000000000000000000001
print('x       :', x)
print('1 + x   :', 1 + x)
print('log(1+x):', math.log(1 + x))
print('log1p(x):', math.log1p(x))
```

Функция `log1p()` дает более точные результаты для значений `x`, близких к нулю, поскольку использует алгоритм, который компенсирует ошибки округления, возникающие при выполнении начальной операции сложения.

```
$ python3 math_log1p.py

x       : 1e-25
1 + x   : 1.0
log(1+x): 0.0
log1p(x): 1e-25
```

Функция `exp()` вычисляет экспоненциальную функцию (`e**x`).

Листинг 5.56. math_exp.py

```
import math

x = 2

fmt = '{:.20f}'
print(fmt.format(math.e ** 2))
print(fmt.format(math.pow(math.e, 2)))
print(fmt.format(math.exp(2)))
```

Подобно другим специальным функциям, функция `exp()` использует алгоритм, позволяющий получать более точные результаты, чем эквивалентная ей универсальная функция `math.pow(math.e,x)`.

```
$ python3 math_exp.py

7.38905609893064951876
7.38905609893064951876
7.38905609893065040694
```

Функция `expm1()` — обратная по отношению к функции `log1p()` и вычисляет выражение `e**x - 1`.

Листинг 5.57. math_expm1.py

```
import math

x = 0.000000000000000000000001

print(x)
print(math.exp(x) - 1)
print(math.expm1(x))
```

При малых значениях x выполнение вычитания как отдельной операции приводит к потере точности, аналогично функции `log1p()`.

```
$ python3 math_expm1.py

1e-25
0.0
1e-25
```

5.4.9. Углы

Несмотря на то что в повседневной жизни углы измеряют в градусах, в математике и других научных областях в качестве стандартной единицы измерения углов используют радианы. Радиан — это угол между двумя лучами, выходящими из центра окружности, которому соответствует длина дуги окружности, равная радиусу.

Длина окружности равна $2\pi r$, поэтому радианы связаны с числом π, которое часто используется в тригонометрических расчетах. Существование этого отно-

шения приводит к тому, что радианы используются в тригонометрии и дифференциальном исчислении, придавая формулам более компактный вид.

Градусы можно преобразовать в радианы с помощью функции `radians()`.

Листинг 5.58. `math_radians.py`

```
import math

print('{:^7} {:^7} {:^7}'.format(
    'Degrees', 'Radians', 'Expected'))
print('{:-^7} {:-^7} {:-^7}'.format(
    '', '', ''))

INPUTS = [
    (0, 0),
    (30, math.pi / 6),
    (45, math.pi / 4),
    (60, math.pi / 3),
    (90, math.pi / 2),
    (180, math.pi),
    (270, 3 / 2.0 * math.pi),
    (360, 2 * math.pi),
]

for deg, expected in INPUTS:
    print('{:7d} {:7.2f} {:7.2f}'.format(
        deg,
        math.radians(deg),
        expected,
    ))
```

Формула преобразования выглядит так: `rad = grad * π / 180`.

```
$ python3 math_radians.py

Degrees Radians Expected
------- ------- -------
      0    0.00    0.00
     30    0.52    0.52
     45    0.79    0.79
     60    1.05    1.05
     90    1.57    1.57
    180    3.14    3.14
    270    4.71    4.71
    360    6.28    6.28
```

Для преобразования радианов в градусы используйте функцию `degrees()`.

Листинг 5.59. `math_degrees.py`

```
import math

INPUTS = [
    (0, 0),
```

```
    (math.pi / 6, 30),
    (math.pi / 4, 45),
    (math.pi / 3, 60),
    (math.pi / 2, 90),
    (math.pi, 180),
    (3 * math.pi / 2, 270),
    (2 * math.pi, 360),
]

print('{:^8} {:^8} {:^8}'.format(
    'Radians', 'Degrees', 'Expected'))
print('{:-^8} {:-^8} {:-^8}'.format('', '', ''))
for rad, expected in INPUTS:
    print('{:8.2f} {:8.2f} {:8.2f}'.format(
        rad,
        math.degrees(rad),
        expected,
    ))
```

Формула этого преобразования выглядит так: `deg = rad * 180 / π`.

```
$ python3 math_degrees.py

Radians  Degrees  Expected
-------- -------- --------
    0.00     0.00     0.00
    0.52    30.00    30.00
    0.79    45.00    45.00
    1.05    60.00    60.00
    1.57    90.00    90.00
    3.14   180.00   180.00
    4.71   270.00   270.00
    6.28   360.00   360.00
```

5.4.10. Тригонометрия

Тригонометрические функции связывают углы треугольника с длинами его сторон. Они появляются в формулах, описывающих периодические свойства, такие как гармоники или круговое движение, а также в формулах для расчета углов. Все тригонометрические функции, содержащиеся в стандартной библиотеке, поддерживают углы, выраженные в радианах.

В прямоугольном треугольнике *синус* любого из его острых углов равен отношению длины противолежащего катета a к длине гипотенузы c ($\sin(A) = a/c$). *Косинус* угла — это отношение длины прилежащего катета b к гипотенузе c ($\cos(A) = b/c$). *Тангенс* угла — это отношение противолежащего катета к прилежащему ($\tan(A) = b/a$).

Листинг 5.60. `math_trig.py`

```
import math

print('{:^7} {:^7} {:^7} {:^7} {:^7}'.format(
```

```
    'Degrees', 'Radians', 'Sine', 'Cosine', 'Tangent'))
print('{:-^7} {:-^7} {:-^7} {:-^7} {:-^7}'.format(
    '-', '-', '-', '-', '-'))

fmt = '{:7.2f} {:7.2f} {:7.2f} {:7.2f} {:7.2f}'

for deg in range(0, 361, 30):
    rad = math.radians(deg)
    if deg in (90, 270):
        t = float('inf')
    else:
        t = math.tan(rad)
    print(fmt.format(deg, rad, math.sin(rad), math.cos(rad), t))
```

Тангенс также можно определить как отношение синуса угла к его косинусу. Поскольку для углов величиной $\pi/2$ и $3\pi/2$ радиан косинус равен 0, тангенс этих углов равен бесконечности.

```
$ python3 math_trig.py

Degrees Radians  Sine   Cosine  Tangent
------- ------- ------- ------- -------
   0.00    0.00    0.00    1.00    0.00
  30.00    0.52    0.50    0.87    0.58
  60.00    1.05    0.87    0.50    1.73
  90.00    1.57    1.00    0.00     inf
 120.00    2.09    0.87   -0.50   -1.73
 150.00    2.62    0.50   -0.87   -0.58
 180.00    3.14    0.00   -1.00   -0.00
 210.00    3.67   -0.50   -0.87    0.58
 240.00    4.19   -0.87   -0.50    1.73
 270.00    4.71   -1.00   -0.00     inf
 300.00    5.24   -0.87    0.50   -1.73
 330.00    5.76   -0.50    0.87   -0.58
 360.00    6.28   -0.00    1.00   -0.00
```

Если задана точка с координатами `(x,y)`, то длина гипотенузы прямоугольного треугольника, вершины которого имеют координаты `[(0, 0), (x, 0), (x, y)]`, равна `(x**2 + y**2) ** 1/2`. Ее можно вычислить с помощью функции `hypot()`.

Листинг 5.61. `math_hypot.py`

```
import math

print('{:^7} {:^7} {:^10}'.format('X', 'Y', 'Hypotenuse'))
print('{:-^7} {:-^7} {:-^10}'.format('', '', ''))

POINTS = [
    # простые точки
    (1, 1),
    (-1, -1),
    (math.sqrt(2), math.sqrt(2)),
```

```
    (3, 4),  # треугольник со сторонами 3-4-5
    # для точки окружности
    (math.sqrt(2) / 2, math.sqrt(2) / 2),  # pi/4 радиана
    (0.5, math.sqrt(3) / 2),  # pi/3 радиана
]

for x, y in POINTS:
    h = math.hypot(x, y)
    print('{:7.2f} {:7.2f} {:7.2f}'.format(x, y, h))
```

Для точек окружности единичного радиуса гипотенуза треугольника всегда равна 1.

```
$ python3 math_hypot.py

   X       Y    Hypotenuse
------- ------- ----------
   1.00    1.00    1.41
  -1.00   -1.00    1.41
   1.41    1.41    2.00
   3.00    4.00    5.00
   0.71    0.71    1.00
   0.50    0.87    1.00
```

Те же самые функции можно использовать для вычисления расстояния между двумя точками.

Листинг 5.62. math_distance_2_points.py

```
import math

print('{:^8} {:^8} {:^8} {:^8} {:^8}'.format(
    'X1', 'Y1', 'X2', 'Y2', 'Distance',
))
print('{:-^8} {:-^8} {:-^8} {:-^8} {:-^8}'.format(
    '', '', '', '', '',
))

POINTS = [
    ((5, 5), (6, 6)),
    ((-6, -6), (-5, -5)),
    ((0, 0), (3, 4)),  # треугольник со сторонами 3-4-5
    ((-1, -1), (2, 3)),  # треугольник со сторонами 3-4-5
]

for (x1, y1), (x2, y2) in POINTS:
    x = x1 - x2
    y = y1 - y2
    h = math.hypot(x, y)
    print('{:8.2f} {:8.2f} {:8.2f} {:8.2f} {:8.2f}'.format(
        x1, y1, x2, y2, h,
    ))
```

Здесь начало координат перенесено во вторую точку, и координаты первой точки, рассчитанные относительно нового начала координат как разности соответствующих исходных значений, передаются функции `hypot()`.

```
$ python3 math_distance_2_points.py

   X1       Y1       X2       Y2     Distance
-------- -------- -------- -------- --------
    5.00     5.00     6.00     6.00     1.41
   -6.00    -6.00    -5.00    -5.00     1.41
    0.00     0.00     3.00     4.00     5.00
   -1.00    -1.00     2.00     3.00     5.00
```

Модуль `math` также определяет обратные тригонометрические функции.

Листинг 5.63. `math_inverse_trig.py`

```
import math

for r in [0, 0.5, 1]:
    print('arcsine({:.1f})    = {:5.2f}'.format(r, math.asin(r)))
    print('arccosine({:.1f})  = {:5.2f}'.format(r, math.acos(r)))
    print('arctangent({:.1f}) = {:5.2f}'.format(r, math.atan(r)))
    print()
```

Значение `1.57`, приблизительно равное $\pi/2$, или 90 градусам, — это угол, синус которого равен 1, а косинус — 0.

```
$ python3 math_inverse_trig.py

arcsine(0.0) = 0.00
arccosine(0.0) = 1.57
arctangent(0.0) = 0.00

arcsine(0.5) = 0.52
arccosine(0.5) = 1.05
arctangent(0.5) = 0.46

arcsine(1.0) = 1.57
arccosine(1.0) = 0.00
arctangent(1.0) = 0.79
```

5.4.11. Гиперболические функции

Гиперболические функции встречаются в линейных дифференциальных уравнениях и используются при описании электромагнитных полей, динамики жидкостей, а также в других областях физики и высшей математики.

Листинг 5.64. `math_hyperbolic.py`

```
import math

print('{:^6} {:^6} {:^6} {:^6}'.format(
```

```
    'X', 'sinh', 'cosh', 'tanh',
))
print('{:-^6} {:-^6} {:-^6} {:-^6}'.format('', '', '', ''))

fmt = '{:6.4f} {:6.4f} {:6.4f} {:6.4f}'

for i in range(0, 11, 2):
    x = i / 10.0
    print(fmt.format(
        x,
        math.sinh(x),
        math.cosh(x),
        math.tanh(x),
    ))
```

В то время как синус и косинус описывают окружность, гиперболические синус и косинус формируют половину гиперболы.

```
$ python3 math_hyperbolic.py

  X     sinh   cosh   tanh
------ ------ ------ ------
0.0000 0.0000 1.0000 0.0000
0.2000 0.2013 1.0201 0.1974
0.4000 0.4108 1.0811 0.3799
0.6000 0.6367 1.1855 0.5370
0.8000 0.8881 1.3374 0.6640
1.0000 1.1752 1.5431 0.7616
```

Также доступны обратные тригонометрические функции `acosh()`, `asinh()` и `atanh()`.

5.4.12. Специальные функции

Гауссова функция ошибок используется в статистике.

Листинг 5.65. `math_erf.py`

```
import math

print('{:^5} {:7}'.format('x', 'erf(x)'))
print('{:-^5} {:-^7}'.format('', ''))

for x in [-3, -2, -1, -0.5, -0.25, 0, 0.25, 0.5, 1, 2, 3]:
    print('{:5.2f} {:7.4f}'.format(x, math.erf(x)))
```

Для функции ошибок справедливо соотношение `erf(-x) == -erf(x)`.

```
$ python3 math_erf.py

  x   erf(x)
----- -------
-3.00 -1.0000
-2.00 -0.9953
```

```
-1.00 -0.8427
-0.50 -0.5205
-0.25 -0.2763
 0.00  0.0000
 0.25  0.2763
 0.50  0.5205
 1.00  0.8427
 2.00  0.9953
 3.00  1.0000
```

Дополнительная функция ошибок `erfc()` вычисляет значения, эквивалентные выражению `1 - erf(x)`.

Листинг 5.66. `math_erfc.py`

```
import math

print('{:^5} {:7}'.format('x', 'erfc(x)'))
print('{:-^5} {:-^7}'.format('', ''))

for x in [-3, -2, -1, -0.5, -0.25, 0, 0.25, 0.5, 1, 2, 3]:
    print('{:5.2f} {:7.4f}'.format(x, math.erfc(x)))
```

Реализация функции `erfc()` устраняет потерю точности для небольших значений `x` при вычитании из `1`.

```
$ python3 math_erfc.py

  x    erfc(x)
----- -------
-3.00  2.0000
-2.00  1.9953
-1.00  1.8427
-0.50  1.5205
-0.25  1.2763
 0.00  1.0000
 0.25  0.7237
 0.50  0.4795
 1.00  0.1573
 2.00  0.0047
 3.00  0.0000
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `math`[9].
- *IEEE floating point arithmetic in Python*[10]. Статья, посвященная специальным значениям и их обработке при выполнении математических вычислений в Python.

[9] https://docs.python.org/3.5/library/math.html

[10] www.johndcook.com/blog/2009/07/21/ieee-arithmetic-python/

- SciPy[11]. Библиотека с открытым исходным кодом, предназначенная для выполнения научных и математических вычислений в программах на языке Python.
- PEP 485[12]. *A Function for testing approximate equality.*

5.5. statistics: статистические расчеты

Модуль statistics реализует многие известные статистические формулы, обеспечивающие проведение эффективных вычислений с использованием различных числовых типов Python (int, float, Decimal и Fraction).

5.5.1. Средние значения

Поддерживаются три разновидности усредненных значений: среднее значение, медиана и мода. Среднее значение вычисляется с помощью функции mean().

Листинг 5.67. statistics_mean.py

```
from statistics import *

data = [1, 2, 2, 5, 10, 12]

print('{:0.2f}'.format(mean(data)))
```

Возвращаемым значением для типов int и float всегда является float. Для типов Decimal и Fraction тип результата всегда совпадает с типом входных данных.

```
$ python3 statistics_mean.py

5.33
```

Для вычисления значения, наиболее часто встречающегося в наборе данных, предназначена функция mode().

Листинг 5.68. statistics_mode.py

```
from statistics import *

data = [1, 2, 2, 5, 10, 12]

print(mode(data))
```

Возвращаемое значение всегда является элементом входного набора данных. Поскольку функция mode() обрабатывает входные данные как набор дискретных значений и лишь подсчитывает количество вхождений каждого из них, входные данные необязательно должны быть числовыми значениями.

```
$ python3 statistics_mode.py

2
```

[11] scipy.org

[12] www.python.org/dev/peps/pep-0485

Для вычисления медианы предусмотрены четыре функции. Первые три из них — это разновидности обычного алгоритма, предоставляющие два разных решения при обработке наборов данных, содержащих четное количество элементов.

Листинг 5.69. `statistics_median.py`

```
from statistics import *

data = [1, 2, 2, 5, 10, 12]

print('median     : {:0.2f}'.format(median(data)))
print('low        : {:0.2f}'.format(median_low(data)))
print('high       : {:0.2f}'.format(median_high(data)))
```

Функция `median()` находит центральное значение. Если набор данных содержит четное количество элементов, она усредняет значения двух центральных элементов. Функция `median_low()` всегда возвращает значение, принадлежащее набору входных данных, используя меньшее из двух центральных значений в случае наборов данных с четным количеством элементов. Функция `median_high()` работает аналогичным образом, возвращая большее из двух центральных элементов.

```
$ python3 statistics_median.py

median     : 3.50
low        : 2.00
high       : 5.00
```

Четвертая версия функций для расчета медианы, `median_grouped()`, обрабатывает входные значения как непрерывные данные. Она рассчитывает 50-й процентиль, соответствующий медиане, посредством предварительного определения диапазона медианы на основе предоставленной ширины интервала и последующей интерполяции в пределах этого диапазона с использованием фактических значений набора данных, попадающих в этот диапазон.

Листинг 5.70. `statistics_median_grouped.py`

```
from statistics import *

data = [10, 20, 30, 40]

print('1: {:0.2f}'.format(median_grouped(data, interval=1)))
print('2: {:0.2f}'.format(median_grouped(data, interval=2)))
print('3: {:0.2f}'.format(median_grouped(data, interval=3)))
```

При изменении ширины интервала значение медианы, вычисленное для одного и того же набора данных, также изменяется.

```
$ python3 statistics_median_grouped.py

1: 29.50
2: 29.00
3: 28.50
```

5.5.2. Дисперсия

Для выражения степени разброса значений относительно среднего по набору статистики используют две характеристики. *Дисперсия* – это среднее значение квадрата разности между каждым из значений и средним, а *стандартное отклонение* – это квадратный корень из дисперсии (эта характеристика удобна тем, что выражается в тех же единицах измерения, что и входные данные). Большие значения дисперсии или стандартного отклонения означают большую степень разброса данных в наборе, тогда как их небольшие значения свидетельствуют о кластеризации данных вокруг среднего значения.

Листинг 5.71. statistics_variance.py

```
from statistics import *
import subprocess


def get_line_lengths():
    cmd = 'wc -l ../[a-z]*/*.py'
    out = subprocess.check_output(
        cmd, shell=True).decode('utf-8')
    for line in out.splitlines():
        parts = line.split()
        if parts[1].strip().lower() == 'total':
            break
        nlines = int(parts[0].strip())
        if not nlines:
            continue  # пропустить пустые файлы
        yield (nlines, parts[1].strip())


data = list(get_line_lengths())

lengths = [d[0] for d in data]
sample = lengths[::2]

print('Basic statistics:')
print('  count     : {:3d}'.format(len(lengths)))
print('  min       : {:6.2f}'.format(min(lengths)))
print('  max       : {:6.2f}'.format(max(lengths)))
print('  mean      : {:6.2f}'.format(mean(lengths)))

print('\nPopulation variance:')
print('  pstdev    : {:6.2f}'.format(pstdev(lengths)))
print('  pvariance : {:6.2f}'.format(pvariance(lengths)))

print('\nEstimated variance for sample:')
print('  count     : {:3d}'.format(len(sample)))
print('  stdev     : {:6.2f}'.format(stdev(sample)))
print('  variance  : {:6.2f}'.format(variance(sample)))
```

Python включает два набора функций для вычисления дисперсии и стандартного отклонения, которые используются в зависимости от того, что именно пред-

ставляет собой набор данных: генеральную совокупность или выборку из нее. В данном примере сначала с помощью команды `wc` подсчитывается общее количество строк во всех файлах примеров. Далее с помощью функций `pvariance()` и `pstdev()` вычисляются дисперсия и стандартное отклонение для генеральной совокупности. В завершение с помощью функций `variance()` и `stddev()` вычисляются выборочные значения дисперсии и стандартного отклонения для поднабора данных, созданного на основании размера каждого второго из найденных файлов.

```
$ python3 statistics_variance.py

Basic statistics:
  count     : 959
  min       : 4.00
  max       : 228.00
  mean      : 28.62

Population variance:
  pstdev    : 18.52
  pvariance : 342.95

Estimated variance for sample:
  count     : 480
  stdev     : 21.09
  variance  : 444.61
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `statistics`[13].
- *Median for Discrete and Continuous Frequency Type Data (grouped data)*[14]. Обсуждение медианы для непрерывных данных.
- **PEP 450**[15]. *Adding A Statistics Module To The Standard Library.*

[13] https://docs.python.org/3.5/library/statistics.html

[14] www.mathstips.com/statistics/median-for-discrete-and-continuous-frequency-type.html

[15] www.python.org/dev/peps/pep-0450

Глава 6

Файловая система

Стандартная библиотека Python включает множество инструментов, предназначенных для работы с файловой системой, создания и синтаксического анализа имен файлов, а также исследования их содержимого.

Первое, что необходимо сделать, — это определить имя файла, с которым вы собираетесь работать. Python использует в качестве имен файлов простые строки, но предоставляет инструменты, позволяющие создавать имена файлов из стандартных платформонезависимых элементов, хранящихся в модуле `os.path` (раздел 6.1).

Модуль `pathlib` (раздел 6.2) предоставляет объектно-ориентированный API для работы с путями файловой системы. Он обеспечивает большие удобства по сравнению с модулем `os.path`, поскольку оперирует объектами на более высоком уровне абстракции.

Функция `listdir()` из модуля `os` (раздел 17.3) позволяет получить список содержимого каталога, в то время как модуль `glob` (раздел 6.3) обеспечивает создание списка имен файлов, определяемых шаблоном.

Модуль `fnmatch` (раздел 6.4) предоставляет непосредственный доступ к средствам сопоставления имен файлов с шаблоном, используемым модулем `glob`, что обеспечивает возможность их применения в других контекстах.

После того как имя файла установлено, можно проверить другие его характеристики, такие как права доступа или размер файла, используя функцию `os.stat()` и константы, хранящиеся в модуле `stat`.

Если приложению нужен произвольный доступ к файлам, модуль `linecache` (раздел 6.5) упростит их построчное чтение по номерам строк. Однако не следует забывать о том, что содержимое файла сохраняется в кеше, что увеличивает потребление оперативной памяти.

Модуль `tempfile` (раздел 6.6) удобно использовать в тех случаях, когда требуется обеспечить временное сохранение файлов или подготовить их к перемещению в место постоянного хранения. Содержащиеся в нем классы предоставляют надежные и безопасные способы создания файлов и каталогов. При этом гарантируется уникальность имен и включение в них случайных элементов, затрудняющих определение назначения файлов по их именам.

Программам нередко приходится манипулировать файлами как таковыми, независимо от их содержимого. Модуль `shutil` (раздел 6.7) включает высокоуровневые файловые операции, такие как копирование файлов и каталогов и создание или извлечение архивированных файлов.

Модуль `filecmp` (раздел 6.8) позволяет сравнивать между собой файлы и каталоги на основе их байтового содержимого, но без использования каких-либо специальных особенностей их формата.

Для чтения и записи файлов, видимых в локальной файловой системе, можно использовать встроенный класс `file`. Однако при доступе к файлам большо-

го размера посредством интерфейсов `theread()` и `write()` может снижаться производительность программы, поскольку соответствующие операции требуют многократного копирования данных при их перемещении с диска в оперативную память, доступную приложению. Модуль `mmap` (раздел 6.9) позволяет известить операционную систему о необходимости использования подсистемы виртуальной памяти для отображения содержимого файла в память, к которой программа может обращаться непосредственно, что позволяет избежать копирования содержимого между операционной системой и внутренним буфером объекта `file`.

Обычно текстовые данные, которые содержат символы, отсутствующие в таблице ASCII, сохраняются в формате Unicode. Поскольку при стандартной обработке файлов предполагается, что каждый байт представляет один символ, то чтение текста Unicode, использующего многобайтовую кодировку, означает необходимость выполнения дополнительной работы по декодированию текста. Модуль `codecs` (раздел 6.10) автоматически выполняет необходимое кодирование и декодирование текста, поэтому во многих случаях работа с файлами в кодировке, отличной от ASCII , не требует внесения каких-либо изменений в программу.

Модуль `io` (раздел 6.11) предоставляет доступ к классам, которые используются для реализации операций файлового ввода-вывода в Python. Для тестирования кода, зависящего от файловых операций чтения-записи, модуль `io` предоставляет объект потока в памяти, который ведет себя подобно файлу, но не располагается на диске, а хранится в оперативной памяти.

6.1. `os.path`: платформонезависимое манипулирование именами файлов

Написание кода для работы с файлами, способного выполняться на многих платформах, значительно упрощается, если использовать функции, включенные в модуль `os.path`. Модуль `os.path` имеет смысл использовать даже в тех программах, которые не предполагается переносить на другие платформы, поскольку это позволяет повысить надежность синтаксического анализа имен файлов.

6.1.1. Анализ путей

Первый из наборов функций модуля `os.path` можно использовать для разбора строк, представляющих имена файлов, на составные элементы. Эти функции никак не связаны с реальными путями и оперируют исключительно строками.

Анализ пути зависит от нескольких переменных, определенных в модуле `os` (раздел 17.3).

- `os.sep`. Разделитель компонентов пути к файлу (например, `/` или `\`).
- `os.extsep`. Разделитель, отделяющий имя файла от расширения (например, `.`).
- `os.pardir`. Компонент пути, означающий переход к родительскому каталогу, расположенному на один уровень выше текущего (например, `..`).
- `os.curdir`. Компонент пути, ссылающийся на текущий каталог (например, `.`).

Функция `split()` разбивает путь на две части, возвращая результат в виде кортежа. Второй элемент кортежа содержит последнюю часть пути, а первый — все, что ей предшествует.

Листинг 6.1. `ospath_split.py`

```
import os.path

PATHS = [
    '/one/two/three',
    '/one/two/three/',
    '/',
    '.',
    '',
]

for path in PATHS:
    print('{!r:>17} : {}'.format(path, os.path.split(path)))
```

Если входной аргумент заканчивается разделителем `os.sep`, последним компонентом пути является пустая строка.

```
$ python3 ospath_split.py

 '/one/two/three' : ('/one/two', 'three')
'/one/two/three/' : ('/one/two/three', '')
              '/' : ('/', '')
              '.' : ('', '.')
               '' : ('', '')
```

Функция `basename()` возвращает базовое имя — значение, эквивалентное второй части пути, возвращаемой функцией `split()`.

Листинг 6.2. `ospath_basename.py`

```
import os.path

PATHS = [
    '/one/two/three',
    '/one/two/three/',
    '/',
    '.',
    '',
]

for path in PATHS:
    print('{!r:>17} : {!r}'.format(path, os.path.basename(path)))
```

Разбивка полного пути осуществляется на последнем компоненте, который ссылается на файл или каталог. Если путь заканчивается разделителем каталогов (`os.sep`), то его базовая часть считается пустой.

```
$ python3 ospath_basename.py

 '/one/two/three' : 'three'
```

```
'/one/two/three/' : ''
                '/' : ''
                '.' : '.'
                 '' : ''
```

Функция `dirname()` возвращает имя каталога для базового имени — первую часть пути, возвращаемую функцией `split()`.

Листинг 6.3. `ospath_dirname.py`

```
import os.path

PATHS = [
    '/one/two/three',
    '/one/two/three/',
    '/',
    '.',
    '',
]

for path in PATHS:
    print('{!r:>17} : {!r}'.format(path, os.path.dirname(path)))
```

Объединение результатов, возвращаемых функциями `basename()` и `dirname()`, дает первоначальный путь.

```
$ python3 ospath_dirname.py

 '/one/two/three' : '/one/two'
'/one/two/three/' : '/one/two/three'
                '/' : '/'
                '.' : ''
                 '' : ''
```

Функция `splitext()` аналогична функции `split()`, но разбивает путь на разделителе расширения, а не на разделителе каталогов.

Листинг 6.4. `ospath_splitext.py`

```
import os.path

PATHS = [
    'filename.txt',
    'filename',
    '/path/to/filename.txt',
    '/',
    '',
    'my-archive.tar.gz',
    'no-extension.',
]

for path in PATHS:
    print('{!r:>21} : {!r}'.format(path, os.path.splitext(path)))
```

При поиске расширения имени файла используется лишь последнее вхождение разделителя `os.extsep`. Таким образом, если имя файла имеет несколько расширений, то часть расширения остается в составе префикса.

```
$ python3 ospath_splitext.py

       'filename.txt' : ('filename', '.txt')
           'filename' : ('filename', '')
'/path/to/filename.txt' : ('/path/to/filename', '.txt')
                  '/' : ('/', '')
                   '' : ('', '')
  'my-archive.tar.gz' : ('my-archive.tar', '.gz')
      'no-extension.' : ('no-extension', '.')
```

Функция `commonprefix()` получает список путей в качестве аргумента и возвращает строку, представляющую префикс, общий для всех путей. Это значение может представлять путь, которого на самом деле не существует, причем последний разделитель каталогов, если он не является общим, не берется в расчет и, как следствие, может отсутствовать.

Листинг 6.5. ospath_commonprefix.py

```
import os.path

paths = ['/one/two/three/four',
         '/one/two/threefold',
         '/one/two/three/',
         ]
for path in paths:
    print('PATH:', path)

print()
print('PREFIX:', os.path.commonprefix(paths))
```

В этом примере общим префиксом является строка `/one/two/three`, несмотря на то, что один из путей вообще не содержит каталог `three`.

```
$ python3 ospath_commonprefix.py

PATH: /one/two/three/four
PATH: /one/two/threefold
PATH: /one/two/three/

PREFIX: /one/two/three
```

Функция `commonpath()` учитывает расположение разделителей каталогов. Она возвращает префикс, который не включает частичные значения путей.

Листинг 6.6. ospath_commonpath.py

```
import os.path

paths = ['/one/two/three/four',
         '/one/two/threefold',
```

```
        '/one/two/three/',
        ]
for path in paths:
    print('PATH:', path)

print()
print('PREFIX:', os.path.commonpath(paths))
```

Поскольку в строке "threefold" разделитель каталогов после подстроки "three" отсутствует, общим префиксом является строка /one/two.

```
$ python3 ospath_commonpath.py

PATH: /one/two/three/four
PATH: /one/two/threefold
PATH: /one/two/three/

PREFIX: \one\two
```

6.1.2. Создание путей

Помимо разбиения путей на отдельные части также требуется создавать пути на основе других строк. Функция join() позволяет объединить несколько компонентов в одну строку пути.

Листинг 6.7. ospath_join.py

```
import os.path

PATHS = [
    ('one', 'two', 'three'),
    ('/', 'one', 'two', 'three'),
    ('/one', '/two', '/three'),
]

for parts in PATHS:
    print('{} : {!r}'.format(parts, os.path.join(*parts)))
```

Если какой-либо из объединяемых компонентов начинается с разделителя os.sep, то все предыдущие компоненты отбрасываются и новый элемент становится начальным компонентом возвращаемого значения.

```
$ python3 ospath_join.py

('one', 'two', 'three') : 'one/two/three'
('/', 'one', 'two', 'three') : '/one/two/three'
('/one', '/two', '/three') : '/three'
```

Также существует возможность работать с путями, включающими "переменные" компоненты, значения которых могут подставляться автоматически. Например, функция expanduser() преобразует символ тильды (~) в имя домашнего каталога пользователя.

Листинг 6.8. ospath_expanduser.py

```
import os.path

for user in ['', 'dhellmann', 'nosuchuser']:
    lookup = '~' + user
    print('{!r:>15} : {!r}'.format(
        lookup, os.path.expanduser(lookup)))
```

Если домашний каталог пользователя, заданный аргументом, найти не удается, то возвращается аргумент без изменений, как это происходит с каталогом ~nosuchuser в данном примере.

```
$ python3 ospath_expanduser.py

             '~' : '/Users/dhellmann'
   '~dhellmann' : '/Users/dhellmann'
  '~nosuchuser' : '~nosuchuser'
```

Более общая функция expandvars() обеспечивает замещение переменных среды командной оболочки, представленных в пути.

Листинг 6.9. ospath_expandvars.py

```
import os.path
import os

os.environ['MYVAR'] = 'VALUE'

print(os.path.expandvars('/path/to/$MYVAR'))
```

При этом проверка того, приводит ли подстановка значения переменной к имени уже существующего файла, не выполняется.

```
$ python3 ospath_expandvars.py

/path/to/VALUE
```

6.1.3. Нормализация путей

В путях, созданных посредством сборки из отдельных строк с помощью функции join() или подстановки значений встроенных переменных, могут оказаться лишние разделители или элементы относительных путей. Функция normpath() удаляет эти элементы.

Листинг 6.10. ospath_normpath.py

```
import os.path

PATHS = [
    'one//two//three',
    'one/./two/./three',
    'one/../alt/two/three',
```

```
]

for path in PATHS:
    print('{!r:>22} : {!r}'.format(path, os.path.normpath(path)))
```

После обработки сегментов пути, включающих элементы `os.curdir` и `os.pardir`, они приводятся к более компактному виду.

```
$ python3 ospath_normpath.py

     'one//two//three' : 'one/two/three'
   'one/./two/./three' : 'one/two/three'
'one/../alt/two/three' : 'alt/two/three'
```

Функция `abspath()` преобразует относительный путь в абсолютное имя файла.

Листинг 6.11. `ospath_abspath.py`

```
import os
import os.path

os.chdir('/usr')

PATHS = [
    '.',
    '..',
    './one/two/three',
    '../one/two/three',
]

for path in PATHS:
    print('{!r:>21} : {!r}'.format(path, os.path.abspath(path)))
```

Результат представляет собой полный путь, начинающийся с вершины дерева файловой системы.

```
$ python3 ospath_abspath.py

                '.' : '/usr'
               '..' : '/'
    './one/two/three' : '/usr/one/two/three'
   '../one/two/three' : '/one/two/three'
```

6.1.4. Временные характеристики файлов

Кроме функций, предназначенных для работы с путями, модуль `os.path` включает функции, позволяющие извлекать свойства файлов, аналогичные тем, которые возвращает функция `os.stat()`.

Листинг 6.12. `ospath_properties.py`

```
import os.path
import time
```

```
print('File         :', __file__)
print('Access time  :', time.ctime(os.path.getatime(__file__)))
print('Modified time:', time.ctime(os.path.getmtime(__file__)))
print('Change time  :', time.ctime(os.path.getctime(__file__)))
print('Size         :', os.path.getsize(__file__))
```

В этом примере функции возвращают следующие значения: `os.path.getatime()` — время последнего доступа к файлу, `os.path.getmtime()` — время последнего изменения файла, `os.path.getctime()` — время создания файла, `os.path.getsize()` — размер файла в байтах.

```
$ python3 ospath_properties.py

File         : ospath_properties.py
Access time  : Fri Aug 26 16:38:05 2016
Modified time: Fri Aug 26 15:50:48 2016
Change time  : Fri Aug 26 15:50:49 2016
Size         : 481
```

6.1.5. Тестирование файлов

Когда программа сталкивается с именем, обозначающим путь, часто требуется знать, к какому объекту он относится — файлу, каталогу или символической ссылке — и существует ли этот объект. Модуль `os.path` включает функции, проверяющие выполнение этих условий.

Листинг 6.13. `ospath_tests.py`

```
import os.path

FILENAMES = [
    __file__,
    os.path.dirname(__file__),
    '/',
    './broken_link',
]

for file in FILENAMES:
    print('File         : {!r}'.format(file))
    print('Absolute     :', os.path.isabs(file))
    print('Is File?     :', os.path.isfile(file))
    print('Is Dir?      :', os.path.isdir(file))
    print('Is Link?     :', os.path.islink(file))
    print('Mountpoint?  :', os.path.ismount(file))
    print('Exists?      :', os.path.exists(file))
    print('Link Exists?:', os.path.lexists(file))
    print()
```

Все тестирующие функции возвращают булевы значения.

```
$ ln -s /does/not/exist broken_link
$ python3 ospath_tests.py
```

```
File        : 'ospath_tests.py'
Absolute    : False
Is File?    : True
Is Dir?     : False
Is Link?    : False
Mountpoint? : False
Exists?     : True
Link Exists?: True

File        : ''
Absolute    : False
Is File?    : False
Is Dir?     : False
Is Link?    : False
Mountpoint? : False
Exists?     : False
Link Exists?: False

File        : '/'
Absolute    : True
Is File?    : False
Is Dir?     : True
Is Link?    : False
Mountpoint? : True
Exists?     : True
Link Exists?: True

File        : './broken_link'
Absolute    : False
Is File?    : False
Is Dir?     : False
Is Link?    : True
Mountpoint? : False
Exists?     : False
Link Exists?: True
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `os.path`[1].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `os.path` (раздел A.6.30).
- `pathlib` (раздел 6.2). Пути как объекты.
- `os` (раздел 17.3). Модуль `os`, являющийся родительским по отношению к модулю `os.path`.
- `time` (см. раздел 4.1). Модуль `time` включает функции, позволяющие выполнять преобразования между представлением, используемым функциями модуля `os.path`, которые возвращают временные характеристики файлов, и удобными для чтения строками.

[1] https://docs.python.org/3.5/library/os.path.html

6.2. pathlib: пути файловой системы как объекты

Модуль pathlib предоставляет объектно-ориентированный API для синтаксического анализа, создания, проверки и других операций, выполняемых над именами и путями, вместо использования низкоуровневых операций со строками.

6.2.1. Представления пути

Модуль pathlib содержит классы для управления путями файловой системы, форматированными либо в соответствии с требованиями стандарта POSIX, либо в соответствии с синтаксисом Windows. Он включает так называемые "чистые" (pure) классы, которые оперируют строками, но не взаимодействуют с фактической файловой системой, и "конкретные" (concrete) классы, которые расширяют API для включения операций, отображающих или изменяющих данные локальной файловой системы.

Чистые классы PurePosixPath и PureWindowsPath могут инстанциализироваться в любой операционной системе, поскольку они работают лишь с именами. Чтобы инстанциализировать подходящий класс для работы с реальной файловой системой, используйте класс Path для получения одного из классов PosixPath или WindowsPath, в зависимости от платформы.

6.2.2. Создание путей

Чтобы инстанциализировать новый класс, предоставьте строку в качестве первого аргумента. Строковым представлением объекта пути является значение этого имени. Чтобы создать новый путь, заданный относительно существующего пути, используйте оператор / для расширения пути. Операндом этого оператора может быть строка или другой объект пути.

Листинг 6.14. pathlib_operator.py

```
import pathlib

usr = pathlib.PurePosixPath('/usr')
print(usr)

usr_local = usr / 'local'
print(usr_local)

usr_share = usr / pathlib.PurePosixPath('share')
print(usr_share)

root = usr / '..'
print(root)

etc = root / '/etc/'
print(etc)
```

Как показывает значение корневого каталога в этом примере, указанный оператор объединяет значения путей в том виде, в каком они предоставляются, и не нормализует результат, если он содержит ссылку на родительский каталог '..'.

Но если сегмент начинается с разделителя каталогов, то он интерпретируется как ссылка на новый корневой каталог точно так же, как и в случае функции `os.path.join()`. Лишние разделители каталогов посреди значения пути удаляются, как это видно здесь на примере каталога `etc`.

```
$ python3 pathlib_operator.py

/usr
/usr/local
/usr/share
/usr/..
/etc
```

Конкретные классы путей включают метод `resolve()`, который нормализует путь, просматривая каталоги и символические ссылки файловой системы и создавая абсолютный путь, на который можно ссылаться по имени.

Листинг 6.15. `pathlib_resolve.py`

```
import pathlib

usr_local = pathlib.Path('/usr/local')
share = usr_local / '..' / 'share'
print(share.resolve())
```

Здесь относительный путь преобразуется в абсолютный путь `/usr/share`. Если входной путь включает символические ссылки, то они тоже раскрываются, позволяя разрешенному пути ссылаться непосредственно на целевой объект.

```
$ python3 pathlib_resolve.py

/usr/share
```

Для создания путей в условиях, когда сегменты пути неизвестны заранее, используйте метод `joinpath()`, передавая ему каждый сегмент в виде отдельного аргумента.

Листинг 6.16. `pathlib_joinpath.py`

```
import pathlib

root = pathlib.PurePosixPath('/')
subdirs = ['usr', 'local']
usr_local = root.joinpath(*subdirs)
print(usr_local)
```

Как и в случае оператора `/`, в результате вызова метода `joinpath()` создается новый экземпляр.

```
$ python3 pathlib_joinpath.py

/usr/local
```

На основе любого существующего объекта пути можно легко создать новый путь, незначительно отличающийся от существующего и ссылающийся, например, на другой файл, находящийся в том же каталоге. Чтобы создать новый путь, отличающийся именем файла, используйте метод `with_name()`. Чтобы создать новый путь, соответствующий тому же имени файла, но с другим расширением, используйте метод `with_suffix()`.

Листинг 6.17. `pathlib_from_existing.py`

```
import pathlib

ind = pathlib.PurePosixPath('source/pathlib/index.rst')
print(ind)

py = ind.with_name('pathlib_from_existing.py')
print(py)

pyc = py.with_suffix('.pyc')
print(pyc)
```

Оба метода возвращают новые объекты, оставляя исходный объект в прежнем состоянии.

```
$ python3 pathlib_from_existing.py

source/pathlib/index.rst
source/pathlib/pathlib_from_existing.py
source/pathlib/pathlib_from_existing.pyc
```

6.2.3. Анализ путей

Объекты пути имеют методы и свойства, позволяющие извлекать из имени значения его составляющих. Например, свойство `parts` предоставляет последовательность сегментов пути, выбираемых на основе разделителя каталогов.

Листинг 6.18. `pathlib_parts.py`

```
import pathlib

p = pathlib.PurePosixPath('/usr/local')
print(p.parts)
```

Указанная последовательность является кортежем, что отражает неизменяемость экземпляра пути.

```
$ python3 pathlib_parts.py

('/', 'usr', 'local')
```

Существуют два способа навигации от данного объекта пути вверх по иерархии файловой системы. Свойство `parent` ссылается на экземпляр нового пути для каталога, содержащего данный путь, — значение, возвращаемое функцией `os.path.dirname()`. Свойство `parents` — итерируемый объект, предоставляю-

щий ссылки на родительские каталоги, которые соответствуют последовательному перемещению вверх по иерархическому дереву путей, пока не будет достигнут корневой каталог.

Листинг 6.19. pathlib_parents.py

```
import pathlib

p = pathlib.PurePosixPath('/usr/local/lib')

print('parent: {}'.format(p.parent))

print('\nhierarchy:')
for up in p.parents:
    print(up)
```

В этом примере выполняются итерации по свойству `parents` и выводятся значения элементов.

```
$ python3 pathlib_parents.py

parent: /usr/local

hierarchy:
/usr/local
/usr
/
```

Доступ к другим составляющим пути можно получить через свойства объекта пути. В свойстве `name` хранится последняя часть пути, располагающаяся вслед за последним разделителем каталогов (эквивалентная значению, возвращаемому вызовом `os.path.basename()`). В свойстве `suffix` хранится значение, располагающееся вслед за разделителем расширений, а в свойстве `stem` — часть имени, предшествующая суффиксу.

Листинг 6.20. pathlib_name.py

```
import pathlib

p = pathlib.PurePosixPath('./source/pathlib/pathlib_name.py')
print('path  : {}'.format(p))
print('name  : {}'.format(p.name))
print('suffix: {}'.format(p.suffix))
print('stem  : {}'.format(p.stem))
```

Несмотря на сходство значений свойств `suffix` и `stem` со значениями, возвращаемыми вызовом `os.path.splitext()`, они основываются только на значении свойства `name`, а не на полном пути.

```
$ python3 pathlib_name.py

path  : source/pathlib/pathlib_name.py
name  : pathlib_name.py
```

```
suffix: .py
stem  : pathlib_name
```

6.2.4. Создание полных путей

Экземпляры конкретного класса `Path` можно создавать из строковых аргументов, ссылающихся на имя (или потенциальное имя) файла, каталога или символической ссылки файловой системы. Кроме того, этот класс предоставляет несколько удобных методов для создания экземпляров с использованием таких известных расположений, как текущий рабочий каталог и домашний каталог пользователя, которые могут иметь различные значения.

Листинг 6.21. `pathlib_convenience.py`

```
import pathlib

home = pathlib.Path.home()
print('home: ', home)

cwd = pathlib.Path.cwd()
print('cwd : ', cwd)
```

Каждый из методов создает экземпляр, содержащий ссылку на абсолютный путь к соответствующему каталогу.

```
$ python3 pathlib_convenience.py

home:  /Users/dhellmann
cwd :  /Users/dhellmann/PyMOTW
```

6.2.5. Содержимое каталога

Доступ к списку содержимого каталога и именам содержащихся в нем файлов обеспечивают три метода. Метод `iterdir()` — это генератор, возвращающий новый экземпляр класса `Path` для каждого элемента, содержащегося в каталоге.

Листинг 6.22. `pathlib_iterdir.py`

```
import pathlib

p = pathlib.Path('.')

for f in p.iterdir():
    print(f)
```

Если экземпляр не ссылается на каталог, метод `iterdir()` возбуждает исключение `NotADirectoryError`.

```
$ python3 pathlib_iterdir.py

example_link
index.rst
```

```
pathlib_chmod.py
pathlib_convenience.py
pathlib_from_existing.py
pathlib_glob.py
pathlib_iterdir.py
pathlib_joinpath.py
pathlib_mkdir.py
pathlib_name.py
pathlib_operator.py
pathlib_ownership.py
pathlib_parents.py
pathlib_parts.py
pathlib_read_write.py
pathlib_resolve.py
pathlib_rglob.py
pathlib_rmdir.py
pathlib_stat.py
pathlib_symlink_to.py
pathlib_touch.py
pathlib_types.py
pathlib_unlink.py
```

Чтобы получить список лишь тех файлов, которые соответствуют заданному шаблону, используйте метод `glob()`.

Листинг 6.23. `pathlib_glob.py`

```
import pathlib

p = pathlib.Path('..')

for f in p.glob('*.rst'):
    print(f)
```

В этом примере отображаются все файлы в формате reStructuredText[2], находящиеся в родительском каталоге сценария.

```
$ python3 pathlib_glob.py

../about.rst
../algorithm_tools.rst
../book.rst
../compression.rst
../concurrency.rst
../cryptographic.rst
../data_structures.rst
../dates.rst
../dev_tools.rst
../email.rst
../file_access.rst
../frameworks.rst
../i18n.rst
```

[2] http://docutils.sourceforge.net/

```
../importing.rst
../index.rst
../internet_protocols.rst
../language.rst
../networking.rst
../numeric.rst
../persistence.rst
../porting_notes.rst
../runtime_services.rst
../text.rst
../third_party.rst
../unix.rst
```

Процессор `glob` поддерживает рекурсивное сканирование с использованием префикса шаблона `**` или посредством вызова метода `rglob()` вместо `glob()`.

Листинг 6.24. `pathlib_rglob.py`

```
import pathlib

p = pathlib.Path('..')

for f in p.rglob('pathlib_*.py'):
    print(f)
```

Поскольку в этом примере экземпляр инициализируется родительским каталогом, нахождение всех файлов примеров, соответствующих шаблону `pathlib_*.py`, требует проведения рекурсивного поиска.

```
$ python3 pathlib_rglob.py

../pathlib/pathlib_chmod.py
../pathlib/pathlib_convenience.py
../pathlib/pathlib_from_existing.py
../pathlib/pathlib_glob.py
../pathlib/pathlib_iterdir.py
../pathlib/pathlib_joinpath.py
../pathlib/pathlib_mkdir.py
../pathlib/pathlib_name.py
../pathlib/pathlib_operator.py
../pathlib/pathlib_ownership.py
../pathlib/pathlib_parents.py
../pathlib/pathlib_parts.py
../pathlib/pathlib_read_write.py
../pathlib/pathlib_resolve.py
../pathlib/pathlib_rglob.py
../pathlib/pathlib_rmdir.py
../pathlib/pathlib_stat.py
../pathlib/pathlib_symlink_to.py
../pathlib/pathlib_touch.py
../pathlib/pathlib_types.py
../pathlib/pathlib_unlink.py
```

6.2.6. Чтение и запись файлов

Каждый экземпляр `Path` включает методы для работы с содержимым файла, на который он ссылается. Для немедленного извлечения содержимого можно использовать метод `read_bytes()` или `read_text()`, а для записи в файл — метод `write_bytes()` или `write_text()`. Вместо передачи имени файла встроенной функции `open()` можно открыть файл и сохранить его дескриптор с помощью метода `open()`.

Листинг 6.25. `pathlib_read_write.py`

```
import pathlib

f = pathlib.Path('example.txt')

f.write_bytes('This is the content'.encode('utf-8'))

with f.open('r', encoding='utf-8') as handle:
    print('read from open(): {!r}'.format(handle.read()))

print('read_text(): {!r}'.format(f.read_text('utf-8')))
```

Вспомогательные методы выполняют определенную проверку типов перед открытием файла и записью данных в него, но во всем остальном они эквивалентны непосредственному выполнению операций.

```
$ python3 pathlib_read_write.py

read from open(): 'This is the content'
read_text(): 'This is the content'
```

6.2.7. Манипулирование каталогами и символическими ссылками

Пути, которые представляют несуществующие каталоги или символические ссылки, можно использовать для создания соответствующих объектов файловой системы.

Листинг 6.26. `pathlib_mkdir.py`

```
import pathlib

p = pathlib.Path('example_dir')

print('Creating {}'.format(p))
p.mkdir()
```

Если создаваемый путь уже существует, то метод `mkdir()` возбуждает исключение `FileExistsError`.

```
$ python3 pathlib_mkdir.py

Creating example_dir
```

```
$ python3 pathlib_mkdir.py

Creating example_dir
Traceback (most recent call last):
  File "pathlib_mkdir.py", line 16, in <module>
    p.mkdir()
  File ".../lib/python3.5/pathlib.py", line 1214, in mkdir
    self._accessor.mkdir(self, mode)
  File ".../lib/python3.5/pathlib.py", line 371, in wrapped
    return strfunc(str(pathobj), *args)
FileExistsError: [Errno 17] File exists: 'example_dir'
```

Для создания символических ссылок предназначен метод `symlink_to()`. Ссылка получает имя на основании значения пути и ссылается на имя, переданное методу `symlink_to()` в качестве аргумента.

Листинг 6.27. `pathlib_symlink_to.py`

```
import pathlib

p = pathlib.Path('example_link')

p.symlink_to('index.rst')

print(p)
print(p.resolve().name)
```

В этом примере создается символическая ссылка, разрешение которой с помощью метода `resolve()` позволяет определить целевой объект и вывести его имя.

```
$ python3 pathlib_symlink_to.py

example_link
index.rst
```

6.2.8. Типы файлов

Экземпляр `Path` включает несколько методов, позволяющих проверить тип файла, на который указывает путь. В следующем примере файлы, созданные в начале сценария, тестируются наряду с некоторыми специальными файлами устройств.

Листинг 6.28. `pathlib_types.py`

```
import itertools
import os
import pathlib

root = pathlib.Path('test_files')

# Удаление объектов, созданных во время предыдущих запусков
if root.exists():
    for f in root.iterdir():
```

```
        f.unlink()
else:
    root.mkdir()

# Создание тестовых файлов
(root / 'file').write_text(
    'This is a regular file', encoding='utf-8')
(root / 'symlink').symlink_to('file')
os.mkfifo(str(root / 'fifo'))

# Проверка типов файлов
to_scan = itertools.chain(
    root.iterdir(),
    [pathlib.Path('/dev/disk0'),
     pathlib.Path('/dev/console')],
)
hfmt = '{:18s}' + ('  {:>5}' * 6)
print(hfmt.format('Name', 'File', 'Dir', 'Link', 'FIFO', 'Block',
                  'Character'))
print()

fmt = '{:20s}  ' + ('{!r:>5}  ' * 6)
for f in to_scan:
    print(fmt.format(
        str(f),
        f.is_file(),
        f.is_dir(),
        f.is_symlink(),
        f.is_fifo(),
        f.is_block_device(),
        f.is_char_device(),
    ))
```

Все эти методы — `is_dir()`, `is_file()`, `is_symlink()`, `is_socket()`, `is_fifo()`, `is_block_device()` и `is_char_device()` — не имеют аргументов.

```
$ python3 pathlib_types.py
Name                 File    Dir   Link   FIFO  Block  Character
test_files/fifo       False  False  False   True  False  False
test_files/file        True  False  False  False  False  False
test_files/symlink     True  False   True  False  False  False
/dev/disk0            False  False  False  False   True  False
/dev/console          False  False  False  False  False   True
```

6.2.9. Свойства файлов

Подробную информацию о файле можно получить с помощью методов `stat()` и `lstat()` (для проверки статуса объекта, который может быть символической ссылкой). Эти методы предоставляют ту же информацию, что и функции `os.stat()` и `os.lstat()`.

Листинг 6.29. pathlib_stat.py

```
import pathlib
import sys
import time

if len(sys.argv) == 1:
    filename = __file__
else:
    filename = sys.argv[1]

p = pathlib.Path(filename)
stat_info = p.stat()

print('{}:'.format(filename))
print('  Size:', stat_info.st_size)
print('  Permissions:', oct(stat_info.st_mode))
print('  Owner:', stat_info.st_uid)
print('  Device:', stat_info.st_dev)
print('  Created      :', time.ctime(stat_info.st_ctime))
print('  Last modified:', time.ctime(stat_info.st_mtime))
print('  Last accessed:', time.ctime(stat_info.st_atime))
```

Вывод зависит от того, каким образом был установлен код примеров. Запустите сценарий `pathlib_stat.py` несколько раз, задавая в командной строке различные файлы.

```
$ python3 pathlib_stat.py

pathlib_stat.py:
  Size: 607
  Permissions: 0o100644
  Owner: 527
  Device: 16777218
  Created      : Thu Dec 29 12:25:25 2016
  Last modified: Thu Dec 29 12:25:25 2016
  Last accessed: Thu Dec 29 12:25:34 2016

$ python3 pathlib_stat.py index.rst

index.rst:
  Size: 19363
  Permissions: 0o100644
  Owner: 527
  Device: 16777218
  Created      : Thu Dec 29 11:27:58 2016
  Last modified: Thu Dec 29 11:27:58 2016
  Last accessed: Thu Dec 29 12:25:33 2016
```

Для получения информации о владельце файла используйте методы `owner()` и `group()`.

Листинг 6.30. pathlib_ownership.py

```
import pathlib

p = pathlib.Path(__file__)

print('{} is owned by {}/{}'.format(p, p.owner(), p.group()))
```

В то время как метод stat() возвращает числовые значения системных идентификаторов, указанные методы предоставляют имена, связанные с этими идентификаторами.

```
$ python3 pathlib_ownership.py

pathlib_ownership.py is owned by dhellmann/dhellmann
```

Метод touch() позволяет создавать файлы, а также обновлять информацию о времени последнего изменения существующих файлов и изменять права доступа к ним, работая подобно команде touch в Unix.

Листинг 6.31. pathlib_touch.py

```
import pathlib
import time

p = pathlib.Path('touched')
if p.exists():
    print('already exists')
else:
    print('creating new')

p.touch()
start = p.stat()

time.sleep(1)

p.touch()
end = p.stat()

print('Start:', time.ctime(start.st_mtime))
print('End :', time.ctime(end.st_mtime))
```

Многократное выполнение этого примера приводит к обновлению существующего файла при последующих запусках.

```
$ python3 pathlib_touch.py

creating new
Start: Thu Dec 29 12:25:34 2016
End : Thu Dec 29 12:25:35 2016

$ python3 pathlib_touch.py

already exists
```

```
Start: Thu Dec 29 12:25:35 2016
End : Thu Dec 29 12:25:36 2016
```

6.2.10. Права доступа

В Unix-подобных системах права доступа можно изменить с помощью метода `chmod()`, передав ему *режим доступа* в виде целого числа. Значения режима образуются посредством использования констант, определенных в модуле `stat`.

В следующем примере переключается бит разрешений, определяющий право пользователя на выполнение файла.

Листинг 6.32. `pathlib_chmod.py`

```
import os
import pathlib
import stat

# Создание исходного тестового файла
f = pathlib.Path('pathlib_chmod_example.txt')
if f.exists():
    f.unlink()
f.write_text('contents')

# Определение уже установленных разрешений с помощью модуля stat
existing_permissions = stat.S_IMODE(f.stat().st_mode)
print('Before: {:o}'.format(existing_permissions))

# Определение способа изменения разрешений
if not (existing_permissions & os.X_OK):
    print('Adding execute permission')
    new_permissions = existing_permissions | stat.S_IXUSR
else:
    print('Removing execute permission')
    # Использовать оператор xor для снятия бита, предоставляющего
    # пользователю право на выполнение файла
    new_permissions = existing_permissions ^ stat.S_IXUSR
%
# Внести изменение и отобразить новое значение режима
f.chmod(new_permissions)
after_permissions = stat.S_IMODE(f.stat().st_mode)
print('After: {:o}'.format(after_permissions))
```

Предполагается, что сценарий имеет разрешение на изменение режима доступа к файлу.

```
$ python3 pathlib_chmod.py

Before: 644
Adding execute permission
After: 744
```

6.2.11. Удаление объекта файловой системы

Для удаления объектов файловой системы существуют два метода, предназначенных для разных типов объектов. Если необходимо удалить пустой каталог, используйте метод `rmdir()`.

Листинг 6.33. `pathlib_rmdir.py`

```
import pathlib

p = pathlib.Path('example_dir')

print('Removing {}'.format(p))
p.rmdir()
```

Если указанного каталога не существует, то возбуждается исключение `FileNotFoundError`. Попытки удаления каталога, не являющегося пустым, также приводят к ошибке.

```
$ python3 pathlib_rmdir.py

Removing example_dir

$ python3 pathlib_rmdir.py

Removing example_dir
Traceback (most recent call last):
  File "pathlib_rmdir.py", line 16, in <module>
    p.rmdir()
  File ".../lib/python3.5/pathlib.py", line 1262, in rmdir
    self._accessor.rmdir(self)
  File ".../lib/python3.5/pathlib.py", line 371, in wrapped
    return strfunc(str(pathobj), *args)
FileNotFoundError: [Errno 2] No such file or directory:
'example_dir'
```

Для удаления файлов, символических ссылок и большинства других типов путей используйте метод `unlink()`.

Листинг 6.34. `pathlib_unlink.py`

```
import pathlib

p = pathlib.Path('touched')

p.touch()

print('exists before removing:', p.exists())

p.unlink()

print('exists after removing:', p.exists())
```

Чтобы удалить файл, символическую ссылку, сокет или другой объект файловой системы, пользователь должен обладать соответствующими правами.

```
$ python3 pathlib_unlink.py

exists before removing: True
exists after removing: False
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `pathlib`[3].
- `os.path` (раздел 6.1). Платформонезависимое манипулирование именами файлов.
- Управление правами доступа к объектам файловой системы (раздел 17.3.2). Обсуждение методов `os.stat()` и `os.lstat()`.
- `glob` (раздел 6.3). Использование шаблонов имен файлов в командной оболочке Unix.
- **PEP 428**[4]. Модель `pathlib`.

6.3. glob: шаблоны имен файлов

Несмотря на небольшие размеры API модуля `glob`, в нем сосредоточены большие возможности. Он полезен в ситуациях, когда программе требуется доступ к списку файлов с именами, соответствующими указанному шаблону. Чтобы получить список имен файлов, имеющих определенное расширение либо префикс или содержащих определенную подстроку в имени, достаточно использовать модуль `glob`, что избавит от необходимости написания собственного кода для просмотра содержимого каталогов.

Правила создания шаблонов для модуля `glob` отличаются от тех, которые используются для работы с регулярными выражениями модуля `re` (раздел 1.3). Они соответствуют правилам расширения путей, действующими в командной оболочке Unix, и подразумевают использование нескольких специальных символов для реализации двух групповых метасимволов и диапазонов символов.

Правила шаблонов применяются к сегментам имени файла (их действие заканчивается на разделителе каталогов, /). Пути, указанные в шаблоне, могут быть относительными или абсолютными. Подстановка значений вместо переменных оболочки и символов тильды (~) не осуществляется.

6.3.1. Данные для примеров

В приведенных ниже примерах предполагается, что в текущем рабочем каталоге содержатся следующие тестовые файлы.

```
$ python3 glob_maketestdata.py

dir
dir/file.txt
dir/file1.txt
```

[3] https://docs.python.org/3.5/library/pathlib.html

[4] www.python.org/dev/peps/pep-0428

```
dir/file2.txt
dir/filea.txt
dir/fileb.txt
dir/file?.txt
dir/file*.txt
dir/file[.txt
dir/subdir
dir/subdir/subfile.txt
```

В случае отсутствия этих файлов создайте их, запустив сценарий `glob_maketestdata.py`, прежде чем пытаться выполнять примеры.

6.3.2. Групповые метасимволы

Символу "звездочка" (*) соответствует любое количество символов в сегменте имени, в том числе ни одного, например `dir/*`.

Листинг 6.35. `glob_asterisk.py`

```
import glob
for name in sorted(glob.glob('dir/*')):
    print(name)
```

Этому шаблону соответствует имя любого пути (к файлу или каталогу) в каталоге `dir`, причем рекурсивный просмотр подкаталогов не выполняется. Данные, возвращенные методом `glob()`, не сортируются, поэтому в примерах их приходится дополнительно сортировать, чтобы упростить чтение результатов.

```
$ python3 glob_asterisk.py

dir/file*.txt
dir/file.txt
dir/file1.txt
dir/file2.txt
dir/file?.txt
dir/file[.txt
dir/filea.txt
dir/fileb.txt
dir/subdir
```

Чтобы получить список файлов, находящихся в определенном подкаталоге, следует включить имя этого подкаталога в шаблон.

Листинг 6.36. `glob_subdir.py`

```
import glob

print('Named explicitly:')
for name in sorted(glob.glob('dir/subdir/*')):
  print(' {}'.format(name))

print('Named with wildcard:')
for name in sorted(glob.glob('dir/*/*')):
  print(' {}'.format(name))
```

В первом из представленных здесь случаев имя подкаталога указано явно, во втором — с помощью группового метасимвола.

```
$ python3 glob_subdir.py

Named explicitly:
  dir/subdir/subfile.txt
Named with wildcard:
  dir/subdir/subfile.txt
```

В данном примере в обоих случаях результат оказался одним и тем же. При наличии еще одного подкаталога оба они соответствовали бы групповому метасимволу, и результирующий список включал бы имена файлов, содержащихся в обоих подкаталогах.

6.3.3. Метасимвол, соответствующий одиночному символу

Другим метасимволом является вопросительный знак (?). В имени ему соответствует любой одиночный символ, находящийся в данной позиции.

Листинг 6.37. glob_question.py

```
import glob

for name in sorted(glob.glob('dir/file?.txt')):
    print(name)
```

В этом примере отыскиваются все файлы с именами, начинающимися с подстроки file, за которой может следовать один произвольный символ, и имеющими расширение .txt.

```
$ python3 glob_question.py

dir/file*.txt
dir/file1.txt
dir/file2.txt
dir/file?.txt
dir/file[.txt
dir/filea.txt
dir/fileb.txt
```

6.3.4. Диапазоны символов

Для поиска соответствия одному из нескольких возможных символов используется диапазон символов ([a-z]). В следующем примере выполняется поиск всех файлов, имена которых содержат цифру перед расширением.

Листинг 6.38. glob_charrange.py

```
import glob

for name in sorted(glob.glob('dir/*[0-9].*')):
    print(name)
```

Диапазону [0-9] соответствует любая одиночная цифра. Диапазоны упорядочиваются на основе числовых кодов символов, а дефис обозначает непрерывную последовательность символов. Тот же самый диапазон можно было бы записать в виде [0123456789].

```
$ python3 glob_charrange.py

dir/file1.txt
dir/file2.txt
```

6.3.5. Экранирование метасимволов

Иногда возникает необходимость в поиске файлов, имена которых содержат специальные символы, используемые в шаблонах модуля glob. Метод escape() позволяет создать подходящий шаблон, в котором специальные символы *экранируются* таким образом, чтобы они не заменялись никакими другими значениями и не интерпретировались как специальные модулем glob.

Листинг 6.39. glob_escape.py

```
import glob

specials = '?*['

for char in specials:
    pattern = 'dir/*' + glob.escape(char) + '.txt'
    print('Searching for: {!r}'.format(pattern))
    for name in sorted(glob.glob(pattern)):
        print(name)
    print()
```

Каждый метасимвол экранируется посредством создания диапазона, содержащего один символ.

```
$ python3 glob_escape.py

Searching for: 'dir/*[?].txt'
dir/file?.txt

Searching for: 'dir/*[*].txt'
dir/file*.txt

Searching for: 'dir/*[[].txt'
dir/file[.txt
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю glob[5].
- *Pattern Matching Notation*[6]. Выдержка из описания нотации шаблонов в спецификации языка командной оболочки Open Group.

[5] https://docs.python.org/3.5/library/glob.html

[6] www.opengroup.org/onlinepubs/000095399/utilities/xcu_chap02.html#tag_02_13

- `fnmatch` (раздел 6.4). Реализация сопоставления шаблонов с именами файлов.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `glob` (раздел A.6.18).

6.4. `fnmatch`: шаблоны модуля `Glob` в стиле Unix

Модуль `fnmatch` используется для сравнения имен файлов с шаблонами модуля `glob` в соответствии с правилами, применяемыми в командных оболочках Unix.

6.4.1. Простое сопоставление

Метод `fnmatch()` сравнивает имя файла с шаблоном и возвращает булево значение, указывающее на то, соответствует ли имя файла шаблону. Чувствительность сравнения к регистру определяется чувствительностью к регистру файловой системы.

Листинг 6.40. `fnmatch_fnmatch.py`

```
import fnmatch
import os

pattern = 'fnmatch_*.py'
print('Pattern :', pattern)
print()
files = os.listdir('.')
for name in files:
    print('Filename: {:<25} {}'.format(
        name, fnmatch.fnmatch(name, pattern)))
```

В этом примере шаблону соответствуют все файлы, имена которых начинаются с подстроки `'fnmatch_'` и заканчиваются подстрокой `'.py'`.

```
$ python3 fnmatch_fnmatch.py

Pattern : fnmatch_*.py

Filename: fnmatch_filter.py         True
Filename: fnmatch_fnmatch.py        True
Filename: fnmatch_fnmatchcase.py    True
Filename: index.rst                 False
```

Чтобы выполнить сравнение, чувствительное к регистру, независимо от файловой или операционной системы, следует использовать метод `fnmatchcase()`.

Листинг 6.41. `fnmatch_fnmatchcase.py`

```
import fnmatch
import os

pattern = 'FNMATCH_*.PY'
print('Pattern :', pattern)
print()
```

```
files = os.listdir('.')

for name in files:
    print('Filename: {:<25} {}'.format(
        name, fnmatch.fnmatchcase(name, pattern)))
```

Поскольку в операционной системе OS X, в которой тестировалась эта программа, используется чувствительная к регистру файловая система, ни один из файлов не соответствует измененному шаблону.

```
$ python3 fnmatch_fnmatchcase.py

Pattern : FNMATCH_*.PY

Filename: fnmatch_filter.py         False
Filename: fnmatch_fnmatch.py        False
Filename: fnmatch_fnmatchcase.py    False
Filename: fnmatch_translate.py      False
Filename: index.rst                 False
```

6.4.2. Фильтрация

Для тестирования последовательности имен файлов используется метод `filter()`, который возвращает список имен, соответствующих шаблону.

Листинг 6.42. `fnmatch_filter.py`

```
import fnmatch
import os
import pprint

pattern = 'fnmatch_*.py'
print('Pattern :', pattern)

files = os.listdir('.')

print('\nFiles :')
pprint.pprint(files)

print('\nMatches :')
pprint.pprint(fnmatch.filter(files, pattern))
```

В этом примере метод `filter()` возвращает список имен файлов, которые находятся в каталоге примеров, относящемся к данному разделу.

```
$ python3 fnmatch_filter.py

Pattern : fnmatch_*.py

Files   :
['fnmatch_filter.py',
 'fnmatch_fnmatch.py',
 'fnmatch_fnmatchcase.py',
```

```
 'fnmatch_translate.py',
 'index.rst']

Matches :
['fnmatch_filter.py',
 'fnmatch_fnmatch.py',
 'fnmatch_fnmatchcase.py',
 'fnmatch_translate.py']
```

6.4.3. Трансляция шаблонов

Внутренний механизм `fnmatch` преобразует шаблоны `glob` в регулярные выражения и использует модуль `re` (раздел 1.3) для сравнения имени и шаблона. Общедоступный метод API для преобразования шаблонов `glob` в регулярные выражения обеспечивает функция `translate()`.

Листинг 6.43. `fnmatch_translate.py`

```
import fnmatch

pattern = 'fnmatch_*.py'
print('Pattern :', pattern)
print('Regex   :', fnmatch.translate(pattern))
```

Для получения действительного регулярного выражения некоторые метасимволы экранируются.

```
$ python3 fnmatch_translate.py

Pattern : fnmatch_*.py
Regex   : fnmatch_.*\.py\Z(?ms)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `fnmatch`[7].
- `glob` (раздел 6.3). Модуль `glob` использует механизм `fnmatch` совместно с функцией `os.listdir()` для получения списков файлов и каталогов, соответствующих заданным шаблонам.
- `re` (раздел 1.3). Поиск соответствий шаблону с помощью регулярных выражений.

6.5. `linecache`: эффективное чтение файлов

Модуль `linecache` используется другими компонентами стандартной библиотеки Python при работе с исходными файлами на языке Python. Реализация кеша сохраняет содержимое файлов, разобранное на отдельные строки, в памяти. Соответствующий API возвращает затребованные строки в виде индексированного списка, экономя время при повторных попытках чтения файлов и поиске нужных строк. Этот модуль особенно полезен в тех случаях, когда приходится

[7] https://docs.python.org/3.5/library/fnmatch.html

просматривать множество строк одного и того же файла, например с целью получения трассировочной информации для вывода отчета об ошибке.

6.5.1. Тестовые данные

В приведенных ниже примерах в качестве образца входной информации используется текст, полученный с помощью генератора Lorem Ipsum.

Листинг 6.44. `linecache_data.py`

```
import os
import tempfile

lorem = '''Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Vivamus eget elit. In posuere mi non
risus. Mauris id quam posuere lectus sollicitudin
varius. Praesent at mi. Nunc eu velit. Sed augue massa,
fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur
eros pede, egestas at, ultricies ac, apellentesque eu,
tellus.

Sed sed odio sed mi luctus mollis. Integer et nulla ac augue
convallis accumsan. Ut felis. Donec lectus sapien, elementum
nec, condimentum ac, interdum non, tellus. Aenean viverra,
mauris vehicula semper porttitor, ipsum odio consectetuer
lorem, ac imperdiet eros odio a sapien. Nulla mauris tellus,
aliquam non, egestas a, nonummy et, erat. Vivamus sagittis
porttitor eros.'''


def make_tempfile():
    fd, temp_file_name = tempfile.mkstemp()
    os.close(fd)
    with open(temp_file_name, 'wt') as f:
        f.write(lorem)
    return temp_file_name

def cleanup(filename):
    os.unlink(filename)
```

6.5.2. Чтение конкретных строк

Нумерация строк, которые читаются с помощью модуля `linecache`, начинается с 1, хотя обычно отсчет индексов элементов массива начинается с 0.

Листинг 6.45. `linecache_getline.py`

```
import linecache
from linecache_data import *

filename = make_tempfile()

# Выбор одной и той же строки из исходного файла и кеша
```

```
# (имейте в виду, что linecache нумерует строки, начиная с 1)
print('SOURCE:')
print('{!r}'.format(lorem.split('\n')[4]))
print()
print('CACHE:')
print('{!r}'.format(linecache.getline(filename, 5)))

cleanup(filename)
```

Каждая возвращенная строка включает символ перевода строки.

```
$ python3 linecache_getline.py

SOURCE:
'fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur'

CACHE:
'fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur\n'
```

6.5.3. Обработка пустых строк

Возвращаемое значение всегда содержит завершающий символ перевода строки. Таким образом, в случае пустой строки этот символ является единственным в возвращенном значении.

Листинг 6.46. `linecache_empty_line.py`

```
import linecache
from linecache_data import *

filename = make_tempfile()

# Пустые строки включают символ перевода строки
print('BLANK : {!r}'.format(linecache.getline(filename, 8)))

cleanup(filename)
```

Строка 8 во входном файле не содержит текста.

```
$ python3 linecache_empty_line.py

BLANK : '\n'
```

6.5.4. Обработка ошибок

Если запрошенный номер строки превышает общее количество строк, содержащихся в файле, функция `getline()` возвращает пустую строку.

Листинг 6.47. `linecache_out_of_range.py`

```
import linecache
from linecache_data import *
```

```
filename = make_tempfile()

# Кеш всегда возвращает строку, используя пустую строку для
# указания того, что запрошенная строка не существует
not_there = linecache.getline(filename, 500)
print('NOT THERE: {!r} includes {} characters'.format(
    not_there, len(not_there)))

cleanup(filename)
```

Входной файл содержит только 15 строк, поэтому запрашивать строку 500 — это все равно что пытаться выполнить чтение за пределами файла.

```
$ python3 linecache_out_of_range.py

NOT THERE: '' includes 0 characters
```

Точно так же обрабатываются попытки чтения из файла, которого не существует.

Листинг 6.48. linecache_missing_file.py

```
import linecache

# Если модулю linecache не удается найти файл, то ошибка скрывается
no_such_file = linecache.getline(
    'this_file_does_not_exist.txt', 1,
)
print('NO FILE: {!r}'.format(no_such_file))
```

Модуль linecache никогда не возбуждает исключений при чтении данных вызывающим кодом.

```
$ python3 linecache_missing_file.py

NO FILE: ''
```

6.5.5. Чтение исходных файлов Python

Поскольку модуль linecache интенсивно используется для получения трассировочной информации, одной из его особенностей является способность находить исходные модули Python в путях импорта по базовому имени модуля.

Листинг 6.49. linecache_path_search.py

```
import linecache
import os

# Поиск модуля linecache с использованием пути,
# указанного в sys.path
module_line = linecache.getline('linecache.py', 3)
print('MODULE:')
print(repr(module_line))
# Непосредственный поиск модуля linecache
```

```
file_src = linecache.__file__
if file_src.endswith('.pyc'):
    file_src = file_src[:-1]
print('\nFILE:')
with open(file_src, 'r') as f:
    file_line = f.readlines()[2]
print(repr(file_line))
```

Если именованный модуль не удается найти в текущем каталоге, то код модуля `linecache`, ответственный за работу с кешем, ищет его в списке путей `sys.path`. В этом примере выполняется поиск модуля `linecache.py`. Ввиду отсутствия копии модуля в текущем каталоге код берется из стандартной библиотеки.

```
$ python3 linecache_path_search.py

MODULE:
'This is intended to read lines from modules imported -- hence
if a filename\n'

FILE:
'This is intended to read lines from modules imported -- hence
if a filename\n'
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `linecache`[8].

6.6. `tempfile`: временные объекты файловой системы

Создание временных файлов с надежными уникальными именами, затрудняющими определение назначения файлов злоумышленниками с целью взлома приложения или кражи данных, — непростая задача. Модуль `tempfile` предоставляет несколько функций, обеспечивающих безопасное создание временных файлов системных ресурсов. Функция `TemporaryFile()` открывает и возвращает неименованный файл, функция `NamedTemporaryFile()` открывает и возвращает именованный файл, функция `SpooledTemporaryFile()` сохраняет содержимое в памяти перед записью на диск, тогда как функция `TemporaryDirectory()` — это менеджер контекста, удаляющий каталог, когда контекст закрывается.

6.6.1. Временные файлы

Приложения, нуждающиеся во временных файлах для сохранения промежуточных данных в условиях, когда не требуется совместное использование файла другими приложениями, должны создавать такие файлы с помощью функции `TemporaryFile()`. На тех платформах, где это возможно, соответствующая запись в каталоге уничтожается сразу же после создания файла. Как следствие, другие программы не могут найти или открыть этот файл ввиду отсутствия ссылки

[8] https://docs.python.org/3.5/library/linecache.html

на него в таблице файловой системы. Созданный функцией `TemporaryFile()` файл автоматически уничтожается при его закрытии как в случае использования метода `close()`, так и совместного использования API менеджера контекста с инструкцией `with`.

Листинг 6.50. `tempfile_TemporaryFile.py`

```
import os
import tempfile

print('Building a filename with PID:')
filename = '/tmp/guess_my_name.{}.txt'.format(os.getpid())
with open(filename, 'w+b') as temp:
    print('temp:')
    print('  {!r}'.format(temp))
    print('temp.name:')
    print('  {!r}'.format(temp.name))

# Самостоятельное удаление временного файла
os.remove(filename)

print()
print('TemporaryFile:')
with tempfile.TemporaryFile() as temp:
    print('temp:')
    print('  {!r}'.format(temp))
    print('temp.name:')
    print('  {!r}'.format(temp.name))

# Автоматическое уничтожение файла
```

Этот пример иллюстрирует различие в способах создания временного файла с помощью общего шаблона для конструирования имен и с помощью функции `TemporaryFile()`. У файла, возвращенного функцией `TemporaryFile()`, нет имени.

```
$ python3 tempfile_TemporaryFile.py

Building a filename with PID:
temp:
  <_io.BufferedRandom name='/tmp/guess_my_name.12151.txt'>
temp.name:
  '/tmp/guess_my_name.12151.txt'

TemporaryFile:
temp:
  <_io.BufferedRandom name=4>
temp.name:
  4
```

По умолчанию дескриптор файла создается с использованием режима `'w+b'`, что обеспечивает его согласованное поведение на всех платформах и позволяет

вызывающему коду использовать его для выполнения файловых операций чтения и записи.

Листинг 6.51. tempfile_TemporaryFile_binary.py

```
import os
import tempfile

with tempfile.TemporaryFile() as temp:
    temp.write(b'Some data')

    temp.seek(0)
    print(temp.read())
```

После выполнения записи файловый дескриптор следует "перемотать назад" с помощью метода `seek()`, чтобы обеспечить возможность последующего чтения данных.

```
$ python3 tempfile_TemporaryFile_binary.py

b'Some data'
```

Чтобы открыть файл в текстовом режиме, при его создании следует задать аргумент `mode` равным `'w+t'`.

Листинг 6.52. tempfile_TemporaryFile_text.py

```
import tempfile

with tempfile.TemporaryFile(mode='w+t') as f:
    f.writelines(['first\n', 'second\n'])

    f.seek(0)
    for line in f:
        print(line.rstrip())
```

В этом случае данные обрабатываются как текстовые.

```
$ python3 tempfile_TemporaryFile_text.py

first
second
```

6.6.2. Именованные файлы

В некоторых случаях наличие именованного временного файла играет важную роль. В случае приложений, охватывающих несколько процессов или даже хостов, именование файла является простейшим способом организации его передачи между различными частями приложения. Функция `NamedTemporaryFile()` создает файл без разрыва его связи с файловой системой, поэтому он сохраняет свое имя (доступное через атрибут `name`).

Листинг 6.53. tempfile_NamedTemporaryFile.py

```
import os
import pathlib
import tempfile

with tempfile.NamedTemporaryFile() as temp:
    print('temp:')
    print(' {!r}'.format(temp))
    print('temp.name:')
    print(' {!r}'.format(temp.name))

    f = pathlib.Path(temp.name)

print('Exists after close:', f.exists())
```

После закрытия дескриптора этот файл удаляется.

```
$ python3 tempfile_NamedTemporaryFile.py

temp:
  <tempfile._TemporaryFileWrapper object at 0x1011b2d30>
temp.name:
  '/var/folders/5q/8gk0wq888xlggz008k8dr7180000hg/T/tmps4qh5zde'
Exists after close: False
```

6.6.3. Буферизуемые файлы

В случае временных файлов, содержащих относительно небольшие объемы данных, более эффективным будет использование функции `SpooledTemporaryFile()`, поскольку она сохраняет содержимое файла в буферных объектах `io.BytesIO` или `io.StringIO`, где оно будет храниться, пока объем данных не превысит заданного порогового значения. По достижении этого значения данные записываются на диск, а буфер заменяется обычным объектом `TemporaryFile`.

Листинг 6.54. tempfile_SpooledTemporaryFile.py

```
import tempfile

with tempfile.SpooledTemporaryFile(max_size=100,
                                   mode='w+t',
                                   encoding='utf-8') as temp:
    print('temp: {!r}'.format(temp))

    for i in range(3):
        temp.write('This line is repeated over and over.\n')
        print(temp._rolled, temp._file)
```

В этом примере для определения того момента, когда данные сбрасываются на диск, используются закрытые атрибуты объекта `SpooledTemporaryFile`. Необходимость в этом возникает лишь изредка, за исключением случаев, когда требуется настройка размера буфера.

```
$ python3 tempfile_SpooledTemporaryFile.py

temp: <tempfile.SpooledTemporaryFile object at 0x1007b2c88>
False <_io.StringIO object at 0x1007a3d38>
False <_io.StringIO object at 0x1007a3d38>
True <_io.TextIOWrapper name=4 mode='w+t' encoding='utf-8'>
```

Для принудительного сброса содержимого буфера и записи его на диск следует вызвать метод `rollover()` или `fileno()`.

Листинг 6.55. tempfile_SpooledTemporaryFile_explicit.py

```
import tempfile

with tempfile.SpooledTemporaryFile(max_size=1000,
                                   mode='w+t',
                                   encoding='utf-8') as temp:
    print('temp: {!r}'.format(temp))

    for i in range(3):
        temp.write('This line is repeated over and over.\n')
        print(temp._rolled, temp._file)
    print('rolling over')
    temp.rollover()
    print(temp._rolled, temp._file)
```

Поскольку размер буфера в этом примере намного превышает объем имеющихся данных, файл не создается до тех пор, пока не вызывается метод `rollover()`.

```
$ python3 tempfile_SpooledTemporaryFile_explicit.py

temp: <tempfile.SpooledTemporaryFile object at 0x1007b2c88>
False <_io.StringIO object at 0x1007a3d38>
False <_io.StringIO object at 0x1007a3d38>
False <_io.StringIO object at 0x1007a3d38>
rolling over
True <_io.TextIOWrapper name=4 mode='w+t' encoding='utf-8'>
```

6.6.4. Временные каталоги

Если возникает необходимость в использовании нескольких временных файлов, может оказаться удобным создать для них общий временный каталог и открывать все файлы в нем.

Листинг 6.56. tempfile_TemporaryDirectory.py

```
import pathlib
import tempfile

with tempfile.TemporaryDirectory() as directory_name:
    the_dir = pathlib.Path(directory_name)
    print(the_dir)
```

```
    a_file = the_dir / 'a_file.txt'
    a_file.write_text('This file is deleted.')

print('Directory exists after?', the_dir.exists())
print('Contents after:', list(the_dir.glob('*')))
```

Менеджер контекста создает имя каталога, которое затем может быть использовано в блоке контекста для создания других имен файлов.

```
$ python3 tempfile_TemporaryDirectory.py

/var/folders/5q/8gk0wq888xlggz008k8dr7180000hg/T/tmp_urhiioj
Directory exists after? False
Contents after: []
```

6.6.5. Предсказуемые имена

Несмотря на то что включение в имя предсказуемых частей делает его менее безопасным, это позволяет отслеживать файл и изучать его в целях отладки. Каждая из описанных до сих пор функций имеет три аргумента, позволяющих в определенной степени управлять именами временных файлов. Имена генерируются с использованием следующей формулы:

```
dir + prefix + random + suffix
```

Все значения, за исключением `random`, могут передаваться в качестве аргументов функциям, с помощью которых создаются файлы или каталоги.

Листинг 6.57. `tempfile_NamedTemporaryFile_args.py`

```
import tempfile

with tempfile.NamedTemporaryFile(suffix='_suffix',
                                 prefix='prefix_',
                                 dir='/tmp') as temp:
    print('temp:')
    print('  ', temp)
    print('temp.name:')
    print('  ', temp.name)
```

Имя файла создается посредством объединения аргументов `prefix` и `suffix` с символьной строкой `random`, тогда как аргумент `dir` используется в том виде, как он указан, и задает расположение нового файла.

```
$ python3 tempfile_NamedTemporaryFile_args.py

temp:
   <tempfile._TemporaryFileWrapper object at 0x1018b2d68>
temp.name:
   /tmp/prefix_q6wd5czl_suffix
```

6.6.6. Расположение временного файла

Если место хранения временных файлов не задано явно с помощью аргумента dir, то путь, используемый для временных файлов, зависит от текущей платформы и настроек. Модуль tempfile включает две функции, предназначенные для опроса и установки используемых настроек во время выполнения.

Листинг 6.58. tempfile_settings.py

```
import tempfile

print('gettempdir():', tempfile.gettempdir())
print('gettempprefix():', tempfile.gettempprefix())
```

Функция gettempdir() возвращает каталог по умолчанию, который будет использоваться для сохранения всех временных файлов, а функция gettempprefix() возвращает строковый префикс для имен нового файла и каталога.

```
$ python3 tempfile_settings.py

gettempdir(): /var/folders/5q/8gk0wq888xlggz008k8dr7180000hg/T
gettempprefix(): tmp
```

Значение, возвращаемое функцией gettempdir(), устанавливается на основании простого просмотра списка доступных расположений в поиске первого подходящего места, в котором текущий процесс может создать файл. Список подходящих мест просматривается в следующей очередности.

1. Переменная среды TMPDIR.
2. Переменная среды TEMP.
3. Переменная среды TMP.
4. Резервный вариант, зависящий от платформы. (Windows использует первый из доступных каталогов C:\temp, C:\tmp, \temp или \tmp; другие платформы используют /tmp, /var/tmp или /usr/tmp.)
5. Если никакой другой каталог найти не удается, то используется текущий рабочий каталог.

Листинг 6.59. tempfile_tempdir.py

```
import tempfile

tempfile.tempdir = '/I/changed/this/path'
print('gettempdir():', tempfile.gettempdir())
```

Программы, нуждающиеся в использовании одного глобального расположения для всех временных файлов без использования любой из перечисленных переменных среды, должны установить атрибут tempfile.tempdir непосредственным присваиванием ему нужного значения.

```
$ python3 tempfile_tempdir.py

gettempdir(): /I/changed/this/path
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `tempfile`[9].
- `random` (см. раздел 5.3). Генератор псевдослучайных чисел, используемый для вставки случайных значений в имена временных файлов.

6.7. `shutil`: высокоуровневые файловые операции

Модуль `shutil` позволяет выполнять высокоуровневые файловые операции, такие как копирование и архивирование.

6.7.1. Копирование файлов

Функция `copyfile()` копирует содержимое источника в указанное место. В случае отсутствия разрешения на выполнение записи в файл назначения возбуждается исключение `IOError`.

Листинг 6.60. `shutil_copyfile.py`

```
import glob
import shutil

print('BEFORE:', glob.glob('shutil_copyfile.*'))

shutil.copyfile('shutil_copyfile.py', 'shutil_copyfile.py.copy')

print('AFTER:', glob.glob('shutil_copyfile.*'))
```

Поскольку функция `copyfile()` открывает входной файл для чтения, независимо от его типа, ее нельзя использовать для копирования специальных файлов (таких, как узлы устройств Unix) в новые специальные файлы.

```
$ python3 shutil_copyfile.py

BEFORE: ['shutil_copyfile.py']
AFTER: ['shutil_copyfile.py', 'shutil_copyfile.py.copy']
```

Реализация функции `copyfile()` использует низкоуровневую функцию `copyfileobj()`. В то время как аргументами функции `copyfile()` являются имена файлов, аргументами функции `copyfileobj()` являются дескрипторы файлов. Необязательный третий аргумент — размер буфера, используемого для чтения блоками.

Листинг 6.61. `shutil_copyfileobj.py`

```
import io
import os
import shutil
import sys
```

[9] https://docs.python.org/3.5/library/tempfile.html

```
class VerboseStringIO(io.StringIO):

    def read(self, n=-1):
        next = io.StringIO.read(self, n)
        print('read({}) got {} bytes'.format(n, len(next)))
        return next


lorem_ipsum = '''Lorem ipsum dolor sit amet, consectetuer
adipiscing elit.  Vestibulum aliquam mollis dolor. Donec
vulputate nunc ut diam. Ut rutrum mi vel sem. Vestibulum
ante ipsum.'''

print('Default:')
input = VerboseStringIO(lorem_ipsum)
output = io.StringIO()
shutil.copyfileobj(input, output)

print()

print('All at once:')
input = VerboseStringIO(lorem_ipsum)
output = io.StringIO()
shutil.copyfileobj(input, output, -1)

print()

print('Blocks of 256:')
input = VerboseStringIO(lorem_ipsum)
output = io.StringIO()
shutil.copyfileobj(input, output, 256)
```

По умолчанию данные читаются крупными блоками. Аргумент `n = -1` означает чтение всех входных данных; любое другое положительное значение устанавливает конкретный размер блока. В этом примере для сравнения используется несколько разных размеров блока.

```
$ python3 shutil_copyfileobj.py

Default:
read(16384) got 166 bytes
read(16384) got 0 bytes

All at once:
read(-1) got 166 bytes
read(-1) got 0 bytes

Blocks of 256:
read(256) got 166 bytes
read(256) got 0 bytes
```

Функция `copy()` интерпретирует имя выходного файла точно так же, как инструмент `cp` командной строки Unix. Если имя объекта назначения ссылается не

на файл, а на каталог, то новый файл создается в этом каталоге с использованием базового имени исходного файла.

Листинг 6.62. shutil_copy.py

```
import glob
import os
import shutil

os.mkdir('example')
print('BEFORE:', glob.glob('example/*'))

shutil.copy('shutil_copy.py', 'example')

print('AFTER :', glob.glob('example/*'))
```

Права на доступ к файлу копируются вместе с содержимым.

```
$ python3 shutil_copy.py

BEFORE: []
AFTER : ['example/shutil_copy.py']
```

Функция `copy2()` работает аналогично функции `copy()`, но копирует также время последнего доступа и последнего изменения файла.

Листинг 6.63. shutil_copy2.py

```
import os
import shutil


import time
    def show_file_info(filename):
    stat_info = os.stat(filename)
    print(' Mode :', oct(stat_info.st_mode))
    print(' Created :', time.ctime(stat_info.st_ctime))
    print(' Accessed:', time.ctime(stat_info.st_atime))
    print(' Modified:', time.ctime(stat_info.st_mtime))


os.mkdir('example')
print('SOURCE:')
show_file_info('shutil_copy2.py')

shutil.copy2('shutil_copy2.py', 'example')

print('DEST:')
show_file_info('example/shutil_copy2.py')
```

Новый файл имеет те же характеристики, что и исходная версия.

```
$ python3 shutil_copy2.py

SOURCE:
  Mode    : 0o100644
```

```
  Created : Wed Dec 28 19:03:12 2016
  Accessed: Wed Dec 28 19:03:49 2016
  Modified: Wed Dec 28 19:03:12 2016
DEST:
  Mode    : 0o100644
  Created : Wed Dec 28 19:03:49 2016
  Accessed: Wed Dec 28 19:03:49 2016
  Modified: Wed Dec 28 19:03:12 2016
```

6.7.2. Копирование метаданных файла

По умолчанию, если файл создается в Unix, права доступа к нему назначаются на основании маски режима создания файлов (umask) текущего пользователя. Для копирования битов разрешений из одного файла в другой используется функция `copymode()`.

Листинг 6.64. `shutil_copymode.py`

```
import os
import shutil
import subprocess

with open('file_to_change.txt', 'wt') as f:
    f.write('content')
os.chmod('file_to_change.txt', 0o444)

print('BEFORE:', oct(os.stat('file_to_change.txt').st_mode))

shutil.copymode('shutil_copymode.py', 'file_to_change.txt')

print('AFTER :', oct(os.stat('file_to_change.txt').st_mode))
```

В этом примере сценарий сначала создает файл, подлежащий изменению, а затем использует функцию `copymode()` для копирования разрешений файла сценария в файл примера.

```
$ python3 shutil_copymode.py

BEFORE: 0o100444
AFTER : 0o100644
```

Для копирования других метаданных файла используется функция `copystat()`.

Листинг 6.65. `shutil_copystat.py`

```
import os
import shutil
import time


def show_file_info(filename):
    stat_info = os.stat(filename)
    print(' Mode :', oct(stat_info.st_mode))
```

```
    print(' Created :', time.ctime(stat_info.st_ctime))
    print(' Accessed:', time.ctime(stat_info.st_atime))
    print(' Modified:', time.ctime(stat_info.st_mtime))


with open('file_to_change.txt', 'wt') as f:
    f.write('content')
os.chmod('file_to_change.txt', 0o444)

print('BEFORE:')
show_file_info('file_to_change.txt')

shutil.copystat('shutil_copystat.py', 'file_to_change.txt')

print('AFTER:')
show_file_info('file_to_change.txt')
```

Функция `copystat()` копирует лишь права доступа и даты, связанные с файлом.

```
$ python3 shutil_copystat.py

BEFORE:
  Mode    : 0o100444
  Created : Wed Dec 28 19:03:49 2016
  Accessed: Wed Dec 28 19:03:49 2016
  Modified: Wed Dec 28 19:03:49 2016
AFTER:
  Mode    : 0o100644
  Created : Wed Dec 28 19:03:49 2016
  Accessed: Wed Dec 28 19:03:49 2016
  Modified: Wed Dec 28 19:03:46 2016
```

6.7.3. Работа с деревьями каталогов

Модуль `shutil` включает три функции для работы с деревьями каталогов. Чтобы скопировать каталог из одного места в другое, используйте функцию `copytree()`, которая рекурсивно копирует исходное дерево каталогов в каталог назначения. Каталог назначения не должен существовать до вызова функции.

Листинг 6.66. `shutil_copytree.py`

```
import glob
import pprint
import shutil

print('BEFORE:')
pprint.pprint(glob.glob('/tmp/example/*'))

shutil.copytree('../shutil', '/tmp/example')

print('\nAFTER:')
pprint.pprint(glob.glob('/tmp/example/*'))
```

Аргумент symlinks управляет тем, в каком виде копируются символические ссылки: в виде ссылок или файлов. По умолчанию в новое дерево копируется содержимое файлов, на которые указывают символические ссылки. Если значение аргумента symlinks задано равным True, то в каталоге назначения создаются символические ссылки.

```
$ python3 shutil_copytree.py

BEFORE:
[]
AFTER:
['/tmp/example/example',
'/tmp/example/example.out',
'/tmp/example/file_to_change.txt',
'/tmp/example/index.rst',
'/tmp/example/shutil_copy.py',
'/tmp/example/shutil_copy2.py',
'/tmp/example/shutil_copyfile.py',
'/tmp/example/shutil_copyfile.py.copy',
'/tmp/example/shutil_copyfileobj.py',
'/tmp/example/shutil_copymode.py',
'/tmp/example/shutil_copystat.py',
'/tmp/example/shutil_copytree.py',
'/tmp/example/shutil_copytree_verbose.py',
'/tmp/example/shutil_disk_usage.py',
'/tmp/example/shutil_get_archive_formats.py',
'/tmp/example/shutil_get_unpack_formats.py',
'/tmp/example/shutil_make_archive.py',
'/tmp/example/shutil_move.py',
'/tmp/example/shutil_rmtree.py',
'/tmp/example/shutil_unpack_archive.py',
'/tmp/example/shutil_which.py',
'/tmp/example/shutil_which_regular_file.py']
```

Поведением функции copytree() управляют два вызываемых объекта, передаваемых ей в качестве аргументов. Аргумент ignore вызывается с передачей ему имени каждого копируемого каталога или подкаталога вместе со списком содержимого. Функция должна вернуть список элементов, которые будут игнорироваться при копировании. Аргумент copy_function вызывается для фактического копирования файла.

Листинг 6.67. shutil_copytree_verbose.py

```
import glob
import pprint
import shutil


def verbose_copy(src, dst):
    print('copying\n {!r}\n to {!r}'.format(src, dst))
    return shutil.copy2(src, dst)
```

```
print('BEFORE:')
pprint.pprint(glob.glob('/tmp/example/*'))
print()

shutil.copytree(
    '../shutil', '/tmp/example',
    copy_function=verbose_copy,
    ignore=shutil.ignore_patterns('*.py'),
)

print('\nAFTER:')
pprint.pprint(glob.glob('/tmp/example/*'))
```

В этом примере функция-фабрика `ignore_patterns()` создает вызываемый объект `ignore`, который используется для исключения копирования исходных файлов Python. Функция `verbose_copy()` сначала выводит имена копируемых файлов, а затем использует функцию `copy2()` — функцию копирования по умолчанию — для создания копий.

```
$ python3 shutil_copytree_verbose.py

BEFORE:
[]

copying
 '../shutil/example.out'
 to '/tmp/example/example.out'
copying
 '../shutil/file_to_change.txt'
 to '/tmp/example/file_to_change.txt'
copying
 '../shutil/index.rst'
 to '/tmp/example/index.rst'

AFTER:
['/tmp/example/example',
 '/tmp/example/example.out',
 '/tmp/example/file_to_change.txt',
 '/tmp/example/index.rst']
```

Для удаления каталога и его содержимого используйте функцию `rmtree()`.

Листинг 6.68. `shutil_rmtree.py`

```
import glob
import pprint
import shutil

print('BEFORE:')
pprint.pprint(glob.glob('/tmp/example/*'))

shutil.rmtree('/tmp/example')
```

```
print('\nAFTER:')
pprint.pprint(glob.glob('/tmp/example/*'))
```

По умолчанию ошибки возбуждают исключения, но ошибки можно игнорировать, задав значение второго аргумента равным `True`. В третьем аргументе можно передать функцию, которая будет обрабатывать ошибки.

```
$ python3 shutil_rmtree.py

BEFORE:
['/tmp/example/example',
 '/tmp/example/example.out',
 '/tmp/example/file_to_change.txt',
 '/tmp/example/index.rst']

AFTER:
 []
```

Для перемещения файла или каталога из одного места в другое используйте функцию `move()`.

Листинг 6.69. `shutil_move.py`

```
import glob
import shutil

with open('example.txt', 'wt') as f:
    f.write('contents')

print('BEFORE: ', glob.glob('example*'))

shutil.move('example.txt', 'example.out')

print('AFTER : ', glob.glob('example*'))
```

Семантика этой команды аналогична семантике команды `mv` в Unix. Если оба аргумента являются файлами, то имя первого файла заменяется именем второго. Если второй аргумент является каталогом, то в нем создается копия исходного файла, после чего исходный файл удаляется.

```
$ python3 shutil_move.py

BEFORE: ['example.txt']
AFTER : ['example.out']
```

6.7.4. Поиск файлов

Функция `which()` находит полный путь к файлу с указанным именем, просматривая пути поиска. Типичным способом применения этой функции является поиск файла исполняемой программы в соответствии с путями поиска, используемыми командной оболочкой, которые содержатся в переменной среды `PATH`.

Листинг 6.70. shutil_which.py

```
import shutil

print(shutil.which('virtualenv'))
print(shutil.which('tox'))
print(shutil.which('no-such-program'))
```

Если функции `which()` не удается найти файл, соответствующий параметрам поиска, то она возвращает значение `None`.

```
$ python3 shutil_which.py

/Users/dhellmann/Library/Python/3.5/bin/virtualenv
/Users/dhellmann/Library/Python/3.5/bin/tox
None
```

Функция `which()` получает аргументы, обеспечивающие фильтрацию файлов на основании прав доступа к ним и путей поиска. По умолчанию аргумент `path` имеет значение `os.environ('PATH')`. Вместо него можно задать любую строку, содержащую список имен каталогов, в котором в качестве разделителя используется `os.pathsep`. Аргумент `mode` должен быть битовой маской, соответствующей правам доступа к файлу. По умолчанию маска соответствует поиску исполняемых файлов, но в следующем примере используется маска, предоставляющая доступ по чтению, и задается альтернативный путь поиска конфигурационного файла.

Листинг 6.71. shutil_which_regular_file.py

```
import os
import shutil

path = os.pathsep.join([
    '.',
    os.path.expanduser('~/pymotw'),
])

mode = os.F_OK | os.R_OK

filename = shutil.which(
    'config.ini',
    mode=mode,
    path=path,
)

print(filename)
```

При таком способе поиске файлов, доступных по чтению, могут возникать условия гонки, поскольку после того, как файл найден, но до того, как предпринимается попытка его использовать, он может быть удален или права доступа к нему могут быть изменены.

```
$ touch config.ini
$ python3 shutil_which_regular_file.py

./config.ini
```

6.7.5. Архивы

Стандартная библиотека Python включает многочисленные модули, такие как tarfile (раздел 8.4) или zipfile (раздел 8.5), для работы с архивными файлами. В дополнение к ним модуль shutil предлагает несколько высокоуровневых функций, предназначенных для создания и распаковки архивов. Функция get_archive_formats() возвращает последовательность имен и описаний форматов, поддерживаемых данной системой.

Листинг 6.72. shutil_get_archive_formats.py

```
import shutil

for format, description in shutil.get_archive_formats():
    print('{:<5}: {}'.format(format, description))
```

Состав поддерживаемых форматов определяется модулями и базовыми библиотеками, доступными в системе. Поэтому результаты выполнения данного примера на различных платформах могут отличаться от тех, которые приведены ниже.

```
$ python3 shutil_get_archive_formats.py

bztar: bzip2'ed tar-file
gztar: gzip'ed tar-file
tar  : uncompressed tar file
xztar: xz'ed tar-file
zip  : ZIP file
```

Для создания нового архивного файла можно использовать метод make_archive(), который обеспечивает рекурсивное архивирование всего каталога и его содержимого. По умолчанию используется текущий рабочий каталог, все файлы и каталоги которого оказываются на верхнем уровне архива. Это поведение можно изменить с помощью аргумента root_dir, позволяющего перейти к другой относительной позиции в файловой системе, и аргумента base_dir, позволяющего указать каталог, в который должен быть добавлен новый архив.

Листинг 6.73. shutil_make_archive.py

```
import logging
import shutil
import sys
import tarfile

logging.basicConfig(
    format='%(message)s',
    stream=sys.stdout,
    level=logging.DEBUG,
)
logger = logging.getLogger('pymotw')

print('Creating archive:')
shutil.make_archive(
    'example', 'gztar',
```

```
    root_dir='..',
    base_dir='shutil',
    logger=logger,
)

print('\nArchive contents:')
with tarfile.open('example.tar.gz', 'r') as t:
    for n in t.getnames():
        print(n)
```

В этом примере отправной точкой служит каталог, в котором находятся примеры для модуля `shutil`. Затем выполняется переход на один уровень вверх по иерархическому дереву файловой системы, и в tar-архив, сжатый с помощью программы `gzip`, добавляется каталог *shutil*. Модуль `logging` (раздел 14.8) конфигурируется для отображения сообщений из функции `make_archive()`, информирующих пользователя о выполняемых действиях.

```
$ python3 shutil_make_archive.py

Creating archive:
changing into '..'
Creating tar archive
changing back to '...'

Archive contents:
shutil
shutil/config.ini
shutil/example.out
shutil/file_to_change.txt
shutil/index.rst
shutil/shutil_copy.py
shutil/shutil_copy2.py
shutil/shutil_copyfile.py
shutil/shutil_copyfileobj.py
shutil/shutil_copymode.py
shutil/shutil_copystat.py
shutil/shutil_copytree.py
shutil/shutil_copytree_verbose.py
shutil/shutil_disk_usage.py
shutil/shutil_get_archive_formats.py
shutil/shutil_get_unpack_formats.py
shutil/shutil_make_archive.py
shutil/shutil_move.py
shutil/shutil_rmtree.py
shutil/shutil_unpack_archive.py
shutil/shutil_which.py
shutil/shutil_which_regular_file.py
```

Модуль `shutil` поддерживает реестр форматов, которые могут быть распакованы на текущей платформе. Доступ к этому реестру обеспечивает функция `get_unpack_formats()`.

Листинг 6.74. shutil_get_unpack_formats.py

```
import shutil

for format, exts, description in shutil.get_unpack_formats():
print('{:<5}: {}, names ending in {}'.format(
    format, description, exts))
```

Реестр, управляемый модулем shutil, отличается от реестра форматов, доступных для создания архивов, поскольку он включает также распространенные расширения имен файлов, используемые для каждого формата. На основании информации о расширении файла, содержащейся в этом реестре, функция, выполняющая распаковку архива, может делать предположения относительно того, какой формат следует использовать.

```
$ python3 shutil_get_unpack_formats.py

bztar: bzip2'ed tar-file, names ending in ['.tar.bz2', '.tbz2']
gztar: gzip'ed tar-file, names ending in ['.tar.gz', '.tgz']
tar  : uncompressed tar file, names ending in ['.tar']
xztar: xz'ed tar-file, names ending in ['.tar.xz', '.txz']
zip  : ZIP file, names ending in ['.zip']
```

Для извлечения архива используйте функцию unpack_archive(), передав ей имя файла архива и необязательный параметр – каталог, в который следует извлечь архив. Если этот каталог не указан, используется текущий каталог.

Листинг 6.75. shutil_unpack_archive.py

```
import pathlib
import shutil
import sys
import tempfile

with tempfile.TemporaryDirectory() as d:
    print('Unpacking archive:')
    shutil.unpack_archive(
        'example.tar.gz',
        extract_dir=d,
    )

    print('\nCreated:')
    prefix_len = len(d) + 1
    for extracted in pathlib.Path(d).rglob('*'):
        print(str(extracted)[prefix_len:])
```

В этом примере функция unpack_archive() может определить формат архива, поскольку имя файла заканчивается расширением .tar.gz, а с этим расширением в реестре распаковываемых форматов связан формат *.gztar*.

```
$ python3 shutil_unpack_archive.py

Unpacking archive:
```

```
Created:
shutil
shutil/config.ini
shutil/example.out
shutil/file_to_change.txt
shutil/index.rst
shutil/shutil_copy.py
shutil/shutil_copy2.py
shutil/shutil_copyfile.py
shutil/shutil_copyfileobj.py
shutil/shutil_copymode.py
shutil/shutil_copystat.py
shutil/shutil_copytree.py
shutil/shutil_copytree_verbose.py
shutil/shutil_disk_usage.py
shutil/shutil_get_archive_formats.py
shutil/shutil_get_unpack_formats.py
shutil/shutil_make_archive.py
shutil/shutil_move.py
shutil/shutil_rmtree.py
shutil/shutil_unpack_archive.py
shutil/shutil_which.py
shutil/shutil_which_regular_file.py
```

6.7.6. Размер файловой системы

Нередко перед выполнением длительных операций, с которыми связан риск исчерпания дискового пространства, полезно определить размер файловой системы, чтобы выяснить, достаточно ли доступного места на диске. Функция `disk_usage()` возвращает кортеж, элементы которого представляют общий объем диска, занятый объем и остающийся объем (свободное дисковое пространство).

Листинг 6.76. `shutil_disk_usage.py`

```
import shutil

total_b, used_b, free_b = shutil.disk_usage('.')

gib = 2 ** 30 # GiB == гибибайт
gb = 10 ** 9  # GB == гигабайт

print('Total: {:6.2f} GB {:6.2f} GiB'.format(
    total_b / gb, total_b / gib))
print('Used : {:6.2f} GB {:6.2f} GiB'.format(
    used_b / gb, used_b / gib))
print('Free : {:6.2f} GB {:6.2f} GiB'.format(
    free_b / gb, free_b / gib))
```

Функция `disk_usage()` возвращает значения, выраженные в количестве байтов, поэтому, прежде чем вывести их, программа выполняет преобразование в более удобные единицы измерения.

```
$ python3 shutil_disk_usage.py

Total: 499.42 GB 465.12 GiB
Used : 246.68 GB 229.73 GiB
Free : 252.48 GB 235.14 GiB
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `shutil`[10].
- Глава 8. Описание модулей, предназначенных для работы с различными форматами сжатия и архивирования данных.

6.8. `filelecmp`: сравнение файлов

Модуль `filecmp` включает функции и класс, позволяющие сравнивать файлы и каталоги, которые хранятся в файловой системе.

6.8.1. Данные для примеров

В последующих примерах используется набор тестовых файлов, созданный с помощью сценария `filecmp_mkexamples.py`.

Листинг 6.77. `filecmp_mkexamples.py`

```
import os


def mkfile(filename, body=None):
    with open(filename, 'w') as f:
        f.write(body or filename)
    return


def make_example_dir(top):
    if not os.path.exists(top):
        os.mkdir(top)
    curdir = os.getcwd()
    os.chdir(top)

    os.mkdir('dir1')
    os.mkdir('dir2')

    mkfile('dir1/file_only_in_dir1')
    mkfile('dir2/file_only_in_dir2')

    os.mkdir('dir1/dir_only_in_dir1')
    os.mkdir('dir2/dir_only_in_dir2')

    os.mkdir('dir1/common_dir')
    os.mkdir('dir2/common_dir')
```

[10] https://docs.python.org/3.5/library/shutil.html

```
    mkfile('dir1/common_file', 'this file is the same')
    mkfile('dir2/common_file', 'this file is the same')

    mkfile('dir1/not_the_same')
    mkfile('dir2/not_the_same')

    mkfile('dir1/file_in_dir1', 'This is a file in dir1')
    os.mkdir('dir2/file_in_dir1')

    os.chdir(curdir)
    return


if __name__ == '__main__':
    os.chdir(os.path.dirname(__file__) or os.getcwd())
    make_example_dir('example')
    make_example_dir('example/dir1/common_dir')
    make_example_dir('example/dir2/common_dir')
```

В результате выполнения этого сценария в каталоге *example* создается дерево файлов.

```
$ find example

example
example/dir1
example/dir1/common_dir
example/dir1/common_dir/dir1
example/dir1/common_dir/dir1/common_dir
example/dir1/common_dir/dir1/common_file
example/dir1/common_dir/dir1/dir_only_in_dir1
example/dir1/common_dir/dir1/file_in_dir1
example/dir1/common_dir/dir1/file_only_in_dir1
example/dir1/common_dir/dir1/not_the_same
example/dir1/common_dir/dir2
example/dir1/common_dir/dir2/common_dir
example/dir1/common_dir/dir2/common_file
example/dir1/common_dir/dir2/dir_only_in_dir2
example/dir1/common_dir/dir2/file_in_dir1
example/dir1/common_dir/dir2/file_only_in_dir2
example/dir1/common_dir/dir2/not_the_same
example/dir1/common_file
example/dir1/dir_only_in_dir1
example/dir1/file_in_dir1
example/dir1/file_only_in_dir1
example/dir1/not_the_same
example/dir2
example/dir2/common_dir
example/dir2/common_dir/dir1
example/dir2/common_dir/dir1/common_dir
example/dir2/common_dir/dir1/common_file
example/dir2/common_dir/dir1/dir_only_in_dir1
example/dir2/common_dir/dir1/file_in_dir1
```

```
example/dir2/common_dir/dir1/file_only_in_dir1
example/dir2/common_dir/dir1/not_the_same
example/dir2/common_dir/dir2
example/dir2/common_dir/dir2/common_dir
example/dir2/common_dir/dir2/common_file
example/dir2/common_dir/dir2/dir_only_in_dir2
example/dir2/common_dir/dir2/file_in_dir1
example/dir2/common_dir/dir2/file_only_in_dir2
example/dir2/common_dir/dir2/not_the_same
example/dir2/common_file
example/dir2/dir_only_in_dir2
example/dir2/file_in_dir1
example/dir2/file_only_in_dir2
example/dir2/not_the_same
```

Одна и та же структура каталогов повторяется по одному разу в каталогах *common_dir*, что открывает интересные возможности для исследования рекурсивных операций сравнения.

6.8.2. Сравнение файлов

Функция `cmp()` сравнивает два файла, принадлежащих файловой системе.

Листинг 6.78. `filecmp_cmp.py`

```
import filecmp

print('common_file :', end=' ')
print(filecmp.cmp('example/dir1/common_file',
                  'example/dir2/common_file'),
      end=' ')
print(filecmp.cmp('example/dir1/common_file',
                  'example/dir2/common_file',
                  shallow=False))

print('not_the_same:', end=' ')
print(filecmp.cmp('example/dir1/not_the_same',
                  'example/dir2/not_the_same'),
      end=' ')
print(filecmp.cmp('example/dir1/not_the_same',
                  'example/dir2/not_the_same',
                  shallow=False))

print('identical   :', end=' ')
print(filecmp.cmp('example/dir1/file_only_in_dir1',
                  'example/dir1/file_only_in_dir1'),
      end=' ')
print(filecmp.cmp('example/dir1/file_only_in_dir1',
                  'example/dir1/file_only_in_dir1',
                  shallow=False))
```

Аргумент `shallow` функции `cmp()` позволяет указать, следует ли сравнивать содержимое файлов, а не только метаданные. По умолчанию выполняется поверхностное сравнение с использованием информации, возвращаемой функцией

os.stat(). Если результаты для обоих файлов совпадают, то они считаются одинаковыми. Таким образом, файлы одного и того же размера, созданные в одно и то же время, считаются одинаковыми, даже если они различаются своим содержимым. Содержимое файлов всегда сравнивается, если аргумент shallow имеет значение False.

```
$ python3 filecmp_cmp.py

common_file : True True
not_the_same: True False
identical   : True True
```

Для сравнения двух наборов файлов, находящихся в двух разных каталогах, без применения рекурсии используйте функцию cmpfiles(). Аргументами этой функции являются имена каталогов и список файлов, подлежащих сравнению. Передаваемый список общих файлов должен содержать только имена файлов (указание каталогов всегда приводит к несовпадению файлов), и все указанные файлы должны находиться в обоих расположениях. В следующем примере продемонстрирован простой способ конструирования списка общих файлов. Как и функция cmp(), данная функция также получает флаг shallow.

Листинг 6.79. filecmp_cmpfiles.py

```
import filecmp
import os

# Определение элементов, существующих в обоих каталогах
d1_contents = set(os.listdir('example/dir1'))
d2_contents = set(os.listdir('example/dir2'))
common = list(d1_contents & d2_contents)
common_files = [
    f
    for f in common
    if os.path.isfile(os.path.join('example/dir1', f))
]
print('Common files:', common_files)

# Сравнение каталогов
match, mismatch, errors = filecmp.cmpfiles(
    'example/dir1',
    'example/dir2',
    common_files,
)
print('Match       :', match)
print('Mismatch    :', mismatch)
print('Errors      :', errors)
```

Функция cmpfiles() возвращает три списка имен файлов, которые соответствуют совпадающим и несовпадающим файлам, а также файлам, сравнение которых не может быть выполнено (ввиду отсутствия соответствующих полномочий доступа или по иным причинам).

```
$ python3 filecmp_cmpfiles.py

Common files: ['common_file', 'not_the_same', 'file_in_dir1']
Match       : ['common_file', 'not_the_same']
Mismatch    : ['file_in_dir1']
Errors      : []
```

6.8.3. Сравнение каталогов

Описанные до этого функции удобны для выполнения относительно простых сравнений. Если требуется рекурсивное сравнение больших деревьев каталогов или проведение более полного катализа, гораздо лучше использовать класс `dircmp`. В простейших случаях для получения отчета о результатах сравнения двух каталогов можно использовать функцию `report()`.

Листинг 6.80. `filecmp_dircmp_report.py`

```
import filecmp

dc = filecmp.dircmp('example/dir1', 'example/dir2')
dc.report()
```

В отчете представлены результаты сравнения, которое было выполнено лишь для каталогов, переданных функции в качестве аргументов, без применения рекурсии. В данном случае файлы `not_the_same` признаны одинаковыми, поскольку их содержимое не сравнивалось. Класс `dircmp` не может сравнивать содержимое файлов, как это делает функция `cmp()`.

```
$ python3 filecmp_dircmp_report.py

diff example/dir1 example/dir2
Only in example/dir1 : ['dir_only_in_dir1', 'file_only_in_dir1']
Only in example/dir2 : ['dir_only_in_dir2', 'file_only_in_dir2']
Identical files : ['common_file', 'not_the_same']
Common subdirectories : ['common_dir']
Common funny cases : ['file_in_dir1']
```

Для получения более подробной информации и выполнения рекурсивной операции сравнения следует использовать функцию `report_full_closure()`.

Листинг 6.81. `filecmp_dircmp_report_full_closure.py`

```
import filecmp

dc = filecmp.dircmp('example/dir1', 'example/dir2')
dc.report_full_closure()
```

Вывод включает результаты сравнения всех параллельных подкаталогов.

```
$ python3 filecmp_dircmp_report_full_closure.py

diff example/dir1 example/dir2
Only in example/dir1 : ['dir_only_in_dir1', 'file_only_in_dir1']
```

```
Only in example/dir2 : ['dir_only_in_dir2', 'file_only_in_dir2']
Identical files : ['common_file', 'not_the_same']
Common subdirectories : ['common_dir']
Common funny cases : ['file_in_dir1']

diff example/dir1/common_dir example/dir2/common_dir
Common subdirectories : ['dir1', 'dir2']

diff example/dir1/common_dir/dir1 example/dir2/common_dir/dir1
Identical files : ['common_file', 'file_in_dir1',
'file_only_in_dir1', 'not_the_same']
Common subdirectories : ['common_dir', 'dir_only_in_dir1']

diff example/dir1/common_dir/dir1/dir_only_in_dir1
example/dir2/common_dir/dir1/dir_only_in_dir1

diff example/dir1/common_dir/dir1/common_dir
example/dir2/common_dir/dir1/common_dir

diff example/dir1/common_dir/dir2 example/dir2/common_dir/dir2
Identical files : ['common_file', 'file_only_in_dir2',
'not_the_same']
Common subdirectories : ['common_dir', 'dir_only_in_dir2',
'file_in_dir1']

diff example/dir1/common_dir/dir2/common_dir
example/dir2/common_dir/dir2/common_dir

diff example/dir1/common_dir/dir2/file_in_dir1
example/dir2/common_dir/dir2/file_in_dir1

diff example/dir1/common_dir/dir2/dir_only_in_dir2
example/dir2/common_dir/dir2/dir_only_in_dir2
```

6.8.4. Использование различий в программах

Кроме вывода отчетов, класс `dircmp` способен подготавливать списки файлов, которые можно использовать непосредственно в программах. Каждый из следующих атрибутов рассчитывается лишь тогда, когда это требуется, поэтому создание экземпляра `dircmp` не влечет за собой напрасное расходование ресурсов для неиспользуемых данных.

Листинг 6.82. `filecmp_dircmp_list.py`

```
import filecmp
import pprint

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print('Left:')
pprint.pprint(dc.left_list)

print('\nRight:')
pprint.pprint(dc.right_list)
```

Файлы и подкаталоги, содержащиеся в сравниваемых каталогах, перечислены в списках inleft_list и right_list.

```
$ python3 filecmp_dircmp_list.py

Left:
['common_dir',
 'common_file',
 'dir_only_in_dir1',
 'file_in_dir1',
 'file_only_in_dir1',
 'not_the_same']

Right:
['common_dir',
 'common_file',
 'dir_only_in_dir2',
 'file_in_dir1',
 'file_only_in_dir2',
 'not_the_same']
```

Входную информацию можно фильтровать, передав конструктору список имен, которые должны игнорироваться. По умолчанию игнорируются имена RCS, CVS и tags.

Листинг 6.83. filecmp_dircmp_list_filter.py

```
import filecmp
import pprint

dc = filecmp.dircmp('example/dir1', 'example/dir2',
                    ignore=['common_file'])

print('Left:')
pprint.pprint(dc.left_list)

print('\nRight:')
pprint.pprint(dc.right_list)
```

В этом случае файл common_file выпадает из списка файлов, подлежащих сравнению.

```
$ python3 filecmp_dircmp_list_filter.py

Left:
['common_dir',
 'dir_only_in_dir1',
 'file_in_dir1',
 'file_only_in_dir1',
 'not_the_same']

Right:
['common_dir',
```

```
 'dir_only_in_dir2',
 'file_in_dir1',
 'file_only_in_dir2',
 'not_the_same']
```

Имена файлов, общих для обоих каталогов, сохраняются в атрибуте common, тогда как имена файлов, уникальных для каждого из каталогов, сохраняются в атрибутах left_only и right_only.

Листинг 6.84. filecmp_dircmp_membership.py

```
import filecmp
import pprint

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print('Common:')
pprint.pprint(dc.common)

print('\nLeft:')
pprint.pprint(dc.left_only)

print('\nRight:')
pprint.pprint(dc.right_only)
```

"Левый" (Left) каталог соответствует первому аргументу конструктора dircmp(), а "правый" (Right) — второму.

```
$ python3 filecmp_dircmp_membership.py

Common:
['file_in_dir1', 'common_file', 'common_dir', 'not_the_same']

Left:
['dir_only_in_dir1', 'file_only_in_dir1']

Right:
['file_only_in_dir2', 'dir_only_in_dir2']
```

Общие элементы могут быть далее разбиты на отдельные категории, соответствующие файлам, подкаталогам и "посторонним" (Funny) элементам (любые объекты, имеющие разные типы в двух каталогах, или такие, для которых функция os.stat() возвращает ошибку).

Листинг 6.85. filecmp_dircmp_common.py

```
import filecmp
import pprint

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print('Common:')
pprint.pprint(dc.common)

print('\nDirectories:')
pprint.pprint(dc.common_dirs)
```

```
print('\nFiles:')
pprint.pprint(dc.common_files)

print('\nFunny:')
pprint.pprint(dc.common_funny)
```

В данных, используемых в примерах, элемент `file_in_dir1` является файлом в одном каталоге и подкаталогом в другом, поэтому он отнесен к категории "посторонних" (`Funny`).

```
$ python3 filecmp_dircmp_common.py

Common:
['file_in_dir1', 'common_file', 'common_dir', 'not_the_same']

Directories:
['common_dir']

Files:
['common_file', 'not_the_same']

Funny:
['file_in_dir1']
```

Различия между файлами классифицируются аналогичным образом.

Листинг 6.86. `filecmp_dircmp_diff.py`

```
import filecmp
dc = filecmp.dircmp('example/dir1', 'example/dir2')
print('Same      :', dc.same_files)
print('Different :', dc.diff_files)
print('Funny     :', dc.funny_files)
```

Файлы `not_the_same` сравниваются посредством функции `os.stat()`, и их содержимое не проверяется, поэтому они включаются в список `same_files`.

```
$ python3 filecmp_dircmp_diff.py

Same      : ['common_file', 'not_the_same']
Different : []
Funny     : []
```

Наконец, подкаталоги сохраняются с целью упрощения рекурсивного сравнения.

Листинг 6.87. `filecmp_dircmp_subdirs.py`

```
import filecmp

dc = filecmp.dircmp('example/dir1', 'example/dir2')
print('Subdirectories:')
print(dc.subdirs)
```

Атрибут `subdirs` — это словарь, сопоставляющий имя каталога с новыми объектами `dircmp`.

```
$ python3 filecmp_dircmp_subdirs.py

Subdirectories:
{'common_dir': <filecmp.dircmp object at 0x1019b2be0>}
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `filecmp`[11].
- `difflib` (раздел 1.4). Вычисление различий между двумя последовательностями.

6.9. mmap: файлы, отображаемые в памяти

Отображение файлов в памяти обеспечивает прямой доступ к данным, хранящимся в файловой системе, за счет использования виртуальной памяти операционной системы вместо обычных функций ввода-вывода. Такой подход позволяет улучшить производительность операций ввода-вывода, поскольку он не требует выполнения отдельных системных вызовов для каждой попытки доступа или копирования данных между буферами. Вместо этого как ядро, так и пользовательское приложение получают прямой доступ к памяти.

В зависимости от конкретных задач файлы, отображаемые в памяти, могут рассматриваться как изменяемые строки или файловые объекты. Отображенный файл поддерживает ожидаемые методы файлового API, такие как `close()`, `flush()`, `read()`, `readline()`, `seek()`, `tell()` и `write()`. Он также поддерживает строковый API, предоставляющий такие средства, как взятие срезов и методы наподобие `find()`.

В примерах этого раздела используется текстовый файл *lorem.txt* (листинг 6.88), в котором содержится часть полного текста Lorem Ipsum.

Листинг 6.88. `lorem.txt`

```
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Donec egestas, enim et consectetuer ullamcorper, lectus ligula
rutrum leo, a elementum elit tortor eu quam. Duis tincidunt nisi ut
ante. Nulla facilisi. Sed tristique eros eu libero. Pellentesque vel
arcu. Vivamus purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra fringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
mauris in nibh placerat egestas. Suspendisse potenti. Mauris
massa. Ut eget velit auctor tortor blandit sollicitudin. Suspendisse
imperdiet justo.
```

[11] https://docs.python.org/3.5/library/filecmp.html

Примечание

Между версиями функции `mmap()` для Unix и Windows существуют различия в поведении и получаемых аргументах, однако ниже эти различия не обсуждаются в полной мере. Для получения более подробной информации по этому вопросу обратитесь к документации стандартной библиотеки.

6.9.1. Чтение

Для создания файлов в памяти предназначена функция `mmap()`. Ее первым аргументом является дескриптор файла, получаемый с помощью метода `fileno()` объекта файла или функции `os.open()`. Ответственность за открытие файла перед вызовом функции `mmap()` и его закрытие, когда необходимость в нем отпадает, возлагается на вызывающий код.

Второй аргумент функции `mmap()` — размер в байтах той части файла, которая подлежит отображению. Если он равен `0`, то отображается весь файл. Если он превышает текущий размер файла, то файл расширяется.

Примечание

Windows не поддерживает создание отображений нулевой длины.

Необязательный именованный аргумент `access` поддерживается обеими платформами. Используйте константу `ACCESS_READ` для доступа только по чтению, константу `ACCESS_WRITE` — для доступа по сквозной записи (присваивание значений отображению в памяти записывается непосредственно в файл) и константу `ACCESS_COPY` — для доступа с правами копирования по записи (присваивание значений отображению в памяти не записывается в файл).

Листинг 6.89. `mmap_read.py`

```
import mmap

with open('lorem.txt', 'r') as f:
    with mmap.mmap(f.fileno(), 0,
                   access=mmap.ACCESS_READ) as m:
        print('First 10 bytes via read :', m.read(10))
        print('First 10 bytes via slice:', m[:10])
        print('2nd   10 bytes via read :', m.read(10))
```

Указатель файла отслеживает последний байт, к которому осуществлялся доступ посредством операции среза. В данном примере указатель перемещается вперед на 10 байт после первого чтения. Затем он перемещается в начало файла посредством операции среза и вновь перемещается вперед на 10 байт по срезу. После выполнения операции среза повторный вызов метода `read()` возвращает байты 11–20 файла.

```
$ python3 mmap_read.py

First 10 bytes via read : b'Lorem ipsu'
First 10 bytes via slice: b'Lorem ipsu'
2nd   10 bytes via read : b'm dolor si'
```

6.9.2. Запись

Чтобы настроить получение обновлений для файла, отображаемого в памяти, откройте его для присоединения данных в режиме `'r+'` (а не `'w'`), прежде чем отображать его. Затем используйте любой из методов API, изменяющих данные (например, метод `write()`, присваивание срезу).

В следующем примере часть строки изменяется на месте с использованием режима доступа по умолчанию `ACCESS_WRITE` и присваивания срезу.

Листинг 6.90. `mmap_write_slice.py`

```
import mmap
import shutil

# Копирование файла примера
shutil.copyfile('lorem.txt', 'lorem_copy.txt')

word = b'consectetuer'
reversed = word[::-1]
print('Looking for    :', word)
print('Replacing with :', reversed)

with open('lorem_copy.txt', 'r+') as f:
    with mmap.mmap(f.fileno(), 0) as m:
        print('Before:\n{}'.format(m.readline().rstrip()))
        m.seek(0)  # перейти в начало

        loc = m.find(word)
        m[loc:loc + len(word)] = reversed
        m.flush()

        m.seek(0)  # перейти в начало
        print('After :\n{}'.format(m.readline().rstrip()))

        f.seek(0)  # перейти в начало
        print('File  :\n{}'.format(f.readline().rstrip()))
```

Слово "consectetuer", находящееся посреди первой строки, изменяется в памяти и в файле.

```
$ python3 mmap_write_slice.py

Looking for : b'consectetuer'
Replacing with : b'reutetcesnoc'
Before:
b'Lorem ipsum dolor sit amet, consectetuer adipiscing elit.'
After :
b'Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit.'
File :
Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit.
```

6.9.2.1. Режим копирования

В режиме доступа ACCESS_COPY изменения не записываются в файл на диске.

Листинг 6.91. mmap_write_copy.py

```
import mmap
import shutil

# Копирование файла примера
shutil.copyfile('lorem.txt', 'lorem_copy.txt')

word = b'consectetuer'
reversed = word[::-1]

with open('lorem_copy.txt', 'r+') as f:
    with mmap.mmap(f.fileno(), 0,
                   access=mmap.ACCESS_COPY) as m:
        print('Memory Before:\n{}'.format(
            m.readline().rstrip()))
        print('File Before  :\n{}\n'.format(
            f.readline().rstrip()))

        m.seek(0)  # перейти в начало
        loc = m.find(word)
        m[loc:loc + len(word)] = reversed

        m.seek(0)  # перейти в начало
        print('Memory After :\n{}'.format(
            m.readline().rstrip()))

        f.seek(0)
        print('File After   :\n{}'.format(
            f.readline().rstrip()))
```

В этом примере дескриптор файла нужно было переместить в начало независимо от дескриптора отображения, поскольку внутренние состояния обоих объектов поддерживаются по отдельности.

```
$ python3 mmap_write_copy.py

Memory Before:
b'Lorem ipsum dolor sit amet, consectetuer adipiscing elit.'
File Before :
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Memory After :
b'Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit.'
File After :
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
```

6.9.3. Регулярные выражения

Поскольку файлы в памяти могут работать подобно строкам, их можно использовать совместно с другими модулями, оперирующими строками, такими как

модуль регулярных выражений. В следующем примере выполняется поиск всех предложений, в которых содержится слово "nulla".

Листинг 6.92. mmap_regex.py

```
import mmap
import re

pattern = re.compile(rb'(\.\W+)?([^.]?nulla[^.]*?\.)',
                     re.DOTALL | re.IGNORECASE | re.MULTILINE)

with open('lorem.txt', 'r') as f:
    with mmap.mmap(f.fileno(), 0,
                   access=mmap.ACCESS_READ) as m:
        for match in pattern.findall(m):
            print(match[1].replace(b'\n', b' '))
```

Поскольку шаблон включает две группы, значение, возвращаемое методом `findall()`, представляет собой последовательность кортежей. Инструкция `print` извлекает предложение, удовлетворяющее условиям поиска, и заменяет символы перевода строки пробелами, чтобы каждый результат выводился в одной строке.

```
$ python3 mmap_regex.py

b'Nulla facilisi.'
b'Nulla feugiat augue eleifend nulla.'
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `mmap`[12].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `mmap` (раздел A.6.27).
- `os` (раздел 17.3). Модуль `os`.
- `re` (раздел 1.3). Регулярные выражения.

6.10. codecs: кодирование и декодирование строк

Модуль `codecs` предоставляет потоковый и файловый интерфейсы для перекодировки текстовых данных между различными представлениями. Чаще всего его используют для работы с текстом в представлении Unicode, но доступны также другие кодировки, используемые для других целей.

6.10.1. Основы Unicode

CPython 3.*x* различает *текстовые* и *байтовые* строки. Экземпляры `bytes` используют последовательности 8-битовых байтовых значений. В противоположность этому внутренний механизм Python использует представление строк типа `str` в виде последовательностей кодовых точек Unicode. Для представления каждой ко-

[12] https://docs.python.org/3.5/library/mmap.html

довой точки используются 2 или 4 байта, в зависимости от опций, заданных при компиляции кода Python.

При выводе значений типа `str` они кодируются с использованием одной из нескольких стандартных схем, чтобы последовательность байтов можно было впоследствии восстановить в виде той же самой текстовой строки. Байты кодированного значения не обязательно должны совпадать со значениями кодовых точек, и схема кодирования определяет способ преобразования между обоими наборами значений. Кроме того, чтение данных Unicode требует знания кодировки, чтобы поступающие байты могли быть преобразованы во внутреннее представление, используемое классом `unicode`.

Наиболее распространенными схемами кодирования для западноевропейских языков являются UTF-8 и UTF-16, основанные соответственно на использовании последовательностей 1- и 2-байтовых значений соответственно для представления каждой кодовой точки. Для работы с другими языками более эффективны другие кодировки, в которых большинство символов представлено кодовыми точками, требующими более двух байтов.

Дополнительные ссылки

Для получения более подробной информации о Unicode обратитесь к списку дополнительных ссылок, приведенному в конце этого раздела, и, в частности, загляните в раздел *Unicode HOWTO* справочной системы Python.

6.10.1.1. Кодировки

Смысл кодировок становится наиболее понятным при изучении различных последовательностей байтов, получаемых в результате кодирования одной и той же строки различными способами. Ниже приведена функция, которая используется в последующих примерах для форматирования текстовой строки с целью приведения ее к виду, наиболее удобному для чтения.

Листинг 6.93. `codecs_to_hex.py`

```
import binascii


def to_hex(t, nbytes):
    """Отформатировать текст t как последовательность значений
    длиной nbytes, разделенных пробелами.
    """
    chars_per_item = nbytes * 2
    hex_version = binascii.hexlify(t)
    return b' '.join(
        hex_version[start:start + chars_per_item]
        for start in range(0, len(hex_version), chars_per_item)
    )


if __name__ == '__main__':
    print(to_hex(b'abcdef', 1))
    print(to_hex(b'abcdef', 2))
```

Данная функция использует модуль binascii для получения шестнадцатеричного представления входной байтовой строки, а затем вставляет пробелы между каждыми nbytes байтами, прежде чем вернуть значение.

```
$ python3 codecs_to_hex.py

b'61 62 63 64 65 66'
b'6162 6364 6566'
```

Первый пример начинается с вывода текста 'français' с использованием "сырого" (Raw) представления класса unicode, за которым следует вывод имен символов, извлекаемых из базы данных Unicode. В следующих двух строках кода исходная строка кодируется с использованием схем UTF-8 и UTF-16 и результирующие байты выводятся в виде шестнадцатеричных значений.

Листинг 6.94. codecs_encodings.py

```
import unicodedata
from codecs_to_hex import to_hex

text = 'français'

print('Raw : {!r}'.format(text))
for c in text:
    print(' {!r}: {}'.format(c, unicodedata.name(c, c)))
print('UTF-8 : {!r}'.format(to_hex(text.encode('utf-8'), 1)))
print('UTF-16: {!r}'.format(to_hex(text.encode('utf-16'), 2)))
```

Результатом кодирования строки str является объект bytes.

```
$ python3 codecs_encodings.py

Raw : 'français'
  'f': LATIN SMALL LETTER F
  'r': LATIN SMALL LETTER R
  'a': LATIN SMALL LETTER A
  'n': LATIN SMALL LETTER N
  'ç': LATIN SMALL LETTER C WITH CEDILLA
  'a': LATIN SMALL LETTER A
  'i': LATIN SMALL LETTER I
  's': LATIN SMALL LETTER S
UTF-8 : b'66 72 61 6e c3 a7 61 69 73'
UTF-16: b'fffe 6600 7200 6100 6e00 e700 6100 6900 7300'
```

Если последовательность кодированных байтов задана в виде экземпляра bytes, метод decode() преобразует байты в кодовые точки и возвращает последовательность в виде экземпляра str.

Листинг 6.95. codecs_decode.py

```
from codecs_to_hex import to_hex

text = 'français'
encoded = text.encode('utf-8')
```

```
decoded = encoded.decode('utf-8')

print('Original :', repr(text))
print('Encoded  :', to_hex(encoded, 1), type(encoded))
print('Decoded  :', repr(decoded), type(decoded))
```

Выводимый тип не зависит от выбора исходной кодировки.

```
$ python3 codecs_decode.py

Original : 'français'
Encoded  : b'66 72 61 6e c3 a7 61 69 73' <class 'bytes'>
Decoded  : 'français' <class 'str'>
```

Примечание

Кодировка, используемая по умолчанию, устанавливается во время запуска интерпретатора при загрузке модуля `site` (раздел 17.1). Более подробное описание настройки кодировки, используемой по умолчанию, содержится в разделе 17.2.1.4.

6.10.2. Работа с файлами

Кодирование и декодирование строк играют особо важную роль в операциях ввода-вывода. Независимо от того, куда осуществляется запись, — в файл, сокет или другой поток, — данные должны использовать подходящую кодировку. Вообще говоря, все данные должны декодироваться из байтового представления при их чтении, а внутренние значения должны кодироваться в определенное представление при их записи. Программа может явно кодировать и декодировать данные, однако, в зависимости от используемой кодировки, определение того, достаточно ли байтов считано для того, чтобы полностью декодировать данные, не всегда является тривиальной задачей. Модуль `codecs` предоставляет классы, управляющие кодированием и декодированием данных, избавляя приложение от необходимости выполнять эту работу.

Простейший интерфейс, предоставляемый модулем `codecs`, является альтернативой встроенной функции `open()`. Новая версия работает так же, как и встроенная, но добавляет два новых аргумента, позволяющих указать кодировку и желаемый способ обработки ошибок.

Листинг 6.96. `codecs_open_write.py`

```
from codecs_to_hex import to_hex

import codecs
import sys

encoding = sys.argv[1]
filename = encoding + '.txt'

print('Writing to', filename)
with codecs.open(filename, mode='w', encoding=encoding) as f:
    f.write('français')
```

```
# Определение группирования байтов для использования с to_hex()
nbytes = {
    'utf-8': 1,
    'utf-16': 2,
    'utf-32': 4,
}.get(encoding, 1)

# Отображение "сырых" байтов, хранящихся в файле
print('File contents:')
with open(filename, mode='rb') as f:
    print(to_hex(f.read(), nbytes))
```

В этом примере строка Unicode, содержащая букву 'ç', сохраняется в виде текста в файле с использованием кодировки, указанной в командной строке.

```
$ python3 codecs_open_write.py utf-8

Writing to utf-8.txt
File contents:
b'66 72 61 6e c3 a7 61 69 73'

$ python3 codecs_open_write.py utf-16

Writing to utf-16.txt
File contents:
b'fffe 6600 7200 6100 6e00 e700 6100 6900 7300'

$ python3 codecs_open_write.py utf-32

Writing to utf-32.txt
File contents:
b'fffe0000 66000000 72000000 61000000 6e000000 e7000000 61000000
69000000 73000000'
```

Чтение данных с помощью функции `open()` осуществляется непосредственно за один заход: кодировка должна быть известна заранее, чтобы можно было правильно задать декодер. Некоторые форматы данных, такие как XML, предполагают указание кодировки в самом файле, но обычно этим должно управлять приложение. Модуль `codecs` просто получает кодировку в качестве аргумента и предполагает, что она задана корректно.

Листинг 6.97. `codecs_open_read.py`

```
import codecs
import sys

encoding = sys.argv[1]
filename = encoding + '.txt'

print('Reading from', filename)
with codecs.open(filename, mode='r', encoding=encoding) as f:
    print(repr(f.read()))
```

В этом примере читаются файлы, созданные предыдущей программой, и данные выводятся на консоль в представлении результирующего объекта unicode.

```
$ python3 codecs_open_read.py utf-8

Reading from utf-8.txt
'français'

$ python3 codecs_open_read.py utf-16

Reading from utf-16.txt
'français'

$ python3 codecs_open_read.py utf-32

Reading from utf-32.txt
'français'
```

6.10.3. Порядок байтов

Многобайтовые кодировки, такие как UTF-16 и UTF-32, создают проблемы при переносе данных между различными компьютерными системами, как при непосредственном копировании файлов, так и при передаче их по сети. В различных системах используется разный порядок следования старших и младших байтов. Эта характеристика данных зависит от таких факторов, как архитектура компьютерного оборудования и выбор, сделанный разработчиками операционной системы и приложения. Невозможно всегда заранее знать, с каким порядком байтов придется иметь дело, поэтому многобайтовые кодировки включают маркер порядка следования байтов (BOM) в качестве первых нескольких байтов кодированных выходных данных. Например, кодировка UTF-16 определена таким образом, что кодам 0xFFFE и 0xFEFF не соответствуют никакие допустимые символы, и их можно использовать для указания порядка байтов. В модуле codecs определены константы, которые служат маркерами порядка байтов, используемыми кодировками UTF-16 и UTF-32.

Листинг 6.98. codecs_bom.py

```
import codecs
from codecs_to_hex import to_hex

BOM_TYPES = [
    'BOM', 'BOM_BE', 'BOM_LE',
    'BOM_UTF8',
    'BOM_UTF16', 'BOM_UTF16_BE', 'BOM_UTF16_LE',
    'BOM_UTF32', 'BOM_UTF32_BE', 'BOM_UTF32_LE',
]

for name in BOM_TYPES:
    print('{:12} : {}'.format(
        name, to_hex(getattr(codecs, name), 2)))
```

Для констант BOM, BOM_UTF16 и BOM_UTF32 автоматически устанавливаются значения, соответствующие прямому ("от старшего к младшему" – big-endian) или обратному ("от младшего к старшему" – little-endian) порядку байтов, в зависимости от собственного машинного порядка следования байтов в данной системе.

```
$ python3 codecs_bom.py

BOM           : b'fffe'
BOM_BE        : b'feff'
BOM_LE        : b'fffe'
BOM_UTF8      : b'efbb bf'
BOM_UTF16     : b'fffe'
BOM_UTF16_BE  : b'feff'
BOM_UTF16_LE  : b'fffe'
BOM_UTF32     : b'fffe 0000'
BOM_UTF32_BE  : b'0000 feff'
BOM_UTF32_LE  : b'fffe 0000'
```

Порядок следования байтов автоматически распознается и обрабатывается декодерами модуля codecs, но при кодировании данных этот порядок можно задать явно.

Листинг 6.99. codecs_bom_create_file.py

```
import codecs
from codecs_to_hex import to_hex

# Выбрать несобственную версию кодировки UTF-16
if codecs.BOM_UTF16 == codecs.BOM_UTF16_BE:
    bom = codecs.BOM_UTF16_LE
    encoding = 'utf_16_le'
else:
    bom = codecs.BOM_UTF16_BE
    encoding = 'utf_16_be'

print('Native order  :', to_hex(codecs.BOM_UTF16, 2))
print('Selected order:', to_hex(bom, 2))

# Кодирование текста
encoded_text = 'français'.encode(encoding)
print('{:14}: {}'.format(encoding, to_hex(encoded_text, 2)))

with open('nonnative-encoded.txt', mode='wb') as f:
    # Записать выбранный маркер порядка следования байтов. Он
    # не включается в кодируемый текст, поскольку порядок
    # следования байтов был задан явно при выборе кодировки.
    f.write(bom)
    # Записать байтовую строку кодированного текста
    f.write(encoded_text)
```

Сценарий codecs_bom_create_file.py определяет собственный машинный порядок следования байтов, а затем явно использует альтернативную форму, чтобы в следующем примере можно было продемонстрировать автоматическое распознавание порядка байтов во время чтения данных.

```
$ python3 codecs_bom_create_file.py

Native order  : b'fffe'
Selected order: b'feff'
utf_16_be     : b'0066 0072 0061 006e 00e7 0061 0069 0073'
```

Сценарий codecs_bom_detection.py не определяет порядок следования байтов при открытии файла, поэтому декодер распознает его, используя значение BOM из первых 2 байтов файла.

Листинг 6.100. codecs_bom_detection.py

```
import codecs
from codecs_to_hex import to_hex

# Чтение "сырых" данных
with open('nonnative-encoded.txt', mode='rb') as f:
    raw_bytes = f.read()

print('Raw    :', to_hex(raw_bytes, 2))

# Заново открыть файл и позволить модулю codecs распознать BOM
with codecs.open('nonnative-encoded.txt',
                 mode='r',
                 encoding='utf-16',
                 ) as f:
    decoded_text = f.read()

print('Decoded:', repr(decoded_text))
```

Поскольку первые 2 байта файла используются для обнаружения порядка следования байтов, они не включаются в данные, возвращаемые методом read().

```
$ python3 codecs_bom_detection.py

Raw    : b'feff 0066 0072 0061 006e 00e7 0061 0069 0073'
Decoded: 'français'
```

6.10.4. Обработка ошибок

Как уже отмечалось в предыдущих разделах, чтение и запись файлов Unicode требует знания используемой кодировки. Корректное задание кодировки важно по следующим двум причинам. Во-первых, если кодировка задана неверно при чтении данных, то они будут неправильно интерпретироваться, что может привести к их порче или невозможности декодирования. Во-вторых, не все символы Unicode могут быть представлены во всех кодировках. Таким образом, в случае использования неподходящей кодировки при записи данных могут возникать ошибки или может происходить потеря данных.

Модуль codecs использует те же пять опций обработки ошибок (табл. 6.1), которые предоставляются методом encode() типа str и методом decode() типа bytes.

Таблица 6.1. Режимы обработки ошибок кодеков

Режим обработки ошибок	Описание
strict	Возбуждение исключения в случае невозможности преобразования данных
replace	Замена специального символа маркера для данных, которые не могут быть закодированы
ignore	Пропуск данных
xmlcharrefreplace	Замена ссылкой на подходящий символ XML (только при кодировании)
backslashreplace	Замена Escape-последовательностями (только при кодировании)

6.10.4.1. Ошибки кодирования

Чаще всего встречается ошибка UnicodeEncodeError, которая возникает при попытке записи данных Unicode в выходной поток ASCII, в качестве которого может выступать обычный файл или поток sys.stdout, без установки более надежной кодировки. Для проведения экспериментов с различными режимами обработки ошибок можно воспользоваться программой, приведенной в листинге 6.101.

Листинг 6.101. codecs_encode_error.py

```
import codecs
import sys

error_handling = sys.argv[1]

text = 'français'

try:
    # Сохранить данные в кодировке ASCII, используя режим
    # обработки ошибок, указанный в командной строке
    with codecs.open('encode_error.txt', 'w',
                     encoding='ascii',
                     errors=error_handling) as f:
        f.write(text)

except UnicodeEncodeError as err:
    print('ERROR:', err)

else:
    # В случае отсутствия ошибок при записи в файл отобразить
    # его содержимое
    with open('encode_error.txt', 'rb') as f:
        print('File contents: {!r}'.format(f.read()))
```

В то время как режим strict надежнее других гарантирует установку приложением корректной кодировки для всех операций ввода-вывода, он может приводить к краху программы при возникновении исключения.

```
$ python3 codecs_encode_error.py strict

ERROR: 'ascii' codec can't encode character '\xe7' in position
4: ordinal not in range(128)
```

Некоторые из оставшихся режимов обработки ошибок обладают большей гибкостью. Например, режим `replace` гарантирует устранение исключений за счет возможной потери данных, которые не могут быть преобразованы с использованием запрошенной кодировки. Для символа Unicode, соответствующего числу `pi`, не существует кода в таблице ASCII, но вместо возбуждения исключения этот символ заменяется вопросительным знаком (`?`).

```
$ python3 codecs_encode_error.py replace

File contents: b'fran?ais'
```

Чтобы всего лишь обойти возможные проблемы с кодировкой данных, используйте режим `ignore`. В этом случае данные, не поддающиеся кодированию, будут отбрасываться.

```
$ python3 codecs_encode_error.py ignore

File contents: b'franais'
```

Также доступны две опции, обеспечивающие кодирование данных без потерь, каждая из которых обеспечивает замену символа альтернативным представлением в соответствии со стандартом, независимым от кодировки. Опции `xmlcharrefreplace` соответствует использование ссылки на символ XML в качестве подстановки (список ссылок на символы определен в документе W3C "XML Entity Definitions for Characters").

```
$ python3 codecs_encode_error.py xmlcharrefreplace

File contents: b'fran&#231;ais'
```

Существует еще одна опция обработки ошибок без потери данных — `backslashreplace`, обеспечивающая вывод данных в формате значений, возвращаемых функцией `repr()` при выводе на печать объекта `unicode`. Символы Unicode заменяются специальным символом `\u`, за которым следует шестнадцатеричное значение кодовой точки.

```
$ python3 codecs_encode_error.py backslashreplace

File contents: b'fran\\xe7ais'
```

6.10.4.2. Ошибки декодирования

Ошибки могут возникать также при декодировании данных, особенно если используется неверная кодировка.

Листинг 6.102. `codecs_decode_error.py`

```
import codecs
import sys

from codecs_to_hex import to_hex

error_handling = sys.argv[1]

text = 'français'
print('Original     :', repr(text))

# Сохранить данные с использованием некоторой кодировки
with codecs.open('decode_error.txt', 'w',
                 encoding='utf-16') as f:
    f.write(text)

# Вывести байтовое содержимое файла
with open('decode_error.txt', 'rb') as f:
    print('File contents:', to_hex(f.read(), 1))

# Попытаться прочитать данные с использованием неверной кодировки
with codecs.open('decode_error.txt', 'r',
                 encoding='utf-8',
                 errors=error_handling) as f:
    try:
        data = f.read()
    except UnicodeDecodeError as err:
        print('ERROR:', err)
    else:
        print('Read         :', repr(data))
```

Как и в случае кодирования данных, если установлен режим обработки ошибок `strict` и байтовый поток не удается декодировать надлежащим образом, возбуждается исключение. В данном примере исключение `UnicodeDecodeError` возникает из-за попытки преобразовать часть маркера UTF-16 BOM в символ с использованием декодера UTF-8.

```
$ python3 codecs_decode_error.py strict

Original     : 'français'
File contents: b'ff fe 66 00 72 00 61 00 6e 00 e7 00 61 00 69 00
73 00'
ERROR: 'utf-8' codec can't decode byte 0xff in position 0:
invalid start byte
```

Переход к использованию опции `ignore` приводит к тому, что декодер пропускает недействительные байты. Однако результат все еще не соответствует тому, который ожидался, поскольку включает внедренные нулевые байты.

```
$ python3 codecs_decode_error.py ignore

Original     : 'français'
```

```
File contents: b'ff fe 66 00 72 00 61 00 6e 00 e7 00 61 00 69 00
73 00'
Read         : 'f\x00r\x00a\x00n\x00\x00a\x00i\x00s\x00'
```

В режиме replace недействительные байты заменяются последовательностью символов \uFFFD, которой соответствует официальный подстановочный символ Unicode в виде ромба с белым вопросительным знаком на черном фоне.

```
$ python3 codecs_decode_error.py replace

Original     : 'français'
File contents: b'ff fe 66 00 72 00 61 00 6e 00 e7 00 61 00 69 00
73 00'
Read         : 'f\x00r\x00a\x00n\x00\x00a\x00i\x00s\x00'
```

6.10.5. Преобразование кодировок

Несмотря на то что большинство приложений работает с данными типа str, используя внутренние возможности Python, которые обеспечивают декодирование и кодирование данных как часть операций ввода-вывода, иногда полезно иметь возможность изменить кодировку файла, не привязываясь к промежуточному формату. Функция EncodedFile() получает дескриптор открытого файла, используя одну кодировку, и оборачивает его классом, который преобразует данные в другую кодировку при выполнении над ними операций ввода-вывода.

Листинг 6.103. codecs_encodedfile.py

```
from codecs_to_hex import to_hex

import codecs
import io

# "Сырая" версия исходных данных
data = 'français'

# Кодирование вручную с использованием кодировки UTF-8
utf8 = data.encode('utf-8')
print('Start as UTF-8   :', to_hex(utf8, 1))

# Задание выходного буфера и обертывание его
# классом EncodedFile
output = io.BytesIO()
encoded_file = codecs.EncodedFile(output, data_encoding='utf-8',
                                  file_encoding='utf-16')
encoded_file.write(utf8)

# Извлечение содержимого буфера в виде байтовой
# строки в кодировке UTF-16
utf16 = output.getvalue()
print('Encoded to UTF-16:', to_hex(utf16, 2))

# Задание другого буфера для чтения данных UTF-16
```

```
# и обертывание его другим классом EncodedFile
buffer = io.BytesIO(utf16)
encoded_file = codecs.EncodedFile(buffer, data_encoding='utf-8',
                                  file_encoding='utf-16')

# Чтение версии данных в кодировке UTF-8
recoded = encoded_file.read()
print('Back to UTF-8    :', to_hex(recoded, 1))
```

В этом примере демонстрируется чтение и запись данных с использованием отдельных дескрипторов, возвращенных функцией `EncodedFile()`. Независимо от того, используется ли дескриптор для чтения или для записи, значение `file_encoding` всегда ссылается на кодировку, используемую открытым дескриптором файла, переданным в качестве первого аргумента, тогда как значение `data_encoding` ссылается на кодировку, используемую данными, которые пропускаются через вызовы `read()` и `write()`.

```
$ python3 codecs_encodedfile.py

Start as UTF-8   : b'66 72 61 6e c3 a7 61 69 73'
Encoded to UTF-16: b'fffe 6600 7200 6100 6e00 e700 6100 6900
7300'
Back to UTF-8    : b'66 72 61 6e c3 a7 61 69 73'
```

6.10.6. Другие кодировки

Несмотря на то что в большинстве предыдущих примеров использовались кодировки Unicode, модуль `codecs` может работать не только с ними. Например, Python включает кодеки для работы с кодировками base64, bzip2, ROT-13, ZIP и многими другими.

Листинг 6.104. `codecs_rot13.py`

```
import codecs
import io

buffer = io.StringIO()
stream = codecs.getwriter('rot_13')(buffer)

text = 'abcdefghijklmnopqrstuvwxyz'

stream.write(text)
stream.flush()

print('Original:', text)
print('ROT-13 :', buffer.getvalue())
```

Любое преобразование, выражаемое в виде функции, которая получает один входной аргумент и возвращает байтовую строку или строку Unicode, может быть зарегистрировано в качестве кодека. Для кодека `'rot_13'` входным аргументом должна быть строка Unicode; на выходе также должна быть строка Unicode.

```
$ python3 codecs_rot13.py

Original: abcdefghijklmnopqrstuvwxyz
ROT-13  : nopqrstuvwxyzabcdefghijklm
```

Использование модуля codecs для обертывания потока данных предоставляет более простой интерфейс, чем работа непосредственно с модулем zlib (раздел 8.1).

Листинг 6.105. codecs_zlib.py

```
import codecs
import io

from codecs_to_hex import to_hex

buffer = io.BytesIO()
stream = codecs.getwriter('zlib')(buffer)

text = b'abcdefghijklmnopqrstuvwxyz\n' * 50

stream.write(text)
stream.flush()

print('Original length :', len(text))
compressed_data = buffer.getvalue()
print('ZIP compressed :', len(compressed_data))

buffer = io.BytesIO(compressed_data)
stream = codecs.getreader('zlib')(buffer)

first_line = stream.readline()
print('Read first line :', repr(first_line))

uncompressed_data = first_line + stream.read()
print('Uncompressed :', len(uncompressed_data))
print('Same :', text == uncompressed_data)
```

Не все алгоритмы сжатия или кодирования поддерживают чтение данных порциями через потоковый интерфейс с помощью функций readline() или read(), поскольку они должны находить конец сжатого сегмента для распаковки данных. Если программа не может сохранить полный распакованный набор данных в памяти, то вместо модуля codecs следует использовать возможности инкрементного доступа библиотеки сжатия.

```
$ python3 codecs_zlib.py

Original length : 1350
ZIP compressed  : 48
Read first line : b'abcdefghijklmnopqrstuvwxyz\n'
Uncompressed    : 1350
Same            : True
```

6.10.7. Инкрементное кодирование

Некоторые кодировки, особенно bz2 и zlib, могут в значительной мере изменять размер потока данных, с которым они работают. В случае крупных наборов данных эти кодировки работают эффективнее в инкрементном режиме, обрабатывая по одной небольшой порции данных за раз. Именно для этих целей предназначен API IncrementalEncoder/IncrementalDecoder.

Листинг 6.106. codecs_incremental_bz2.py

```
import codecs
import sys

from codecs_to_hex import to_hex

text = b'abcdefghijklmnopqrstuvwxyz\n'
repetitions = 50

print('Text length :', len(text))
print('Repetitions :', repetitions)
print('Expected len:', len(text) * repetitions)

# Выполнить многократное кодирование текста для
# создания большого объема данных
encoder = codecs.getincrementalencoder('bz2')()
encoded = []

print()
print('Encoding:', end=' ')
last = repetitions - 1
for i in range(repetitions):
    en_c = encoder.encode(text, final=(i == last))
    if en_c:
        print('\nEncoded : {} bytes'.format(len(en_c)))
        encoded.append(en_c)
    else:
        sys.stdout.write('.')

all_encoded = b''.join(encoded)
print()
print('Total encoded length:', len(all_encoded))
print()

# Декодирование байтовой строки по одному байту за раз
decoder = codecs.getincrementaldecoder('bz2')()
decoded = []

print('Decoding:', end=' ')
for i, b in enumerate(all_encoded):
    final = (i + 1) == len(text)
    c = decoder.decode(bytes([b]), final)
    if c:
        print('\nDecoded : {} characters'.format(len(c)))
```

```
        print('Decoding:', end=' ')
        decoded.append(c)
    else:
        sys.stdout.write('.')
print()

restored = b''.join(decoded)

print()
print('Total uncompressed length:', len(restored))
```

Каждый раз, когда данные передаются кодеру или декодеру, его внутреннее состояние обновляется. По достижении определенного состояния (определяемого кодом) происходит возврат данных и сброс состояния. До этого момента вызовы `encode()` и `decode()` не возвращают данные. Когда передана последняя порция данных, для аргумента `final` должно быть установлено значение `True`, извещающее кодек о необходимости выталкивания на диск данных, оставшихся в буфере.

```
$ python3 codecs_incremental_bz2.py

Text length : 27
Repetitions : 50
Expected len: 1350

Encoding: ..............................................
Encoded : 99 bytes

Total encoded length: 99

Decoding: ..........................................................
.....................
Decoded : 1350 characters
Decoding: ..........

Total uncompressed length: 1350
```

6.10.8. Unicode и сетевой обмен данными

Сетевые сокеты — это байтовые потоки, и в отличие от стандартных потоков ввода-вывода они по умолчанию не поддерживают преобразование данных. Как следствие, программы, которым необходимо передавать или получать данные Unicode по сети, должны самостоятельно позаботиться о преобразовании данных в байты, прежде чем записывать их в сокет. В следующем примере сервер пытается возвращать переданные ему данные отправителю.

Листинг 6.107. `codecs_socket_fail.py`

```
import sys
import socketserver


class Echo(socketserver.BaseRequestHandler):
```

```
    def handle(self):
        # Получить байты и вернуть их клиенту
        data = self.request.recv(1024)
        self.request.send(data)
        return


if __name__ == '__main__':
    import codecs
    import socket
    import threading

    address = ('localhost', 0)  # позволить ядру назначить порт
    server = socketserver.TCPServer(address, Echo)
    ip, port = server.server_address  # Какой порт назначен?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # превратить поток в поток-демон
    t.start()

    # Подключение к серверу
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Отправка данных
    # НЕПРАВИЛЬНО: Not encoded first!
    text = 'français'
    len_sent = s.send(text)

    # Получение ответа
    response = s.recv(len_sent)
    print(repr(response))

    # Освобождение ресурсов
    s.close()
    server.socket.close()
```

Данные можно было преобразовывать в байты перед каждым вызовом метода `send()`, но поскольку этого не было сделано, попытка вызова `send()` приводит к ошибке.

```
$ python3 codecs_socket_fail.py

Traceback (most recent call last):
  File "codecs_socket_fail.py", line 43, in <module>
    len_sent = s.send(text)
TypeError: a bytes-like object is required, not 'str'
```

Использование метода `makefile()` с целью получения дескриптора наподобие файлового для сокета и последующее его обертывание объектом потокового чтения или записи гарантирует выполнение необходимого преобразования строк Unicode на входе и выходе сокета.

Листинг 6.108. codecs_socket.py

```
import sys
import socketserver


class Echo(socketserver.BaseRequestHandler):

    def handle(self):
        """Получить байты и вернуть их клиенту.

        Декодировать данные не нужно, поскольку они не
        используются.

        """
        data = self.request.recv(1024)
        self.request.send(data)


class PassThrough:

    def __init__(self, other):
        self.other = other

    def write(self, data):
        print('Writing :', repr(data))
        return self.other.write(data)

    def read(self, size=-1):
        print('Reading :', end=' ')
        data = self.other.read(size)
        print(repr(data))
        return data

    def flush(self):
        return self.other.flush()

    def close(self):
        return self.other.close()


if __name__ == '__main__':
    import codecs
    import socket
    import threading

    address = ('localhost', 0)  # позволить ядру назначить порт
    server = socketserver.TCPServer(address, Echo)
    ip, port = server.server_address  # Какой порт назначен?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # превратить поток в поток-демон
    t.start()
```

```
    # Подключение к серверу
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Обертывание сокета объектом чтения или записи
    read_file = s.makefile('rb')
    incoming = codecs.getreader('utf-8')(PassThrough(read_file))
    write_file = s.makefile('wb')
    outgoing = codecs.getwriter('utf-8')(PassThrough(write_file))

    # Отправка данных
    text = 'français'
    print('Sending :', repr(text))
    outgoing.write(text)
    outgoing.flush()

    # Получение ответа
    response = incoming.read()
    print('Received:', repr(response))

    # Освобождение ресурсов
    s.close()
    server.socket.close()
```

В этом примере класс `PassThrough` используется для демонстрации того, что данные кодируются перед их отправкой и декодируются после их получения клиентом.

```
$ python3 codecs_socket.py

Sending : 'français'
Writing : b'fran\xc3\xa7ais'
Reading : b'fran\xc3\xa7ais'
Reading : b''
Received: 'français'
```

6.10.9. Определение нестандартной кодировки

Поскольку в состав Python входит большое количество стандартных кодеков, вероятность того, что приложению понадобится определять собственные нестандартные кодеры или декодеры, довольно мала. Но если этот шаг окажется необходимым, то его можно упростить, воспользовавшись несколькими базовыми классами, предоставляемыми модулем `codecs`.

Эффективное использование этой возможности предполагает знание природы преобразований, выполняемых в процессе кодирования данных. В примерах, приведенных в этом разделе, будет использоваться кодирование, заключающееся в инвертировании регистра букв. Для этого воспользуемся следующей простой функцией, выполняющей указанное преобразование входной строки.

Листинг 6.109. codecs_invertcaps.py

```
import string


def invertcaps(text):
    """Возвращает новую строку, в которой регистр всех
    букв изменен на обратный.
    """
    return ''.join(
        c.upper() if c in string.ascii_lowercase
        else c.lower() if c in string.ascii_uppercase
        else c
        for c in text
    )


if __name__ == '__main__':
    print(invertcaps('ABCdef'))
    print(invertcaps('abcDEF'))
```

В данном случае кодер и декодер представлены одной функцией (такая же ситуация имеет место в случае кодировки ROT-13).

```
$ python3 codecs_invertcaps.py

abcDEF
ABCdef
```

Эта реализация проста и понятна, но она неэффективна, особенно при работе с большими текстовыми строками. К счастью, модуль codecs включает некоторые вспомогательные функции, предназначенные для создания кодеков на основе таблиц символов, таких как кодек, обращающий регистр букв. А в основе кодирования с помощью таблицы символов лежат два словаря. Кодирующая таблица преобразует значения символов из входной строки в выходные байтовые значения, тогда как декодирующая таблица решает обратную задачу. Прежде всего следует создать декодирующую таблицу, а затем преобразовать ее в кодирующую с помощью функции make_encoding_map(). Эти таблицы используются C-функциями charmap_encode() и charmap_decode() для эффективного преобразования их входных данных.

Листинг 6.110. codecs_invertcaps_charmap.py

```
import codecs
import string

# Отобразить каждый символ на самого себя
decoding_map = codecs.make_identity_dict(range(256))

# Создать список пар порядковых значений для букв
# в верхнем и нижнем регистрах
pairs = list(zip(
    [ord(c) for c in string.ascii_lowercase],
```

```
    [ord(c) for c in string.ascii_uppercase],
))

# Изменить отображение для преобразования букв
# верхнего регистра в нижний и наоборот
decoding_map.update({
    upper: lower
    for (lower, upper)
    in pairs
})
decoding_map.update({
    lower: upper
    for (lower, upper)
    in pairs
})

# Создать отдельную кодирующую таблицу
encoding_map = codecs.make_encoding_map(decoding_map)

if __name__ == '__main__':
    print(codecs.charmap_encode('abcDEF', 'strict',
                                encoding_map))
    print(codecs.charmap_decode(b'abcDEF', 'strict',
                                decoding_map))
    print(encoding_map == decoding_map)
```

Несмотря на то что кодирующая и декодирующая таблицы для обращения регистра букв совпадают, в общем случае это не всегда так. Функция `make_encoding_map()` распознает ситуации, когда несколько входных символов преобразуются в один и тот же выходной байт, и помечает кодируемое значение меткой `None` как неопределенное.

```
$ python3 codecs_invertcaps_charmap.py

(b'ABCdef', 6)
('ABCdef', 6)
True
```

Кодовые таблицы кодера и декодера поддерживают все стандартные методы обработки ошибок, о которых ранее упоминалось, поэтому выполнять какую-либо дополнительную работу по согласованию с этой частью API не потребуется.

Листинг 6.111. `codecs_invertcaps_error.py`

```
import codecs
from codecs_invertcaps_charmap import encoding_map

text = 'pi: \u03c0'

for error in ['ignore', 'replace', 'strict']:
    try:
        encoded = codecs.charmap_encode(
            text, error, encoding_map)
```

```
    except UnicodeEncodeError as err:
        encoded = str(err)
    print('{:7}: {}'.format(error, encoded))
```

Поскольку кодовая точка для числа ϖ отсутствует в кодировочной таблице, в режиме обработки ошибок `strict` возбуждается исключение.

```
$ python3 codecs_invertcaps_error.py

ignore : (b'PI: ', 5)
replace: (b'PI: ?', 5)
strict : 'charmap' codec can't encode character '\u03c0' in
position 4: character maps to <undefined>
```

Определив таблицы для кодирования и декодирования символов, следует настроить несколько дополнительных классов и зарегистрировать схему кодирования. Метод `register()` добавляет поисковую функцию в реестр, чтобы модуль `codecs` мог найти ее, когда пользовать захочет использовать эту кодировку. Поисковая функция должна получать единственный строковый аргумент, содержащий название кодировки, и возвращать объект `CodecInfo`, если кодировка известна, или значение `None`, когда это не так.

Листинг 6.112. `codecs_register.py`

```
import codecs
import encodings


def search1(encoding):
    print('search1: Searching for:', encoding)
    return None


def search2(encoding):
    print('search2: Searching for:', encoding)
    return None


codecs.register(search1)
codecs.register(search2)

utf8 = codecs.lookup('utf-8')
print('UTF-8:', utf8)

try:
    unknown = codecs.lookup('no-such-encoding')
except LookupError as err:
    print('ERROR:', err)
```

Можно зарегистрировать несколько поисковых функций, и каждая из них будет вызываться по очереди до тех пор, пока одна из них не возвратит объект `CodecInfo` или их список не будет исчерпан. Внутренней поисковой функции, зарегистрированной модулем `codecs`, известно, как загружать стандартные кодеки

наподобие UTF-8 из модуля encodings, поэтому их имена никогда не передаются пользовательским поисковым функциям.

```
$ python3 codecs_register.py

UTF-8: <codecs.CodecInfo object for encoding utf-8 at
0x1007773a8>
search1: Searching for: no-such-encoding
search2: Searching for: no-such-encoding
ERROR: unknown encoding: no-such-encoding
```

Экземпляр CodecInfo, возвращенный поисковой функцией, сообщает модулю codecs о том, как кодировать и декодировать данные, используя различные поддерживаемые механизмы: без сохранения состояния, инкрементный и потоковый. Модуль codecs включает базовые классы, облегчающие настройку таблиц кодировки. В следующем примере все эти разрозненные части сводятся воедино для регистрации поисковой функции, которая возвращает экземпляр CodecInfo, сконфигурированный для кодека, обращающего регистр букв.

Листинг 6.113. codecs_invertcaps_register.py

```
import codecs

from codecs_invertcaps_charmap import encoding_map, decoding_map


class InvertCapsCodec(codecs.Codec):
    "Кодер/декодер без сохранения состояния"

    def encode(self, input, errors='strict'):
        return codecs.charmap_encode(input, errors, encoding_map)

    def decode(self, input, errors='strict'):
        return codecs.charmap_decode(input, errors, decoding_map)


class InvertCapsIncrementalEncoder(codecs.IncrementalEncoder):
    def encode(self, input, final=False):
        data, nbytes = codecs.charmap_encode(input,
                                             self.errors,
                                             encoding_map)
        return data


class InvertCapsIncrementalDecoder(codecs.IncrementalDecoder):
    def decode(self, input, final=False):
        data, nbytes = codecs.charmap_decode(input,
                                             self.errors,
                                             decoding_map)
        return data


class InvertCapsStreamReader(InvertCapsCodec,
```

```
                              codecs.StreamReader):
    pass


class InvertCapsStreamWriter(InvertCapsCodec,
                              codecs.StreamWriter):
    pass


def find_invertcaps(encoding):
    """Возвращает кодек для 'invertcaps' (обращение регистра
    букв).
    """
    if encoding == 'invertcaps':
        return codecs.CodecInfo(
            name='invertcaps',
            encode=InvertCapsCodec().encode,
            decode=InvertCapsCodec().decode,
            incrementalencoder=InvertCapsIncrementalEncoder,
            incrementaldecoder=InvertCapsIncrementalDecoder,
            streamreader=InvertCapsStreamReader,
            streamwriter=InvertCapsStreamWriter,
        )
    return None


codecs.register(find_invertcaps)

if __name__ == '__main__':

    # Кодер/декодер без сохранения состояния
    encoder = codecs.getencoder('invertcaps')
    text = 'abcDEF'
    encoded_text, consumed = encoder(text)
    print('Encoded "{}" to "{}", consuming {} characters'.format(
        text, encoded_text, consumed))

    # Объект записи потока
    import io
    buffer = io.BytesIO()
    writer = codecs.getwriter('invertcaps')(buffer)
    print('StreamWriter for io buffer: ')
    print('  writing "abcDEF"')
    writer.write('abcDEF')
    print('  buffer contents: ', buffer.getvalue())

    # Инкрементный декодер
    decoder_factory = codecs.getincrementaldecoder('invertcaps')
    decoder = decoder_factory()
    decoded_text_parts = []
    for c in encoded_text:
        decoded_text_parts.append(
            decoder.decode(bytes([c]), final=False)
```

```
    )
decoded_text_parts.append(decoder.decode(b'', final=True))
decoded_text = ''.join(decoded_text_parts)
print('IncrementalDecoder converted {!r} to {!r}'.format(
    encoded_text, decoded_text))
```

Базовым классом для кодера и декодера без сохранения состояния является класс `Codec`. Переопределите методы `encode()` и `decode()` в новой реализации (в данном случае посредством вызовов `charmap_encode()` и `charmap_decode()` соответственно). Каждый метод должен возвращать кортеж, содержащий преобразованные данные и количество извлеченных входных байтов или символов. Удобно то, что функции `charmap_encode()` и `charmap_decode()` возвращают эту информацию, не требуя вмешательства.

Классы `IncrementalEncoder` и `IncrementalDecoder` служат базовыми для инкрементных интерфейсов. Методы `encode()` и `decode()` инкрементных классов определены таким образом, чтобы они возвращали только фактически обработанные данные. Информация о буферизации сохраняется во внутреннем состоянии. Для кодировки, обращающей регистр букв, буферизация не требуется (используется отображение "один к одному"). Для кодировок с переменным размером вывода, зависящим от типа обрабатываемых данных, таких как алгоритмы сжатия, более подходящими базовыми классами являются `BufferedIncrementalEncoder` и `BufferedIncrementalDecoder`, поскольку они управляют необработанной порцией входных данных.

Классы `StreamReader` и `StreamWriter` также нуждаются в методах `encode()` и `decode()`. Поскольку ожидается, что они будут возвращать то же значение, что и версия на основе класса `Codec`, в реализации может быть использовано множественное наследование.

```
$ python3 codecs_invertcaps_register.py

Encoded "abcDEF" to "b'ABCdef'", consuming 6 characters
StreamWriter for io buffer:
  writing "abcDEF"
  buffer contents:  b'ABCdef'
IncrementalDecoder converted b'ABCdef' to 'abcDEF'
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `codecs`[13].
- `locale` (раздел 15.2). Управление параметрами локализации, обеспечивающими адаптацию программ к особенностям культуры определенной страны.
- `io` (раздел 6.11). Модуль `io` включает классы-обертки для файлов и потоков, которые также управляют кодированием и декодированием данных.
- `socketserver` (раздел 11.5). Более полный пример эхо-сервера, приведенный в связи с обсуждением модуля `socketserver`.
- `encodings`. Пакет стандартной библиотеки, содержащий реализации кодеров и декодеров, которые предоставляет Python.

[13] https://docs.python.org/3.5/library/codecs.html

- **PEP 100**[14]. *Python Unicode Integration*.
- *Unicode HOWTO*[15]. Официальное руководство по использованию стандарта Unicode в Python.
- *Text vs. Data Instead of Unicode vs. 8-bit*[16]. Раздел статьи "What's New in Python 3.0" в справочном руководстве Python, в котором описаны изменения в подходе к обработке текста, реализованные в линейке продуктов Python 3.*x*.
- *Python Unicode Objects*[17] (Fredrik Lundh). Статья, в которой обсуждается использование наборов символов, отличных от набора ASCII, в версии Python 2.0.
- *How to Use UTF-8 with Python*[18] (Evan Jones). Краткое руководство по работе с текстом в кодировке Unicode, включающее рассмотрение данных XML и маркера порядка следования байтов.
- *On the Goodness of Unicode*[19] (Tim Bray). Введение в интернационализацию и методы работы с Unicode.
- *On Character Strings*[20] (Tim Bray). Исторический обзор методов обработки строк в языках программирования.
- *Characters vs. Bytes*[21] (Tim Bray). Часть I статьи для программистов, посвященной современным методам обработки символьных строк. В этой части рассматривается представление в машинной памяти текста в форматах, отличных от байтового представления ASCII.
- Википедия: *Порядок байтов*[22]. Обсуждение порядка следования байтов.
- *W3C XML Entity Definitions for Characters*[23]. Спецификация XML-представления ссылок на символы, которые нельзя представить в доступных кодировках.

6.11. `io`: инструменты для работы с текстовыми, двоичными и "сырыми" потоками ввода-вывода

Модуль `io` реализует классы, обеспечивающие работу встроенной функции `open()` в операциях файлового ввода-вывода. Эти классы разбиты на составные части таким образом, чтобы их можно было комбинировать разными способами для различных целей, – например? для того, чтобы можно было записывать данные Unicode в сетевой сокет.

---

[14] www.python.org/dev/peps/pep-0100

[15] https://docs.python.org/3/howto/unicode.html

[16] https://docs.python.org/3.0/whatsnew/3.0.html#text-vs-data-instead-of-unicode-vs-8-bit

[17] http://effbot.org/zone/unicode-objects.htm

[18] http://evanjones.ca/python-utf8.html

[19] www.tbray.org/ongoing/When/200x/2003/04/06/Unicode

[20] www.tbray.org/ongoing/When/200x/2003/04/13/Strings

[21] www.tbray.org/ongoing/When/200x/2003/04/26/UTF

[22] https://ru.wikipedia.org/wiki/Порядок_байтов

[23] www.w3.org/TR/xml-entity-names/

6.11.1. Потоки, отображаемые в памяти

Класс `StringIO` предоставляет удобные средства для работы с текстом в памяти с использованием файлового API (например, `read()`, `write()`). В некоторых случаях использование класса `StringIO` для создания больших строк может обеспечить экономию ресурсов по сравнению с другими методами конкатенации строк. Кроме того, буферы потоков, отображаемых в памяти, удобно использовать в целях тестирования, когда запись в реальный файл на диске, может замедлять работу тестового набора.

Некоторые стандартные примеры использования буферов приведены ниже.

Листинг 6.114. `io_stringio.py`

```
import io

# Запись в буфер
output = io.StringIO()
output.write('This goes into the buffer. ')
print('And so does this.', file=output)

# Извлечение записанного значения
print(output.getvalue())
output.close() # освободить память буфера

# Инициализация буфера чтения
input = io.StringIO('Inital value for read buffer')

# Чтение из буфера
print(input.read())
```

В данном примере используется метод `read()`, но для этого подходят также методы `readline()` и `readlines()`. Класс `StringIO` предоставляет метод `seek()`, обеспечивающий перемещение в пределах буфера в процессе чтения данных, что может оказаться полезным для перемещения в обратном направлении, если используется алгоритм анализа с просмотром только вперед.

```
$ python3 io_stringio.py

This goes into the buffer. And so does this.

Inital value for read buffer
```

Для работы с "сырыми" байтами вместо текста Unicode следует использовать класс `BytesIO`.

Листинг 6.115. `io_bytesio.py`

```
import io

# Запись в буфер
output = io.BytesIO()
output.write('This goes into the buffer. '.encode('utf-8'))
output.write('ÁÇĒ'.encode('utf-8'))
```

```
# Извлечение записанного значения
print(output.getvalue())
output.close() # освободить память буфера

# Инициализация буфера чтения
input = io.BytesIO(b'Inital value for read buffer')

# Чтение из буфера
print(input.read())
```

Значения, записываемые в экземпляр `BytesIO`, должны иметь тип `bytes`, а не `str`.

```
$ python3 io_bytesio.py

b'This goes into the buffer. \xc3\x81\xc3\x87\xc3\x8a'
b'Inital value for read buffer'
```

6.11.2. Обертывание байтовых потоков для текстовых данных

"Сырые" байтовые потоки, например сокеты, можно обертывать слоем, обеспечивающим кодирование и декодирование текстовых строк, упрощая их использование с текстовыми данными. Класс `TextIOWrapper` поддерживает как чтение, так и запись данных. Аргумент `write_through` отключает буферизацию и сразу же выталкивает все данные, записываемые в оболочку, в базовый буфер.

Листинг 6.116. `io_textiowrapper.py`

```
import io

# Запись в буфер
output = io.BytesIO()
wrapper = io.TextIOWrapper(
    output,
    encoding='utf-8',
    write_through=True,
)
wrapper.write('This goes into the buffer. ')
wrapper.write('ÁÇÊ')

# Извлечение записанного значения
print(output.getvalue())

output.close()  # освободить память буфера

# Инициализация буфера чтения
input = io.BytesIO(
    b'Inital value for read buffer with unicode characters ' +
    'ÁÇÊ'.encode('utf-8')
)
wrapper = io.TextIOWrapper(input, encoding='utf-8')
```

```
# Чтение из буфера
print(wrapper.read())
```

В этом примере экземпляр `BytesIO` используется в качестве потока. В примерах для модулей `bz2` (раздел 8.3), `http.server` (раздел 12.5) и `subprocess` (раздел 10.1) демонстрируется использование класса `TextIOWrapper` без привлечения других типов файловых объектов.

```
$ python3 io_textiowrapper.py

b'This goes into the buffer. \xc3\x81\xc3\x87\xc3\x8a'
Inital value for read buffer with unicode characters ÁÇÊ
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `io`[24].
- Раздел 12.2.3. Использование метода `detach()` класса `TextIOWrapper` для управления классом-оберткой отдельно от обернутого им сокета.
- *Efficient String Concatenation in Python*[25]. Сравнение различных методов объединения строк.

[24] https://docs.python.org/3.5/library/io.html

[25] www.skymind.com/%7Eocrow/python_string/

Глава 7

Постоянное хранение и обмен данными

С организацией постоянного хранения данных, обеспечивающего возможность их дальнейшего использования даже после того, как программа завершила работу и объекты были удалены из памяти, связаны два аспекта: преобразование формата данных в процессе обмена между оперативной памятью и хранилищем и работа с преобразованными данными, находящимися в хранилище. Стандартная библиотека включает ряд модулей, позволяющих решать эти задачи в различных ситуациях.

Для преобразования объектов в формат, обеспечивающий их передачу или сохранение на носителе (процесс, известный как *сериализация* данных), предназначены два модуля. Чаще всего постоянное хранение данных организуют с помощью модуля `pickle` (раздел 7.1) — для этих целей он интегрируется с другими модулями стандартной библиотеки, в частности, `shelve`, которые выполняют всю фактическую работу по сохранению сериализованных данных. Однако в случае веб-приложений в основном используют модуль `json`, поскольку он лучше интегрируется с существующими веб-службами хранения данных.

Определившись с форматом, в который должен быть преобразован находящийся в памяти объект для его последующего сохранения, необходимо выбрать тип долговременного хранилища. В случае данных, не требующих индексированного доступа, вполне можно обойтись так называемыми *плоскими* файлами, обеспечивающими минимальную структуризацию сохраняемых данных, в которые сериализуемые объекты записываются один за другим. Python включает коллекцию модулей, предназначенных для хранения пар "ключ–значение" в простой базе данных с использованием различных вариантов формата DBM, когда требуются возможности доступа по индексам.

Наиболее непосредственный способ воспользоваться всеми преимуществами формата DBM обеспечивает модуль `shelve` (раздел 7.2). Достаточно открыть файл хранилища `shelve`, и им можно пользоваться примерно так же, как словарем. Объекты, сохраненные в базе данных, автоматически извлекаются и сохраняются, не требуя выполнения каких-либо дополнительных действий со стороны вызывающего кода.

Одним из недостатков модуля `shelve` является то, что в случае использования интерфейса, предлагаемого по умолчанию, невозможно предсказать, какой именно формат DBM будет использован: модуль `shelve` выбирает его, исходя из того, какие библиотеки доступны в системе, в которой создается база данных. Выбор формата не имеет значения, если приложение не собирается обмениваться файлами базы данных с другими хостами, на которых могут быть установлены другие библиотеки, но если переносимость данных является одним из требований, то для гарантии выбора определенного формата можно использовать один из классов, включенных в модуль.

В случае веб-приложений, которые уже работают с данными в формате JSON, следует использовать модули `json` (раздел 12.9) и `dbm` (раздел 4.3), обеспечиваю-

щие другой механизм сохранения данных. Непосредственное использование модуля dbm требует выполнения немного большего объема работы по сравнению с модулем shelve, поскольку объекты не будут автоматически воссоздаваться при доступе к значениям в базе данных в силу того, что ключи и значения базы данных DBM являются строками.

Модуль sqlite3 (раздел 7.4), входящий в состав большинства дистрибутивов Python, обеспечивает организацию хранения данных в более сложных конфигурациях, чем простые пары "ключ–значение". База данных может храниться в памяти или в локальном файле, и все обращения к ней выполняются в одном и том же процессе, что приводит к отсутствию задержек, связанных с сетевым обменом. Компактность sqlite3 делает этот модуль особенно пригодным для внедрения в настольные приложения или веб-приложения.

Также имеются другие модули, предназначенные для анализа более формализованных форматов, что может быть использовано для обмена данными между программами на Python и приложениями, написанными на других языках. Библиотека xml.etree.ElementTree (раздел 7.5) предоставляет инструменты для выполнения синтаксического анализа XML-документов и обеспечивает несколько рабочих режимов для различных приложений. Кроме инструментария для синтаксического анализа, библиотека ElementTree включает интерфейс для создания корректно сформированных XML-документов на основе объектов, хранящихся в памяти. Модуль csv (раздел 7.6) обеспечивает чтение и запись табличных данных в форматах, создаваемых электронными таблицами или приложениями баз данных, что делает его весьма удобным для групповой загрузки данных или преобразования данных из одного формата в другой.

7.1. pickle: сериализация объектов

Модуль pickle реализует алгоритм преобразования произвольного объекта Python в последовательность байтов. Этот процесс также называют *сериализацией* объекта. После этого байтовый поток, представляющий объект, может быть передан или сохранен и впоследствии реконструирован для создания нового объекта с теми же характеристиками.

Предупреждение

В документации модуля pickle ясно сказано о том, что никакие гарантии безопасности не предоставляются. В действительности восстановление данных, сериализованных с помощью модуля pickle, может выполнить любой код. Будьте внимательны, когда используете модуль pickle для внутрипроцессного обмена данными или сохранения данных, и не доверяйте данным, которые не могут быть верифицированы как безопасные. Пример безопасного способа верификации источника данных, сериализованных с помощью модуля pickle, приведен в описании модуля hmac (раздел 9.2).

7.1.1. Кодирование и декодирование строковых данных

В первом из приведенных ниже примеров структура данных преобразуется в строку с помощью функции dumps(), и полученная строка выводится на консоль. Используемая структура данных полностью состоит из встроенных типов. Как будет показано в следующем примере, таким способом могут быть сериализованы экземпляры любого класса.

Листинг 7.1. pickle_string.py

```
import pickle
import pprint

data = [{'a': 'A', 'b': 2, 'c': 3.0}]
print('DATA:', end=' ')
pprint.pprint(data)

data_string = pickle.dumps(data)
print('PICKLE: {!r}'.format(data_string))
```

По умолчанию модуль pickle записывает данные в двоичном формате, обеспечивающем наибольшую совместимость при обмене данными между программами, использующими версию Python 3.

```
$ python3 pickle_string.py

DATA: [{'a': 'A', 'b': 2, 'c': 3.0}]
PICKLE: b'\x80\x03]q\x00}q\x01(X\x01\x00\x00\x00cq\x02G@\x08\x00
\x00\x00\x00\x00\x00X\x01\x00\x00\x00bq\x03K\x02X\x01\x00\x00\x0
0aq\x04X\x01\x00\x00\x00Aq\x05ua.'
```

Сериализованные данные можно записать в файл, сокет, канал или другое место. Позже можно будет прочитать и десериализовать данные для конструирования нового объекта с теми же значениями.

Листинг 7.2. pickle_unpickle.py

```
import pickle
import pprint

data1 = [{'a': 'A', 'b': 2, 'c': 3.0}]
print('BEFORE: ', end=' ')
pprint.pprint(data1)

data1_string = pickle.dumps(data1)

data2 = pickle.loads(data1_string)
print('AFTER : ', end=' ')
pprint.pprint(data2)

print('SAME? :', (data1 is data2))
print('EQUAL?:', (data1 == data2))
```

Вновь созданный объект эквивалентен исходному, но не является тем же объектом.

```
$ python3 pickle_unpickle.py

BEFORE:  [{'a': 'A', 'b': 2, 'c': 3.0}]
AFTER :  [{'a': 'A', 'b': 2, 'c': 3.0}]
SAME? : False
EQUAL?: True
```

7.1.2. Работа с потоками

В дополнение к методам `dumps()` и `loads()` модуль `pickle` предоставляет вспомогательные функции для работы с потоками, подобными файлам. В поток можно записать несколько объектов, а затем прочитать их, не зная заранее, сколько объектов было записано и каковы их размеры.

Листинг 7.3. `pickle_stream.py`

```
import io
import pickle
import pprint


class SimpleObject:

    def __init__(self, name):
        self.name = name
        self.name_backwards = name[::-1]
        return


data = []
data.append(SimpleObject('pickle'))
data.append(SimpleObject('preserve'))
data.append(SimpleObject('last'))

# Имитировать файл
out_s = io.BytesIO()

# Записать в поток
for o in data:
    print('WRITING : {} ({})'.format(o.name, o.name_backwards))
    pickle.dump(o, out_s)
    out_s.flush()

# Настроить читаемый поток
in_s = io.BytesIO(out_s.getvalue())

# Прочитать данные
while True:
    try:
        o = pickle.load(in_s)
    except EOFError:
        break
    else:
        print('READ    : {} ({})'.format(
            o.name, o.name_backwards))
```

В этом примере для имитации потоков используются два буфера `BytesIO`. Первый из них получает объекты, сериализованные с помощью модуля `pickle`, и его значение подается во второй буфер, из которого данные читаются с помощью функции `load()`. Простые форматы баз данных также могут использовать сериа-

лизованные данные для сохранения объектов. Одним из примеров подобной реализации может служить модуль shelve (раздел 7.2).

```
$ python3 pickle_stream.py

WRITING : pickle (elkcip)
WRITING : preserve (evreserp)
WRITING : last (tsal)
READ    : pickle (elkcip)
READ    : preserve (evreserp)
READ    : last (tsal)
```

Помимо сохранения данных сериализация объектов с помощью модуля pickle удобна для межпроцессного обмена данными. Например, можно использовать функции os.fork() и os.pipe() для создания рабочих процессов, которые читают инструкции заданий из одного канала и записывают результаты в другой канал. Основная часть кода, управляющая пулом рабочих процессов, раздачей заданий и получением ответов, может использоваться многократно, поскольку объекты заданий и ответов не должны базироваться на каком-то определенном классе. Используя каналы или сокеты, не забывайте сбрасывать буфер после записи каждого объекта, чтобы вытолкнуть данные на другой конец соединения. Повторно используемый менеджер пула рабочих процессов рассматривается при обсуждении модуля multiprocessing (раздел 10.4).

7.1.3. Проблемы реконструирования объектов

При работе с пользовательскими классами сериализуемый с помощью модуля pickle класс должен быть виден в пространстве имен процесса, выполняющего чтение сериализованных объектов. Сериализуются лишь данные экземпляра, но не определение класса. Для нахождения конструктора, с помощью которого из десериализуемых данных создается новый объект, используется имя класса.

Листинг 7.4. pickle_dump_to_file_1.py

```
import pickle
import sys


class SimpleObject:

    def __init__(self, name):
        self.name = name
        l = list(name)
        l.reverse()
        self.name_backwards = ''.join(l)


if __name__ == '__main__':
    data = []
    data.append(SimpleObject('pickle'))
    data.append(SimpleObject('preserve'))
    data.append(SimpleObject('last'))
```

```
    filename = sys.argv[1]

    with open(filename, 'wb') as out_s:
        for o in data:
            print('WRITING: {} ({})'.format(
                o.name, o.name_backwards))
            pickle.dump(o, out_s)
```

Когда вы запустите этот сценарий, он создаст файл на основании имени, предоставленного вами в командной строке в качестве аргумента.

```
$ python3 pickle_dump_to_file_1.py test.dat

WRITING: pickle (elkcip)
WRITING: preserve (evreserp)
WRITING: last (tsal)
```

Простая попытка загрузки результирующих объектов не увенчается успехом.

Листинг 7.5. pickle_load_from_file_1.py

```
import pickle
import pprint
import sys

filename = sys.argv[1]

with open(filename, 'rb') as in_s:
    while True:
        try:
            o = pickle.load(in_s)
        except EOFError:
            break
        else:
            print('READ: {} ({})'.format(
                o.name, o.name_backwards))
```

Эта версия не работает ввиду недоступности класса `SimpleObject`.

```
$ python3 pickle_load_from_file_1.py test.dat

Traceback (most recent call last):
  File "pickle_load_from_file_1.py", line 15, in <module>
    o = pickle.load(in_s)
AttributeError: Can't get attribute 'SimpleObject' on <module '_
_main__' from 'pickle_load_from_file_1.py'>
```

В то же время скорректированной версии, которая импортирует объект `SimpleObject` из исходного сценария, удается завершить операцию. Добавление следующей инструкции в конец списка импорта позволяет сценарию обнаружить класс и сконструировать объект.

```
from pickle_dump_to_file_1 import SimpleObject
```

Теперь выполнение видоизмененного сценария обеспечивает получение желаемого результата.

```
$ python3 pickle_load_from_file_2.py test.dat

READ: pickle (elkcip)
READ: preserve (evreserp)
READ: last (tsal)
```

7.1.4. Объекты, не сериализуемые с помощью модуля pickle

Не все объекты могут быть сериализованы с помощью модуля pickle. Сокеты, дескрипторы файлов, подключения к базам данных и другие объекты с состоянием времени выполнения, которые зависят от операционной системы или другого процесса, могут быть непригодными для сохранения каким-либо разумным способом. Объекты, имеющие подобные непригодные для сериализации атрибуты, могут определить методы __getstate__() и __setstate__() для возврата подмножества состояния экземпляра, подлежащего сериализации.

Метод __getstate__() должен вернуть объект, содержащий внутреннее состояние объекта. Одним из удобных способов представления состояния является словарь, но значением может быть любой объект, поддающийся сериализации с помощью модуля pickle. Состояние сохраняется, а затем передается методу __setstate__() при загрузке сериализованного объекта.

Листинг 7.6. pickle_state.py

```
import pickle


class State:

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return 'State({!r})'.format(self.__dict__)


class MyClass:

    def __init__(self, name):
        print('MyClass.__init__({})'.format(name))
        self._set_name(name)

    def _set_name(self, name):
        self.name = name
        self.computed = name[::-1]
```

```
    def __repr__(self):
        return 'MyClass({!r}) (computed={!r})'.format(
            self.name, self.computed)

    def __getstate__(self):
        state = State(self.name)
        print('__getstate__ -> {!r}'.format(state))
        return state

    def __setstate__(self, state):
        print('__setstate__({!r})'.format(state))
        self._set_name(state.name)


inst = MyClass('name here')
print('Before:', inst)

dumped = pickle.dumps(inst)

reloaded = pickle.loads(dumped)
print('After:', reloaded)
```

В этом примере для хранения внутреннего состояния объекта `MyClass` используется отдельный объект `State`. При загрузке сериализованного экземпляра `MyClass` методу `__setstate__()` передается экземпляр `State`, который используется для инициализации объекта.

```
$ python3 pickle_state.py

MyClass.__init__(name here)
Before: MyClass('name here') (computed='ereh eman')
__getstate__ -> State({'name': 'name here'})
__setstate__(State({'name': 'name here'}))
After: MyClass('name here') (computed='ereh eman')
```

Предупреждение

Если возвращаемое значение равно `False`, то метод `__setstate__()` не вызывается при десериализации объекта.

7.1.5. Циклические ссылки

Протокол модуля `pickle` автоматически управляет циклическими ссылками, поэтому сложные структуры данных не требуют никакой особой организации управления. Взгляните на направленный граф, приведенный на рис. 7.1. Он включает несколько циклических ссылок, однако корректную структуру можно сериализовать, а затем перезагрузить.

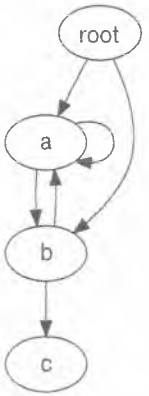

Рис. 7.1. Сериализация структуры данных с циклическими ссылками с помощью модуля `pickle`

Листинг 7.7. `pickle_cycle.py`

```
import pickle


class Node:
    """Простой дираф.
    """
    def __init__(self, name):
        self.name = name
        self.connections = []

    def add_edge(self, node):
        "Создает границу между данным и другим узлом."
        self.connections.append(node)

    def __iter__(self):
        return iter(self.connections)


def preorder_traversal(root, seen=None, parent=None):
    """Функция-генератор, возвращающая границы для графа.
    """
    if seen is None:
        seen = set()
    yield (parent, root)
    if root in seen:
        return
    seen.add(root)
    for node in root:
        recurse = preorder_traversal(node, seen, root)
        for parent, subnode in recurse:
            yield (parent, subnode)
```

```
def show_edges(root):
    "Выводит все границы в графе."
    for parent, child in preorder_traversal(root):
        if not parent:
            continue
        print('{:>5} -> {:>2} ({})'.format(
            parent.name, child.name, id(child)))


# Задание узлов
root = Node('root')
a = Node('a')
b = Node('b')
c = Node('c')

# Добавление границ между ними
root.add_edge(a)
root.add_edge(b)
a.add_edge(b)
b.add_edge(a)
b.add_edge(c)
a.add_edge(a)

print('ORIGINAL GRAPH:')
show_edges(root)

# Сериализация и десериализация графа для
# создания нового набора узлов
dumped = pickle.dumps(root)
reloaded = pickle.loads(dumped)

print('\nRELOADED GRAPH:')
show_edges(reloaded)
```

Перезагруженные узлы не являются тем же объектом, но связи между узлами сохраняются, и перезагружается только один экземпляр объекта с множественными ссылками. В справедливости обоих этих утверждений можно убедиться, проверив значения идентификаторов объектов, возвращаемых функцией `id()` до и после их сериализации и десериализации с помощью модуля `pickle`.

```
$ python3 pickle_cycle.py

ORIGINAL GRAPH:
 root -> a (4315798272)
    a -> b (4315798384)
    b -> a (4315798272)
    b -> c (4315799112)
    a -> a (4315798272)
 root -> b (4315798384)

RELOADED GRAPH:
 root -> a (4315904096)
    a -> b (4315904152)
```

```
    b -> a (4315904096)
    b -> c (4315904208)
    a -> a (4315904096)
 root -> b (4315904152)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `pickle`[1].
- **PEP 3154**[2]. *Pickle protocol version 4.*
- `shelve` (раздел 7.2). Модуль `shelve` использует модуль `pickle` для сохранения данных в базе данных DBM.
- *Pickle: An interesting stack language*[3] (Alexandre Vassalotti). Статья, посвященная использованию модуля `pickle`.

7.2. `shelve`: постоянное хранение объектов

Модуль `shelve` позволяет достаточно просто организовать постоянное хранение объектов Python в тех случаях, когда отсутствует необходимость в использовании реляционных баз данных. Доступ к хранилищу `shelve` (для которого выбрана метафора полки – англ. *shelf*) осуществляется по ключам, как при работе с обычным словарем. Модуль `shelve` сериализует объект с помощью модуля `pickle` и записывает его в базу данных, которая создается и управляется модулем `dbm` (раздел 7.3).

7.2.1. Создание нового хранилища

Проще всего использовать модуль `shelve` посредством класса `Dbfilename Shelf`. В свою очередь для сохранения данных этот класс использует интерфейс `dbm` (раздел 7.3). Класс `DbfilenameShelf` можно использовать либо непосредственно, либо путем вызова функции `shelve.open()`.

Листинг 7.8. `shelve_create.py`

```
import shelve

with shelve.open('test_shelf.db') as s:
    s['key1'] = {
        'int': 10,
        'float': 9.5,
        'string': 'Sample data',
    }
```

Чтобы вновь получить доступ к данным, откройте хранилище и используйте его так, словно оно является словарем.

[1] `https://docs.python.org/3.5/library/pickle.html`

[2] `www.python.org/dev/peps/pep-3154`

[3] `http://peadrop.com/blog/2007/06/18/pickle-an-interesting-stack-language/`

Листинг 7.9. shelve_existing.py

```
import shelve

with shelve.open('test_shelf.db') as s:
    existing = s['key1']

print(existing)
```

Выполнив оба сценария примеров, вы получите следующий вывод.

```
$ python3 shelve_create.py
$ python3 shelve_existing.py

{'string': 'Sample data', 'int': 10, 'float': 9.5}
```

Модуль dbm (раздел 7.3) не поддерживает одновременную запись в одну и ту же базу данных несколькими приложениями одновременно, но поддерживает параллельные обращения к ней по чтению несколькими клиентами. Если клиент не будет изменять хранилище, откройте базу данных только для чтения, передав аргумент flag='r'.

Листинг 7.10. shelve_readonly.py

```
import dbm
import shelve

with shelve.open('test_shelf.db', flag='r') as s:
    print('Existing:', s['key1'])
    try:
        s['key1'] = 'new value'
    except dbm.error as err:
        print('ERROR: {}'.format(err))
```

Если программа попытается изменить базу данных, в то время как она открыта только для чтения, будет сгенерировано исключение. Тип исключения зависит от того, какой модуль базы данных был выбран модулем dbm (раздел 7.3) при ее создании.

```
$ python3 shelve_readonly.py

Existing: {'string': 'Sample data', 'int': 10, 'float': 9.5}
ERROR: cannot add item to database
```

7.2.2. Обратная запись

По умолчанию хранилища, создаваемые модулем shelve, не отслеживают изменения, происходящие во временных объектах. Таким образом, если содержимое объекта, ранее сохраненного в хранилище, изменяется, хранилище необходимо обновить явно путем повторного сохранения в нем данного элемента.

Листинг 7.11. shelve_withoutwriteback.py

```
import shelve

with shelve.open('test_shelf.db') as s:
    print(s['key1'])
    s['key1']['new_value'] = 'this was not here before'

with shelve.open('test_shelf.db', writeback=True) as s:
    print(s['key1'])
```

В этом примере значение, соответствующее ключу 'key1', не сохраняется повторно, в чем можно убедиться, повторно открыв хранилище.

```
$ python3 shelve_create.py
$ python3 shelve_withoutwriteback.py

{'string': 'Sample data', 'int': 10, 'float': 9.5}
{'string': 'Sample data', 'int': 10, 'float': 9.5}
```

Чтобы обеспечить автоматическое обновление хранилища при изменении объектов, временно существующих в памяти компьютера, его необходимо открыть с установленным флагом writeback. Установка этого флага приводит к тому, что хранилище будет запоминать в кеш-памяти все объекты, извлеченные из базы данных. При закрытии хранилища каждый объект, хранящийся в кеше, записывается в базу данных.

Листинг 7.12. shelve_writeback.py

```
import shelve
import pprint

with shelve.open('test_shelf.db', writeback=True) as s:
    print('Initial data:')
    pprint.pprint(s['key1'])

    s['key1']['new_value'] = 'this was not here before'
    print('\nModified:')
    pprint.pprint(s['key1'])
with shelve.open('test_shelf.db', writeback=True) as s:
    print('\nPreserved:')
    pprint.pprint(s['key1'])
```

Несмотря на то что указанная мера позволяет снизить вероятность совершения ошибок программистом и может сделать организацию долговременного хранения объектов более прозрачной, использование режима обратной записи не всегда желательно. Создание кеша при открытии хранилища приводит к дополнительному расходованию памяти и приостанавливает выполнение приложения всякий раз, когда кешированный объект записывается в базу данных. В нее записываются все кешированные объекты, поскольку нет возможности явно указать, какие именно объекты претерпели изменения. Если приложение в основном читает, а не записывает данные, режим обратной записи может ухудшить его производительность.

```
$ python3 shelve_create.py
$ python3 shelve_writeback.py

Initial data:
{'float': 9.5, 'int': 10, 'string': 'Sample data'}

Modified:
{'float': 9.5,
 'int': 10,
 'new_value': 'this was not here before',
 'string': 'Sample data'}

Preserved:
{'float': 9.5,
 'int': 10,
 'new_value': 'this was not here before',
 'string': 'Sample data'}
```

7.2.3. Специализированные типы хранилищ

Во всех предыдущих примерах использовалась реализация хранилища `shelve`, заданная по умолчанию. Использование функции `shelve.open()` вместо непосредственного использования других реализаций — обычный подход, особенно если тип базы данных, используемой для сохранения данных, не имеет значения. Однако иногда тип базы данных играет важную роль. В подобных ситуациях для построения пользовательского решения используйте непосредственно класс `DbfilenameShelf` или `BsdDbShelf`.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `shelve`[4].
- `dbm` (раздел 4.3). Модуль `dbm` создает новую базу данных, находя доступную библиотеку `DBM`.
- `feedcache`[5]. Модуль `feedcache` использует `shelve` в качестве хранилища по умолчанию.
- `shove`[6]. Модуль `shove` реализует аналогичный API с большим количеством внутренних форматов.

7.3. dbm: базы данных Unix с доступом по ключу

Модуль `dbm` — это интерфейс доступа к базам данных в стиле DBM, в которых простые строковые значения служат ключами доступа к записям, содержащим строки. Он использует функцию `whichdb()` для идентификации баз данных, а затем открывает их с помощью соответствующего модуля. Модуль `dbm` используется модулем `shelve` (см. раздел 7.2), который сохраняет объекты в базе данных DBM с помощью модуля `pickle` (см. раздел 7.1).

[4] https://docs.python.org/3.5/library/shelve.html

[5] https://bitbucket.org/dhellmann/feedcache

[6] http://pypi.python.org/pypi/shove/

7.3.1. Типы баз данных

Python поставляется с несколькими модулями для доступа к базам данных в стиле DBM. Выбор реализации по умолчанию зависит от наличия библиотек, доступных в данной системе, и опций, использованных, когда компилировался Python. Наличие отдельных интерфейсов доступа к специализированным реализациям позволяет программам на Python обмениваться данными с программами, написанными на других языках программирования, которые не способны автоматически переключаться между доступными форматами, и записывать переносимые файлы данных, которые могут использоваться на ряде платформ.

7.3.1.1. dbm.gnu

Модуль `dbm.gnu` обеспечивает доступ к версии библиотеки `dbm` из проекта GNU. Он работает точно так же, как другие описанные здесь реализации DBM, отличаясь лишь некоторыми изменениями в поддержке флагов функцией `open()`.

Помимо стандартных флагов `'r'`, `'w'`, `'c'` и `'n'` функция `dbm.gnu.open()` поддерживает ряд дополнительных флагов.

- Флаг `'f'` управляет открытием базы данных в быстром режиме. В этом режиме операции записи в базу данных не синхронизируются.
- Флаг `'s'` управляет открытием базы данных в синхронизированном режиме. Изменения базы данных записываются в файл сразу же, как они вносятся, а не откладываются до момента закрытия базы данных или ее явной синхронизации.
- Флаг `'u'` управляет открытием базы данных в неблокированном состоянии.

7.3.1.2. dbm.ndbm

Модуль `dbm.ndbm` предоставляет интерфейс доступа к различным `ndbm`-реализациям формата `dbm` в Unix, в зависимости от того, какой модуль был сконфигурирован в процессе компиляции. Атрибут `library` модуля идентифицирует используемую реализацию библиотеки `ndbm`, которую сценарию `configure` удалось обнаружить в процессе компиляции модуля расширения.

7.3.1.3. dbm.dumb

Модуль `dbm.dumb` — это переносимая резервная реализация DBM API, которая используется в отсутствие других реализаций. Модуль `dbm.dumb` не требует никаких внешних зависимостей, но работает медленнее, чем большинство других реализаций.

7.3.2. Создание новой базы данных

Формат хранения для новых баз данных выбирается посредством поиска пригодных версий каждого из подмодулей в следующем порядке:

- `dbm.gnu`
- `dbm.ndbm`
- `dbm.dumb`

Функция `open()` получает флаги, обеспечивающие управление файлами базы данных. Чтобы создать новую базу данных, следует указать флаг `'c'`. Флаг `'n'` означает создание новой базы данных с заменой существующего файла.

Листинг 7.13. `dbm_new.py`

```
import dbm

with dbm.open('/tmp/example.db', 'n') as db:
    db['key'] = 'value'
    db['today'] = 'Sunday'
    db['author'] = 'Doug'
```

В этом примере файл всегда инициализируется заново.

```
$ python3 dbm_new.py
```

Функция `whichdb()` сообщает о типе созданного файла.

Листинг 7.14. `dbm_whichdb.py`

```
import dbm

print(dbm.whichdb('/tmp/example.db'))
```

Вывод программы из этого примера будет меняться в зависимости то того, какие модули установлены в системе.

```
$ python3 dbm_whichdb.py

dbm.ndbm
```

7.3.3. Открытие существующей базы данных

Чтобы открыть существующую базу данных, используйте флаг `'r'` (режим только чтения) или `'w'` (режим чтения-записи). Существующие базы данных автоматически передаются функции `whichdb()`, чтобы их можно было идентифицировать. Таким образом, коль скоро файл может быть идентифицирован, он может быть открыт с использованием соответствующего модуля.

Листинг 7.15. `dbm_existing.py`

```
import dbm

with dbm.open('/tmp/example.db', 'r') as db:
    print('keys():', db.keys())
    for k in db.keys():
        print('iterating:', k, db[k])
    print('db["author"] =', db['author'])
```

Открытому файлу соответствует объект `db`, который ведет себя подобно файлу. При добавлении в базу данных новые ключи всегда преобразуются в байтовые строки и возвращаются в виде байтовых строк.

```
$ python3 dbm_existing.py

keys(): [b'key', b'today', b'author']
iterating: b'key' b'value'
iterating: b'today' b'Sunday'
iterating: b'author' b'Doug'
db["author"] = b'Doug'
```

7.3.4. Примеры ошибок

Ключи базы данных должны быть строками.

Листинг 7.16. dbm_intkeys.py

```
import dbm

with dbm.open('/tmp/example.db', 'w') as db:
    try:
        db[1] = 'one'
    except TypeError as err:
        print(err)
```

Передача другого типа приводит к ошибке TypeError.

```
$ python3 dbm_intkeys.py

dbm mappings have bytes or string keys only
```

Значениями должны быть строки или None.

Листинг 7.17. dbm_intvalue.py

```
import dbm

with dbm.open('/tmp/example.db', 'w') as db:
    try:
        db['one'] = 1
    except TypeError as err:
        print(err)
```

Аналогичная ошибка возникает, если значение не является строкой.

```
$ python3 dbm_intvalue.py

dbm mappings have byte or string elements only
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю dbm[7].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля anydbm (раздел A.6.2).

[7] https://docs.python.org/3.5/library/dbm.html

- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `whichdb` (раздел A.6.50).
- `shelve` (см. раздел 7.2). Примеры для модуля `shelve`, в которых для сохранения данных используется модуль `dbm`.

7.4. `sqlite3`: встроенная реляционная база данных

Модуль `sqlite3` реализует интерфейс доступа к внутрипроцессной базе данных SQLite, совместимый с Python DB-API 2.0[8]. База данных SQLite спроектирована таким образом, чтобы ее можно было встраивать в приложения, а не использовать отдельную серверную программу, такую как `MySQL`, `PostgreSQL` или `Oracle`. SQLite – быстрая, тщательно протестированная и гибкая база данных, что делает ее весьма удобной для прототипирования и производственного развертывания в случае некоторых приложений.

7.4.1. Создание базы данных

База данных SQLite хранится в файловой системе в виде одиночного файла. Библиотека управляет доступом к этому файлу, включая его блокирование с целью предотвращения повреждения данных, когда одновременно несколько программ пытаются записать данные в файл. База данных создается при первой попытке доступа к файлу, но ответственность за создание таблицы определений, или схемы базы данных, возлагается на приложение.

В этом примере программа сначала осуществляет поиск файла базы данных, прежде чем открыть его с помощью функции `connect()`, поэтому ей известно, в каких случаях следует создавать схему для новых баз данных.

Листинг 7.18. `sqlite3_createdb.py`

```
import os
import sqlite3

db_filename = 'todo.db'

db_is_new = not os.path.exists(db_filename)

conn = sqlite3.connect(db_filename)

if db_is_new:
    print('Need to create schema')
else:
    print('Database exists; assume schema does, too.')

conn.close()
```

Выполнение этого сценария два раза подряд показывает, что в том случае, когда файл не существует, создается пустой файл.

[8] www.python.org/dev/peps/pep-0249/

```
$ ls *.db

ls: *.db: No such file or directory

$ python3 sqlite3_createdb.py

Need to create schema

$ ls *.db

todo.db

$ python3 sqlite3_createdb.py

Database exists; assume schema does, too.
```

После создания нового файла базы данных следующим шагом является создание схемы для определения таблиц в базе данных. Во всех остальных примерах в этом разделе используется одна и та же схема с таблицами для управления задачами. Подробные сведения о схеме базы данных представлены в табл. 7.1 и 7.2.

Таблица 7.1. Таблица проекта

Столбец	Тип	Описание
`name`	`text`	Имя проекта
`description`	`text`	Длинное описание проекта
`deadline`	`date`	Срок завершения всего проекта

Таблица 7.2. Таблица задачи

Столбец	Тип	Описание
`id`	`number`	Уникальный идентификатор задачи
`priority`	`integer`	Числовые приоритеты (меньшее число означает больший приоритет)
`details`	`text`	Подробное описание задачи
`status`	`text`	Статус задачи (`new`, `pending`, `done` или `canceled`)
`deadline`	`date`	Срок выполнения задачи
`completed_on`	`date`	Дата фактического выполнения задачи
`project`	`text`	Имя проекта для данной задачи

Инструкции *языка описания данных* (Data Definition Language — DDL), с помощью которого создаются таблицы, представлены в следующем листинге.

Листинг 7.19. `todo_schema.sql`

```
-- Схема для примеров приложения to-do

-- Проекты - это высокоуровневые операции, состоящие из задач
create table project (
    name        text primary key,
    description text,
```

```
    deadline    date
);

-- Задачи - это шаги, которые должны быть выполнены
-- для завершения проекта
create table task (
    id          integer primary key autoincrement not null,
    priority    integer default 1,
    details text,
    status text,
    deadline date,
    completed_on date,
    project text not null references project(name)
);
```

Для выполнения инструкций по созданию DDL-схемы можно использовать метод `executescript()` объекта `Connection`.

Листинг 7.20. `sqlite3_create_schema.py`

```
import os
import sqlite3

db_filename = 'todo.db'
schema_filename = 'todo_schema.sql'

db_is_new = not os.path.exists(db_filename)

with sqlite3.connect(db_filename) as conn:
    if db_is_new:
        print('Creating schema')
        with open(schema_filename, 'rt') as f:
            schema = f.read()
        conn.executescript(schema)

        print('Inserting initial data')

        conn.executescript("""
        insert into project (name, description, deadline)
        values ('pymotw', 'Python Module of the Week',
                '2016-11-01');

        insert into task (details, status, deadline, project)
        values ('write about select', 'done', '2016-04-25',
                'pymotw');

        insert into task (details, status, deadline, project)
        values ('write about random', 'waiting', '2016-08-22',
                'pymotw');

        insert into task (details, status, deadline, project)
        values ('write about sqlite3', 'active', '2017-07-31',
                'pymotw');
        """)
```

```
    else:
        print('Database exists, assume schema does, too.')
```

Вслед за созданием таблиц выполняются инструкции вставки, с помощью которых создается пробный проект и относящиеся к нему задачи. Содержимое базы данных можно проверить, введя в командной строке команду `sqlite3`.

```
$ rm -f todo.db
$ python3 sqlite3_create_schema.py

Creating schema
Inserting initial data

$ sqlite3 todo.db 'select * from task'

1|1|write about select|done|2016-04-25||pymotw
2|1|write about random|waiting|2016-08-22||pymotw
3|1|write about sqlite3|active|2017-07-31||pymotw
```

7.4.2. Извлечение данных

Чтобы извлечь значения, сохраненные в таблице задач, создайте объект `Cursor` для соединения с базой данных. Курсор обеспечивает согласованное представление текущей обрабатываемой записи для различных типов данных и является основным средством взаимодействия с транзакционными базами данных наподобие SQLite.

Листинг 7.21. `sqlite3_select_tasks.py`

```
import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
    select id, priority, details, status, deadline from task
    where project = 'pymotw'
    """)

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

Выполнение запроса осуществляется в два этапа. Прежде всего, следует вызвать метод `execute()` объекта курсора, чтобы сообщить базы данных, какие данные должны быть выбраны. После этого следует извлечь результирующий набор, вызвав метод `fetchall()`. Возвращаемое значение представляет собой последовательность кортежей, которая содержит значения столбцов, включенных в оператор `select` запроса.

```
$ python3 sqlite3_select_tasks.py

1 [1] write about select          [done    ] (2016-04-25)
2 [1] write about random          [waiting ] (2016-08-22)
3 [1] write about sqlite3         [active  ] (2017-07-31)
```

Результаты можно извлекать по одному за один раз с помощью метода `fetchone()` или блоками фиксированного размера с помощью метода `fetchmany()`.

Листинг 7.22. sqlite3_select_variations.py

```
import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
    select name, description, deadline from project
    where name = 'pymotw'
    """)
    name, description, deadline = cursor.fetchone()

    print('Project details for {} ({})\n  due {}'.format(
        description, name, deadline))

    cursor.execute("""
    select id, priority, details, status, deadline from task
    where project = 'pymotw' order by deadline
    """)

    print('\nNext 5 tasks:')
    for row in cursor.fetchmany(5):
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

Значение, передаваемое методу `fetchmany()`, определяет максимальное количество возвращаемых записей. Если последовательность доступных записей меньше этого значения, то размер возвращаемой последовательности будет меньше максимального.

```
$ python3 sqlite3_select_variations.py

Project details for Python Module of the Week (pymotw)
  due 2016-11-01

Next 5 tasks:
 1 [1] write about select          [done    ] (2016-04-25)
 2 [1] write about random          [waiting ] (2016-08-22)
 3 [1] write about sqlite3         [active  ] (2017-07-31)
```

7.4.3. Метаданные запроса

В соответствии со спецификацией DB-API 2.0, после вызова метода `execute()` объект `Cursor` должен установить свой атрибут `description`, используемый для хранения информации о данных, которые будут возвращаться методами `fetch`. Спецификация API определяет значение `description` как последовательность кортежей, содержащих имя столбца, тип данных, отображаемый размер, внутренний размер, точность, шкалу и флаг, указывающий на то, приемлемы ли значения `null`.

Листинг 7.23. `sqlite3_cursor_description.py`

```
import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
    select * from task where project = 'pymotw'
    """)

    print('Task table has these columns:')
    for colinfo in cursor.description:
        print(colinfo)
```

Поскольку `sqlite3` не навязывает задание типа или размера данных, добавляемых в базу данных, значениями заполняется лишь столбец `name`.

```
$ python3 sqlite3_cursor_description.py

Task table has these columns:
('id', None, None, None, None, None, None)
('priority', None, None, None, None, None, None)
('details', None, None, None, None, None, None)
('status', None, None, None, None, None, None)
('deadline', None, None, None, None, None, None)
('completed_on', None, None, None, None, None, None)
('project', None, None, None, None, None, None)
```

7.4.4. Объекты `Row`

По умолчанию значения, возвращаемые методами `fetch` в качестве строк результата запроса, являются кортежами. Ответственность за корректное использование порядка следования столбцов в запросе и извлечение индивидуальных значений из кортежа возлагается на вызывающий код. В тех случаях, когда количество значений в запросе велико или данные рассредоточены по библиотеке, обычно проще работать с объектами и получать доступ к значением по именам их столбцов. Это позволяет со временем изменить количество и порядок следования элементов кортежа при редактировании запроса и уменьшает вероятность разрушения кода, зависящего от результатов запроса.

Объекты подключения имеют свойство row_factory, с помощью которого вызывающий код может управлять типом объектов, создаваемых для представления строк результирующего набора запроса. Кроме того, модуль sqlite3 включает класс Row, предназначенный для использования в качестве фабрики строк результирующего набора. Доступ к значениям столбцов посредством экземпляров Row может осуществляться с использованием индекса или имени столбца.

Листинг 7.24. sqlite3_row_factory.py

```
import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    # Изменить атрибут row_factory, чтобы использовать
    # объект Row
    conn.row_factory = sqlite3.Row

    cursor = conn.cursor()

    cursor.execute("""
    select name, description, deadline from project
    where name = 'pymotw'
    """)
    name, description, deadline = cursor.fetchone()

    print('Project details for {} ({})\n  due {}'.format(
        description, name, deadline))

    cursor.execute("""
    select id, priority, status, deadline, details from task
    where project = 'pymotw' order by deadline
    """)

    print('\nNext 5 tasks:')
    for row in cursor.fetchmany(5):
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            row['id'], row['priority'], row['details'],
            row['status'], row['deadline'],
        ))
```

Этот пример представляет собой переписанную версию примера sqlite3_select_variations.py, в которой вместо кортежей используются экземпляры Row. Строки таблицы проекта (project) по-прежнему выводятся с использованием доступа к значениям столбцов в соответствии с их позициями, однако в инструкциях print для задач (tasks) используется обращение к значениям по ключам. Вследствие этого изменение порядка следования столбцов в запросе никак не влияет на конечный результат.

```
$ python3 sqlite3_row_factory.py

Project details for Python Module of the Week (pymotw)
```

```
  due 2016-11-01

Next 5 tasks:
 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

7.4.5. Использование переменных в запросах

Использование запросов, определенных посредством встроенных в программу литеральных строк, лишено гибкости. Например, если в базу данных добавляется другой проект, то запрос, отображающий первые пять задач проекта, должен быть обновлен таким образом, чтобы он был применим к любому проекту. Одним из способов повышения гибкости запросов является создание SQL-инструкций с желаемым запросом посредством объединения значений в Python. Однако с созданием строки запроса таким способом связаны определенные риски, которых следует избегать. Ошибки, допущенные при экранировании специальных символов в переменных частях запроса, могут приводить к ошибкам при анализе SQL-инструкций или, что еще хуже, создавать уязвимости, которыми могут воспользоваться злоумышленники для атак путем внедрения SQL-кода (так называемая *SQL-инъекция* или *SQL-внедрение*), позволяющих выполнять любые SQL-запросы к базе данных.

Правильный способ включения динамических значений в запросы заключается в использовании переменных, передаваемых методу `execute()` вместе с SQL-инструкцией. При выполнении такой инструкции подстановочные символы заменяются значением переменной. Использование переменных вместо вставки произвольных значений до того, как они будут анализироваться, позволяет избежать атак SQL-внедрением, поскольку ненадежные значения никак не смогут повлиять на результаты анализа инструкций SQL. База данных SQLite поддерживает две формы параметризованных инструкций, обеспечивающих замену подстановочных символов: позиционную и именованную.

7.4.5.1. Позиционные параметры

Вопросительный знак (`?`) означает позиционный аргумент, передаваемый методу `execute()` в качестве элемента кортежа.

Листинг 7.25. `sqlite3_argument_positional.py`

```
import sqliteimport sqlite3
import sys

db_filename = 'todo.db'
project_name = sys.argv[1]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    query = """
    select id, priority, details, status, deadline from task
    where project = ?
```

```
    """

    cursor.execute(query, (project_name,))

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

Аргумент командной строки безопасно передается запросу в качестве позиционного аргумента, и некорректные данные не смогут повредить базу данных.

```
$ python3 sqlite3_argument_positional.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

7.4.5.2. Именованные параметры

В случае более сложных запросов со многими параметрами или параметрами, повторяющимися несколько раз, целесообразно использовать именованные параметры. Именованным параметрам должно предшествовать двоеточие (например, `:param_name`).

Листинг 7.26. `sqlite3_argument_named.py`

```
import sqlite3
import sys

db_filename = 'todo.db'
project_name = sys.argv[1]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    query = """
    select id, priority, details, status, deadline from task
    where project = :project_name
    order by deadline, priority
    """

    cursor.execute(query, {'project_name': project_name})

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

Ни позиционные, ни именованные параметры не нуждаются в заключении в кавычки или экранировании, поскольку они обрабатываются специальным образом анализатором запросов.

```
$ python3 sqlite3_argument_named.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

Параметры запроса можно использовать с операторами select, insert и update. Они могут появляться в любом месте запроса, в котором допустимо использование литерального значения.

Листинг 7.27. sqlite3_argument_update.py

```
import sqlite3
import sys

db_filename = 'todo.db'
id = int(sys.argv[1])
status = sys.argv[2]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
    query = "update task set status = :status where id = :id"
    cursor.execute(query, {'status': status, 'id': id})
```

В этой инструкции update используются два именованных параметра. Значение id служит для нахождения строки, подлежащей изменению, а значение status записывается в таблицу.

```
$ python3 sqlite3_argument_update.py 2 done
$ python3 sqlite3_argument_named.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [done    ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

7.4.6. Групповая загрузка

Чтобы применить одну и ту же SQL-инструкцию к большому набору данных, используйте метод executemany(). Этот метод удобно использовать для загрузки данных, поскольку он позволяет избежать организации цикла по входным данным в Python и предоставить базовой библиотеке возможность применить оптимизацию цикла. В следующем примере программа читает список задач из CSV-файла, в котором значения разделены запятыми, используя модуль csv (раздел 7.6), и загружает их в базу данных.

Листинг 7.28. sqlite3_load_csv.py

```
import csv
import sqlite3
import sys

db_filename = 'todo.db'
```

```
data_filename = sys.argv[1]

SQL = """
insert into task (details, priority, status, deadline, project)
values (:details, :priority, 'active', :deadline, :project)
"""

with open(data_filename, 'rt') as csv_file:
    csv_reader = csv.DictReader(csv_file)

    with sqlite3.connect(db_filename) as conn:
        cursor = conn.cursor()
        cursor.executemany(SQL, csv_reader)
```

Файл *tasks.csv* содержит следующие данные.

```
deadline,project,priority,details
2016-11-30,pymotw,2,"finish reviewing markup"
2016-08-20,pymotw,2,"revise chapter intros"
2016-11-01,pymotw,1,"subtitle"
```

Программа выводит следующие результаты.

```
$ python3 sqlite3_load_csv.py tasks.csv
$ python3 sqlite3_argument_named.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 5 [2] revise chapter intros     [active  ] (2016-08-20)
 2 [1] write about random        [done    ] (2016-08-22)
 6 [1] subtitle                  [active  ] (2016-11-01)
 4 [2] finish reviewing markup   [active  ] (2016-11-30)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

7.4.7. Описание новых типов столбцов

База данных SQLite предоставляет собственную поддержку столбцов, содержащих целые числа, числа с плавающей точкой и текст. Модуль `sqlite3` автоматически преобразует данные этих типов в значения, которые могут сохраняться в базе данных и извлекаться из нее по мере необходимости. Целочисленные значения загружаются из базы данных в переменные типа `int` или `long`, в зависимости от величины значения. Текст сохраняется и извлекается в виде значений типа `str`, если только этот тип не был изменен с помощью атрибута `text_factory` объекта `Connection`.

Несмотря на то что база данных SQLite внутренне поддерживает лишь несколько типов данных, модуль `sqlite3` поддерживает средства, обеспечивающие возможность определения пользовательских типов, что позволяет приложениям Python сохранять в столбцах любые типы данных. Преобразование в типы, не входящие в число поддерживаемых по умолчанию, задается с помощью флага `detect_types` объекта `Connection`. Если столбец был объявлен с использованием желаемого типа, когда создавалась таблица, используйте константу `PARSE_DECLTYPES`.

Листинг 7.29. sqlite3_date_types.py

```
import sqlite3
import sys

db_filename = 'todo.db'

sql = "select id, details, deadline from task"


def show_deadline(conn):
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()
    cursor.execute(sql)
    row = cursor.fetchone()
    for col in ['id', 'details', 'deadline']:
        print('  {:<8}  {!r:<26} {}'.format(
            col, row[col], type(row[col])))
    return


print('Without type detection:')
with sqlite3.connect(db_filename) as conn:
    show_deadline(conn)

print('\nWith type detection:')
with sqlite3.connect(db_filename,
                     detect_types=sqlite3.PARSE_DECLTYPES,
                     ) as conn:
    show_deadline(conn)
```

Модуль `sqlite3` предоставляет конвертеры для столбцов даты и времени, используя соответственно классы `date` и `datetime` из модуля `datetime` (см. раздел 4.2) для представления значений в Python. Оба конвертера активизируются автоматически при включенном обнаружении типов.

```
$ python3 sqlite3_date_types.py

Without type detection:
  id        1                          <class 'int'>
  details   'write about select'       <class 'str'>
  deadline  '2016-04-25'               <class 'str'>

With type detection:
  id        1                          <class 'int'>
  details   'write about select'       <class 'str'>
  deadline  datetime.date(2016, 4, 25) <class 'datetime.date'>
```

Прежде чем определять новый тип, необходимо зарегистрировать две функции. *Адаптер* принимает объект Python в качестве входных данных и возвращает байтовую строку, которую можно сохранить в базе данных. *Конвертер* получает строку из базы данных и возвращает объект Python. Для определения адаптера используйте функцию `register_adapter()`, а для определения конвертера — функцию `register_converter()`.

Листинг 7.30. `sqlite3_custom_type.py`

```
import pickle
import sqlite3

db_filename = 'todo.db'


def adapter_func(obj):
    """Преобразует данные, хранящиеся в памяти,
       в представление хранилища.
    """
    print('adapter_func({})\n'.format(obj))
    return pickle.dumps(obj)


def converter_func(data):
    """Преобразует данные, находящиеся в хранилище, в
       представление данных, хранящихся в памяти.
    """
    print('converter_func({!r})\n'.format(data))
    return pickle.loads(data)


class MyObj:

    def __init__(self, arg):
        self.arg = arg

    def __str__(self):
        return 'MyObj({!r})'.format(self.arg)


# Регистрация функций для манипулирования типами
sqlite3.register_adapter(MyObj, adapter_func)
sqlite3.register_converter("MyObj", converter_func)

# Создание объектов для сохранения. Используйте список
# кортежей, чтобы последовательность можно было передать
# непосредственно методу executemany().
to_save = [
    (MyObj('this is a value to save'),),
    (MyObj(42),),
]

with sqlite3.connect(
        db_filename,
        detect_types=sqlite3.PARSE_DECLTYPES) as conn:
    # Создание таблицы со столбцами типа "MyObj"
    conn.execute("""
    create table if not exists obj (
        id    integer primary key autoincrement not null,
        data  MyObj
```

```
    )
    """)
    cursor = conn.cursor()

    # Вставка объектов в базу данных
    cursor.executemany("insert into obj (data) values (?)",
                       to_save)

    # Запрос объектов, только что сохраненных в базе данных
    cursor.execute("select id, data from obj")
    for obj_id, obj in cursor.fetchall():
        print('Retrieved', obj_id, obj)
        print('  with type', type(obj))
        print()
```

В этом примере для сохранения объекта в виде строки, которая может быть сохранена в базе данных, используется модуль `pickle` (см. раздел 7.1) – полезный прием, пригодный для сохранения произвольных объектов, однако не позволяющий выполнять запросы на основании атрибутов объекта. В случае больших объемов данных более полезной будет библиотека SQLAlchemy[9], использующая технологию *объектно-реляционного отображения* и сохраняющая значения атрибутов в их собственных столбцах.

```
$ python3 sqlite3_custom_type.py

adapter_func(MyObj('this is a value to save'))

adapter_func(MyObj(42))

converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\
x03\x00\x00\x00argq\x03X\x17\x00\x00\x00this is a value tosaveq\x0
4sb.')
converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\
x03\x00\x00\x00argq\x03K*sb.')

Retrieved 1 MyObj('this is a value to save')
  with type <class '__main__.MyObj'>

Retrieved 2 MyObj(42)
  with type <class '__main__.MyObj'>
```

7.4.8. Определение типов столбцов

Существуют два типа источников информации, касающейся типов данных для запроса. Для идентификации типа реального столбца можно использовать определение исходной таблицы, как было продемонстрировано ранее. Альтернативный вариант заключается в том, чтобы включить спецификатор типа в предложение `select` самого запроса, используя форму "*имя* `[`*тип*`]`".

[9] www.sqlalchemy.org

Листинг 7.31. `sqlite3_custom_type_column.py`

```
import pickle
import sqlite3

db_filename = 'todo.db'


def adapter_func(obj):
    """Преобразует данные, хранящиеся в памяти,
       в представление хранилища.
    """
    print('adapter_func({})\n'.format(obj))
    return pickle.dumps(obj)


def converter_func(data):
    """Преобразует данные, находящиеся в хранилище, в
       представление данных, хранящихся в памяти.
    """
    print('converter_func({!r})\n'.format(data))
    return pickle.loads(data)


class MyObj:

    def __init__(self, arg):
        self.arg = arg

    def __str__(self):
        return 'MyObj({!r})'.format(self.arg)


# Регистрация функций для манипулирования типами
sqlite3.register_adapter(MyObj, adapter_func)
sqlite3.register_converter("MyObj", converter_func)

# Создание объектов для сохранения. Используется список
# кортежей, чтобы последовательность можно было передать
# непосредственно методу executemany().
to_save = [
    (MyObj('this is a value to save'),),
    (MyObj(42),),
]

with sqlite3.connect(
        db_filename,
        detect_types=sqlite3.PARSE_COLNAMES) as conn:
    # Создание таблицы со столбцами типа "text"
    conn.execute("""
    create table if not exists obj2 (
        id    integer primary key autoincrement not null,
        data  text
    )
    """)
```

```
    cursor = conn.cursor()

    # Вставка объектов в базу данных
    cursor.executemany("insert into obj2 (data) values (?)",
                       to_save)

    # Запрос объектов, только что сохраненных в базе данных,
    # с использованием спецификатора типа для преобразования
    # текста в объекты
    cursor.execute(
        'select id, data as "pickle [MyObj]" from obj2',
    )
    for obj_id, obj in cursor.fetchall():
        print('Retrieved', obj_id, obj)
        print('  with type', type(obj))
        print()
```

Используйте для атрибута `detect_types` флаг `PARSE_COLNAMES`, если вместо определения исходной таблицы используется тип, включенный в запрос.

```
$ python3 sqlite3_custom_type_column.py

adapter_func(MyObj('this is a value to save'))

adapter_func(MyObj(42))

converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\
x03\x00\x00\x00argq\x03X\x17\x00\x00\x00this is a value to saveq\x
04sb.')

converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\
x03\x00\x00\x00argq\x03K*sb.')

Retrieved 1 MyObj('this is a value to save')
  with type <class '__main__.MyObj'>

Retrieved 2 MyObj(42)
  with type <class '__main__.MyObj'>
```

7.4.9. Транзакции

Одной из важных особенностей реляционных баз данных является возможность использования транзакций для поддержания согласованного внутреннего состояния. В режиме транзакций можно внести некоторые изменения посредством одного подключения, не влияя на работу других пользователей до тех пор, пока эти изменения не будут зафиксированы и фактически записаны в базу данных.

7.4.9.1. Сохранение изменений

Изменения, вносимые в базу данных с помощью инструкций `insert` и `update`, должны сохраняться явно посредством вызова метода `commit()`. Это требование предоставляет приложению возможность вносить сразу несколько изменений, которые можно будет сохранить атомарно, а не по очереди. При таком подходе

удается избежать ситуаций, когда частичные изменения ощущаются всеми клиентами, одновременно подключенными к базе данных.

Для наблюдения за эффектом вызова метода commit() воспользуемся приведенной ниже программой, в которой создается одновременно несколько подключений к базе данных. Сначала с помощью одного соединения в базу данных вставляется новая строка, после чего делаются две попытки прочитать эту же строку с помощью других соединений.

Листинг 7.32. sqlite3_transaction_commit.py

```
import sqlite3

db_filename = 'todo.db'


def show_projects(conn):
    cursor = conn.cursor()
    cursor.execute('select name, description from project')
    for name, desc in cursor.fetchall():
        print('  ', name)


with sqlite3.connect(db_filename) as conn1:
    print('Before changes:')
    show_projects(conn1)

    # Вставка в один курсор
    cursor1 = conn1.cursor()
    cursor1.execute("""
    insert into project (name, description, deadline)
    values ('virtualenvwrapper', 'Virtualenv Extensions',
            '2011-01-01')
    """)

    print('\nAfter changes in conn1:')
    show_projects(conn1)

    # Выборка из другого соединения до фиксации
    # изменений, выполненных в первом соединении
Select from another connection, without committing first
    print('\nBefore commit:')
    with sqlite3.connect(db_filename) as conn2:
        show_projects(conn2)

    # Фиксация изменений с последующей
    # выборкой из другого соединения
    conn1.commit()
    print('\nAfter commit:')
    with sqlite3.connect(db_filename) as conn3:
        show_projects(conn3)
```

Если функция show_projects() вызывается до фиксации изменений, выполненных посредством соединения conn1, то результат зависит от того, какое соединение используется. Так как изменение было выполнено с использованием со-

единения conn1, оно остается видимым для этого соединения, но невидимым для соединения conn2. После фиксации изменений вставленная строка становится доступной для нового соединения conn3.

```
$ python3 sqlite3_transaction_commit.py

Before changes:
    pymotw

After changes in conn1:
    pymotw
    virtualenvwrapper

Before commit:
    pymotw

After commit:
    pymotw
    virtualenvwrapper
```

7.4.9.2. Отмена изменений

От незафиксированных изменений можно полностью отказаться, используя метод rollback(). Как правило, методы commit() и rollback() вызываются из разных частей блока try:except, обеспечивающего отмену изменений ("откат") при возникновении ошибок.

Листинг 7.33. sqlite3_transaction_rollback.py

```
import sqlite3

db_filename = 'todo.db'


def show_projects(conn):
    cursor = conn.cursor()
    cursor.execute('select name, description from project')
    for name, desc in cursor.fetchall():
        print('  ', name)


with sqlite3.connect(db_filename) as conn:

    print('Before changes:')
    show_projects(conn)

    try:

        # Вставить строку
        cursor = conn.cursor()
        cursor.execute("""delete from project
                       where name = 'virtualenvwrapper'
                       """)
```

```
        # Отобразить результаты
        print('\nAfter delete:')
        show_projects(conn)

        # Искусственно возбудить исключение
        raise RuntimeError('simulated error')

    except Exception as err:
        # Отказаться от изменений
        print('ERROR:', err)
        conn.rollback()

    else:
        # Сохранить изменения
        conn.commit()

    # Отобразить результаты
    print('\nAfter rollback:')
    show_projects(conn)
```

После вызова метода `rollback()` изменения не остаются в базе данных.

```
$ python3 sqlite3_transaction_rollback.py

Before changes:
    pymotw
    virtualenvwrapper

After delete:
    pymotw
ERROR: simulated error

After rollback:
    pymotw
    virtualenvwrapper
```

7.4.10. Уровни изоляции

Модуль `sqlite3` поддерживает три режима блокировки, называемые *уровнями изоляции*. Они позволяют управлять выбором метода, используемого для предотвращения внесения изменений, которые приводят к несовместимости состояний базы данных в различных соединениях. Уровень изоляции устанавливается посредством передачи строки в качестве аргумента, задающего значение атрибута `isolation_level` при открытии соединения, так что разные соединения могут иметь разные уровни изоляции.

В приведенной ниже программе продемонстрировано влияние уровней изоляции на очередность событий в потоках, использующих отдельные соединения для подключения к одной и той же базе данных. Создаются четыре потока: два из них записывают изменения в базу данных посредством обновления существующих строк, тогда как два других пытаются прочитать все строки из таблицы `task`.

Листинг 7.34. sqlite3_isolation_levels.py

```
import logging
import sqlite3
import sys
import threading
import time

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s (%(threadName)-10s) %(message)s',
)

db_filename = 'todo.db'
isolation_level = sys.argv[1]


def writer():
    with sqlite3.connect(
            db_filename,
            isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        cursor.execute('update task set priority = priority + 1')
        logging.debug('waiting to synchronize')
        ready.wait()  # synchronize threads
        logging.debug('PAUSING')
        time.sleep(1)
        conn.commit()
        logging.debug('CHANGES COMMITTED')


def reader():
    with sqlite3.connect(
            db_filename,
            isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        logging.debug('waiting to synchronize')
        ready.wait()  # synchronize threads
        logging.debug('wait over')
        cursor.execute('select * from task')
        logging.debug('SELECT EXECUTED')
        cursor.fetchall()
        logging.debug('results fetched')


if __name__ == '__main__':
    ready = threading.Event()

    threads = [
        threading.Thread(name='Reader 1', target=reader),
        threading.Thread(name='Reader 2', target=reader),
        threading.Thread(name='Writer 1', target=writer),
        threading.Thread(name='Writer 2', target=writer),
    ]
```

```
    [t.start() for t in threads]

    time.sleep(1)
    logging.debug('setting ready')
    ready.set()

    [t.join() for t in threads]
```

Потоки синхронизируются с помощью объекта `Event` из модуля `threading` (раздел. 10.3). Функция `writer()` подключается к базе данных и вносит в нее изменения, но не фиксирует их до тех пор, пока не наступит событие синхронизации.

7.4.10.1. DEFERRED

По умолчанию используется уровень изоляции `DEFERRED`. В этом режиме база данных блокируется, но только с момента попытки изменить ее. Именно этот режим использовался во всех предыдущих примерах.

```
$ python3 sqlite3_isolation_levels.py DEFERRED

2016-08-20 17:46:26,972 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:26,972 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:26,973 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:27,977 (MainThread) setting ready
2016-08-20 17:46:27,979 (Reader 1  ) wait over
2016-08-20 17:46:27,979 (Writer 1  ) PAUSING
2016-08-20 17:46:27,979 (Reader 2  ) wait over
2016-08-20 17:46:27,981 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:27,982 (Reader 1  ) results fetched
2016-08-20 17:46:27,982 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:27,982 (Reader 2  ) results fetched
2016-08-20 17:46:28,985 (Writer 1  ) CHANGES COMMITTED
2016-08-20 17:46:29,043 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:29,043 (Writer 2  ) PAUSING
2016-08-20 17:46:30,044 (Writer 2  ) CHANGES COMMITTED
```

7.4.10.2. IMMEDIATE

В режиме `IMMEDIATE` база данных блокируется, как только делается попытка ее изменения, и до тех пор, пока транзакция не завершится, другие курсоры лишаются возможности изменять базу данных. Этот режим целесообразно использовать для баз данных со сложной структурой записей, при работе с которыми выполняются в основном операции чтения, но не записи данных, поскольку во время выполнения транзакции операции чтения не блокируются.

```
$ python3 sqlite3_isolation_levels.py IMMEDIATE

2016-08-20 17:46:30,121 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:30,121 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:30,123 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:31,122 (MainThread) setting ready
```

```
2016-08-20 17:46:31,122 (Reader 1  ) wait over
2016-08-20 17:46:31,122 (Reader 2  ) wait over
2016-08-20 17:46:31,122 (Writer 1  ) PAUSING
2016-08-20 17:46:31,124 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:31,124 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:31,125 (Reader 2  ) results fetched
2016-08-20 17:46:31,125 (Reader 1  ) results fetched
2016-08-20 17:46:32,128 (Writer 1  ) CHANGES COMMITTED
2016-08-20 17:46:32,199 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:32,199 (Writer 2  ) PAUSING
2016-08-20 17:46:33,200 (Writer 2  ) CHANGES COMMITTED
```

7.4.10.3. EXCLUSIVE

В исключительном режиме база данных блокируется для всех объектов чтения и записи. К этому режиму следует прибегать лишь в ситуациях, в которых важна производительность, так как каждое соединение, обладающее правами исключительного доступа, блокирует всех других пользователей.

```
$ python3 sqlite3_isolation_levels.py EXCLUSIVE

2016-08-20 17:46:33,320 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:33,320 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:33,324 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:34,323 (MainThread) setting ready
2016-08-20 17:46:34,323 (Reader 1  ) wait over
2016-08-20 17:46:34,323 (Writer 2  ) PAUSING
2016-08-20 17:46:34,323 (Reader 2  ) wait over
2016-08-20 17:46:35,327 (Writer 2  ) CHANGES COMMITTED
2016-08-20 17:46:35,368 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:35,368 (Reader 2  ) results fetched
2016-08-20 17:46:35,369 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:35,369 (Reader 1  ) results fetched
2016-08-20 17:46:35,385 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:35,385 (Writer 1  ) PAUSING
2016-08-20 17:46:36,386 (Writer 1  ) CHANGES COMMITTED
```

Как только первый объект записи начинает вносить изменения, объекты чтения и второй объект записи блокируются до завершения транзакции. Вызов `sleep()` вводит искусственную задержку в поток, выполняющий запись, чтобы более отчетливо продемонстрировать блокирование других соединений.

7.4.10.4. Автоматическая фиксация транзакций

Установка значения `None` для параметра `isolation_level` соединения включает режим автоматической фиксации транзакций. В этом режиме изменения фиксируются после каждого успешного вызова `execute()` по завершении инструкции. Режим автоматической фиксации хорошо подходит для коротких транзакций, например для вставки небольшого объема данных в одну таблицу. При этом длительность блокирования базы данных сводится к минимуму, что значительно уменьшает вероятность возникновения ситуации гонки.

В сценарии sqlite3_autocommit.py удален явный вызов метода commit() и для уровня изоляции установлено значение None. Во всем остальном используемый в нем подход аналогичен тому, который использовался в сценарии sqlite3_isolation_levels.py. В то же время вывод, создаваемый этим сценарием, отличается, поскольку оба потока записи заканчивают работу до того, как объекты чтения обращаются с запросами к базе данных.

```
$ python3 sqlite3_autocommit.py

2016-08-20 17:46:36,451 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:36,451 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:36,455 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:36,456 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:37,452 (MainThread) setting ready
2016-08-20 17:46:37,452 (Reader 1  ) wait over
2016-08-20 17:46:37,452 (Writer 2  ) PAUSING
2016-08-20 17:46:37,452 (Reader 2  ) wait over
2016-08-20 17:46:37,453 (Writer 1  ) PAUSING
2016-08-20 17:46:37,453 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:37,454 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:37,454 (Reader 1  ) results fetched
2016-08-20 17:46:37,454 (Reader 2  ) results fetched
```

7.4.11. Базы данных в памяти

SQLite поддерживает управление базой данной, хранящейся в оперативной памяти компьютера, а не на диске. Базы данных, хранимые в памяти, удобно использовать для автоматизированного тестирования, когда не требуется сохранение базы данных в промежутках времени между тестовыми запусками программы, или для проведения экспериментов со схемой или другими объектами базы данных. Чтобы открыть базу данных в памяти, следует использовать строку ':memory:' вместо имени файла при создании объекта Connection. Каждое соединение типа ':memory:' создает независимый экземпляр базы данных, поэтому изменения, создаваемые одним курсором, не влияют на другие соединения.

7.4.12. Экспорт содержимого базы данных

Содержимое базы данных в памяти может быть сохранено с помощью метода iterdump() объекта Connection. Итератор, возвращенный методом iterdump(), возвращает последовательность строк, которые совместно образуют SQL-инструкции, воссоздающие состояние базы данных.

Листинг 7.35. sqlite3_iterdump.py

```
import sqlite3

schema_filename = 'todo_schema.sql'

with sqlite3.connect(':memory:') as conn:
    conn.row_factory = sqlite3.Row
```

```
    print('Creating schema')
    with open(schema_filename, 'rt') as f:
        schema = f.read()
    conn.executescript(schema)

    print('Inserting initial data')
    conn.execute("""
    insert into project (name, description, deadline)
    values ('pymotw', 'Python Module of the Week',
            '2010-11-01')
    """)
    data = [
        ('write about select', 'done', '2010-10-03',
         'pymotw'),
        ('write about random', 'waiting', '2010-10-10',
         'pymotw'),
        ('write about sqlite3', 'active', '2010-10-17',
         'pymotw'),
    ]
    conn.executemany("""
    insert into task (details, status, deadline, project)
    values (?, ?, ?, ?)
    """, data)

    print('Dumping:')
    for text in conn.iterdump():
        print(text)
```

Метод `iterdump()` также можно использовать по отношению к базам данных, сохраненным в файлах, но он оказывается наиболее полезным в случае баз данных, не требующих сохранения. Вывод, создаваемый в данном примере, отредактирован таким образом, чтобы он вписывался в страницу, но при этом оставался синтаксически корректным.

```
$ python3 sqlite3_iterdump.py

Creating schema
Inserting initial data
Dumping:
BEGIN TRANSACTION;
CREATE TABLE project (
    name        text primary key,
    description text,
    deadline    date
);
INSERT INTO "project" VALUES('pymotw','Python Module of the
Week','2010-11-01');
DELETE FROM "sqlite_sequence";
INSERT INTO "sqlite_sequence" VALUES('task',3);
CREATE TABLE task (
    id          integer primary key autoincrement not null,
    priority    integer default 1,
    details     text,
```

```
    status       text,
    deadline     date,
    completed_on date,
    project      text not null references project(name)
);
INSERT INTO "task" VALUES(1,1,'write about
select','done','2010-10-03',NULL,'pymotw');
INSERT INTO "task" VALUES(2,1,'write about
random','waiting','2010-10-10',NULL,'pymotw');
INSERT INTO "task" VALUES(3,1,'write about
sqlite3','active','2010-10-17',NULL,'pymotw');
COMMIT;
```

7.4.13. Использование функций Python в SQL

Синтаксис SQL поддерживает вызов функций в ходе выполнения запросов, будь то в списке столбцов или в предложении `where` инструкции `select`. Это позволяет организовать обработку данных, прежде чем они будут возвращаться в ответ на запрос. Эту возможность можно использовать для преобразования данных между различными форматами, выполнения вычислений, которые были бы слишком громоздкими, будучи выражены исключительно на языке SQL, а также для повторного использования кода приложения.

Листинг 7.36. `sqlite3_create_function.py`

```
import codecs
import sqlite3

db_filename = 'todo.db'


def encrypt(s):
    print('Encrypting {!r}'.format(s))
    return codecs.encode(s, 'rot-13')


def decrypt(s):
    print('Decrypting {!r}'.format(s))
    return codecs.encode(s, 'rot-13')


with sqlite3.connect(db_filename) as conn:

    conn.create_function('encrypt', 1, encrypt)
    conn.create_function('decrypt', 1, decrypt)
    cursor = conn.cursor()

    # "Сырые" значения
    print('Original values:')
    query = "select id, details from task"
    cursor.execute(query)
    for row in cursor.fetchall():
```

```
        print(row)

    print('\nEncrypting...')
    query = "update task set details = encrypt(details)"
    cursor.execute(query)

    print('\nRaw encrypted values:')
    query = "select id, details from task"
    cursor.execute(query)
    for row in cursor.fetchall():
        print(row)

    print('\nDecrypting in query...')
    query = "select id, decrypt(details) from task"
    cursor.execute(query)
    for row in cursor.fetchall():
        print(row)

    print('\nDecrypting...')
    query = "update task set details = decrypt(details)"
    cursor.execute(query)
```

Функции предоставляются посредством метода `create_function()` объекта `Connection`. Параметрами являются имя функции (в том виде, в каком оно должно использоваться в инструкции SQL), количество аргументов, передаваемых функции, и предоставляемая функция Python.

```
$ python3 sqlite3_create_function.py

Original values:
(1, 'write about select')
(2, 'write about random')
(3, 'write about sqlite3')
(4, 'finish reviewing markup')
(5, 'revise chapter intros')
(6, 'subtitle')

Encrypting...
Encrypting 'write about select'
Encrypting 'write about random'
Encrypting 'write about sqlite3'
Encrypting 'finish reviewing markup'
Encrypting 'revise chapter intros'
Encrypting 'subtitle'

Raw encrypted values:
(1, 'jevgr nobhg fryrpg')
(2, 'jevgr nobhg enaqbz')
(3, 'jevgr nobhg fdyvgr3')
(4, 'svavfu erivrjvat znexhc')
(5, 'erivfr puncgre vagebf')
(6, 'fhogvgyr')
```

```
Decrypting in query...
Decrypting 'jevgr nobhg fryrpg'
Decrypting 'jevgr nobhg enaqbz'
Decrypting 'jevgr nobhg fdyvgr3'
Decrypting 'svavfu erivrjvat znexhc'
Decrypting 'erivfr puncgre vagebf'
Decrypting 'fhogvgyr'
(1, 'write about select')
(2, 'write about random')
(3, 'write about sqlite3')
(4, 'finish reviewing markup')
(5, 'revise chapter intros')
(6, 'subtitle')

Decrypting...
Decrypting 'jevgr nobhg fryrpg'
Decrypting 'jevgr nobhg enaqbz'
Decrypting 'jevgr nobhg fdyvgr3'
Decrypting 'svavfu erivrjvat znexhc'
Decrypting 'erivfr puncgre vagebf'
Decrypting 'fhogvgyr'
```

7.4.14. Использование регулярных выражений в запросах

SQLite поддерживает несколько специальных пользовательских функций, связанных с синтаксисом SQL. Например, для проверки соответствия строкового значения столбца регулярному выражению можно использовать функцию REGEXP.

```
SELECT * FROM table
WHERE column REGEXP '.*pattern.*'
```

В приведенном ниже примере создаваемая функция связывается с функцией regexp() для тестирования значений с использованием модуля Python re (см. раздел 1.3).

Листинг 7.37. sqlite3_regex.py

```
import re
import sqlite3

db_filename = 'todo.db'


def regexp(pattern, input):
    return bool(re.match(pattern, input))


with sqlite3.connect(db_filename) as conn:
    conn.row_factory = sqlite3.Row
    conn.create_function('regexp', 2, regexp)
    cursor = conn.cursor()
```

```
    pattern = '.*[wW]rite [aA]bout.*'

    cursor.execute(
        """
        select id, priority, details, status, deadline from task
        where details regexp :pattern
        order by deadline, priority
        """,
        {'pattern': pattern},
    )

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

Выводятся все задачи, в которых столбец `details` соответствует шаблону.

```
$ python3 sqlite3_regex.py

 1 [9] write about select          [done    ] (2016-04-25)
 2 [9] write about random          [done    ] (2016-08-22)
 3 [9] write about sqlite3         [active  ] (2017-07-31)
```

7.4.15. Пользовательское агрегирование

Функции агрегирования объединяют разрозненные порции отдельных данных и формирует на их основе сводные данные различного рода. Примерами функций агрегирования могут служить `avg()` (среднее), `min()`, `max()` и `count()`.

API для агрегаторов, используемых модулем `sqlite3`, определяется в терминах класса, имеющего два метода. Метод `step()` вызывается для каждого значения данных в процессе обработки запроса. Метод `finalize()` вызывается один раз в конце запроса и должен возвращать агрегированное значение. В следующем примере реализован агрегатор для арифметической *моды*. Он возвращает значение, которое встречается среди входных данных чаще других.

Листинг 7.38. `sqlite3_create_aggregate.py`

```
import sqlite3
import collections

db_filename = 'todo.db'


class Mode:

    def __init__(self):
        self.counter = collections.Counter()

    def step(self, value):
        print('step({!r})'.format(value))
        self.counter[value] += 1
```

```
    def finalize(self):
        result, count = self.counter.most_common(1)[0]
        print('finalize() -> {!r} ({} times)'.format(
            result, count))
        return result


with sqlite3.connect(db_filename) as conn:
    conn.create_aggregate('mode', 1, Mode)

    cursor = conn.cursor()
    cursor.execute("""
    select mode(deadline) from task where project = 'pymotw'
    """)
    row = cursor.fetchone()
    print('mode(deadline) is:', row[0])
```

Класс агрегатора регистрируется с помощью метода `create_aggregate()` объекта `Connection`. Параметрами являются имя функции (в том виде, в каком оно используется в SQL-инструкциях), количество аргументов, передаваемых методу `step()`, и используемый класс.

```
$ python3 sqlite3_create_aggregate.py

step('2016-04-25')
step('2016-08-22')
step('2017-07-31')
step('2016-11-30')
step('2016-08-20')
step('2016-11-01')
finalize() -> '2016-11-01' (1 times)
mode(deadline) is: 2016-11-01
```

7.4.16. Многопоточность и совместное использование соединений

По историческим причинам, уходящим корнями в старые версии SQLite, объекты `Connection` не могут совместно использоваться несколькими потоками. Каждый поток должен создавать собственное подключение к базе данных.

Листинг 7.39. `sqlite3_threading.py`

```
import sqlite3
import sys
import threading
import time

db_filename = 'todo.db'
isolation_level = None  # режим автоматической фиксации изменений


def reader(conn):
    print('Starting thread')
```

```
    try:
        cursor = conn.cursor()
        cursor.execute('select * from task')
        cursor.fetchall()
        print('results fetched')
    except Exception as err:
        print('ERROR:', err)


if __name__ == '__main__':
    with sqlite3.connect(db_filename,
                         isolation_level=isolation_level,
                         ) as conn:
        t = threading.Thread(name='Reader 1',
                             target=reader,
                             args=(conn,),
                             )
        t.start()
        t.join()
```

Попытки совместного использования соединения одновременно несколькими потоками приводят к возбуждению исключения.

```
$ python3 sqlite3_threading.py

Starting thread
ERROR: SQLite objects created in a thread can only be used in that
same thread.The object was created in thread id 140735234088960
and this is thread id 123145307557888
```

7.4.17. Ограничение доступа к данным

Несмотря на то что в SQLite нет средств управления правами доступа пользователей, предусмотренных в других, более мощных СУБД, в ней имеется механизм, позволяющий ограничивать доступ к столбцам. Каждое соединение может установить *функцию авторизации*, которая будет управлять возможностью доступа к столбцам на этапе выполнения на основании указанных критериев. Функция авторизации вызывается в процессе анализа SQL-инструкций. Ей передаются пять параметров. Первый из них — код действия (`action`) — указывает на тип выполняемой операции (например, чтение, запись или удаление), а остальные аргументы зависят от него. Для операции `SQLITE_READ` аргументами являются имя таблицы, имя столбца, расположение в SQL-коде, из которого осуществляется доступ (например, основной запрос, триггер), и значение `None`.

Листинг 7.40. `sqlite3_set_authorizer.py`

```
import sqlite3

db_filename = 'todo.db'


def authorizer_func(action, table, column, sql_location, ignore):
    print('\nauthorizer_func({}, {}, {}, {}, {})'.format(
```

```
        action, table, column, sql_location, ignore))

    response = sqlite3.SQLITE_OK  # по умолчанию разрешить

    if action == sqlite3.SQLITE_SELECT:
        print('requesting permission to run a select statement')
        response = sqlite3.SQLITE_OK

    elif action == sqlite3.SQLITE_READ:
        print('requesting access to column {}.{} from {}'.format(
            table, column, sql_location))
        if column == 'details':
            print('  ignoring details column')
            response = sqlite3.SQLITE_IGNORE
        elif column == 'priority':
            print('  preventing access to priority column')
            response = sqlite3.SQLITE_DENY

    return response


with sqlite3.connect(db_filename) as conn:
    conn.row_factory = sqlite3.Row
    conn.set_authorizer(authorizer_func)

    print('Using SQLITE_IGNORE to mask a column value:')
    cursor = conn.cursor()
    cursor.execute("""
    select id, details from task where project = 'pymotw'
    """)
    for row in cursor.fetchall():
        print(row['id'], row['details'])

    print('\nUsing SQLITE_DENY to deny access to a column:')
    cursor.execute("""
    select id, priority from task where project = 'pymotw'
    """)
    for row in cursor.fetchall():
        print(row['id'], row['details'])
```

В этом примере константа `SQLITE_IGNORE` задает замену строк из столбца `task.details` значениями `NULL` в результатах запроса. Кроме того, функция авторизации запрещает любой доступ к столбцу `task.priority`, возвращая для него константу `SQLITE_DENY`, что, в свою очередь, приводит к тому, что SQLite возбуждает исключение.

```
$ python3 sqlite3_set_authorizer.py

Using SQLITE_IGNORE to mask a column value:

authorizer_func(21, None, None, None, None)
requesting permission to run a select statement
```

```
authorizer_func(20, task, id, main, None)
requesting access to column task.id from main

authorizer_func(20, task, details, main, None)
requesting access to column task.details from main
  ignoring details column

authorizer_func(20, task, project, main, None)
requesting access to column task.project from main
1 None
2 None
3 None
4 None
5 None
6 None

Using SQLITE_DENY to deny access to a column:

authorizer_func(21, None, None, None, None)
requesting permission to run a select statement

authorizer_func(20, task, id, main, None)
requesting access to column task.id from main

authorizer_func(20, task, priority, main, None)
requesting access to column task.priority from main
  preventing access to priority column
Traceback (most recent call last):
  File "sqlite3_set_authorizer.py", line 53, in <module>
    """)
sqlite3.DatabaseError: access to task.priority is prohibited
```

Возможные коды действий предоставляются модулем `sqlite3` в виде констант, имена которых начинаются с префикса `SQLITE_`. Их использование обеспечивает управление правами доступа к столбцам таблиц для всех типов SQL-инструкций.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `sqlite3`[10].
- **PEP 249**[11]. *Python Database API Specification v2.0*. Стандартный интерфейс для модулей, обеспечивающих доступ к реляционным базам данных.
- SQLite[12]. Официальный сайт библиотеки SQLite.
- `shelve` (раздел 7.2). Хранилище с доступом по ключу, предназначенное для хранения произвольных объектов Python.
- SQLAlchemy[13]. Популярная библиотека, содержащая средства объектно-реляционного отображения и поддерживающая SQLite наряду с другими реляционными базами данных.

[10] https://docs.python.org/3.5/library/sqlite3.html

[11] www.python.org/dev/peps/pep-0249

[12] www.sqlite.org

[13] www.sqlalchemy.org

7.5. xml.etree.ElementTree: API для манипулирования XML-элементами

Библиотека *ElementTree* включает инструментальные средства, предназначенные для синтаксического анализа (парсинга) XML-документов с использованием двух подходов, основанных соответственно на обработке событий в процессе чтения документа и на объектной модели документа, а также для поиска XML-элементов с помощью выражений XPath и создания или изменения существующих документов.

7.5.1. Синтаксический анализ XML-документов

XML-документы, прошедшие синтаксический анализ, представляются в памяти объектами `ElementTree` и `Element`, объединяемыми в единую иерархическую структуру на основе способа вложения узлов в исходном документе.

Парсинг всего документа с помощью метода `parse()` возвращает экземпляр `ElementTree`. Иерархическому дереву доступны все данные документа, что обеспечивает возможность поиска нужных узлов и манипулирования ими на месте. Несмотря на гибкость этого подхода и предлагаемые им удобства, он требует загрузки всего документа, что приводит к значительному увеличению потребления памяти при работе с большими файлами по сравнению с подходом на основе обработки событий, генерируемых в процессе анализа документа.

В то же время при работе с простыми документами малого размера объем потребляемой оперативной памяти оказывается небольшим, как в приведенном ниже примере документа, представляющего список подкастов в формате OPML (от англ. *Outline Processor Markup Language*).

Листинг 7.41. `podcasts.opml`

```
<?xml version="1.0" encoding="UTF-8"?>
<opml version="1.0">
<head>
    <title>My Podcasts</title>
    <dateCreated>Sat, 06 Aug 2016 15:53:26 GMT</dateCreated>
    <dateModified>Sat, 06 Aug 2016 15:53:26 GMT</dateModified>
</head>
<body>
  <outline text="Non-tech">
    <outline
        text="99% Invisible" type="rss"
        xmlUrl="http://feeds.99percentinvisible.org/
Ä99percentinvisible"
        htmlUrl="http://99percentinvisible.org" />
  </outline>
  <outline text="Python">
    <outline
        text="Talk Python to Me" type="rss"
        xmlUrl="https://talkpython.fm/episodes/rss"
        htmlUrl="https://talkpython.fm" />
    <outline
        text="Podcast.__init__" type="rss"
```

```
        xmlUrl="http://podcastinit.podbean.com/feed/"
        htmlUrl="http://podcastinit.com" />
  </outline>
</body>
</opml>
```

Чтобы выполнить анализ, передайте дескриптор открытого файла методу `parse()`.

Листинг 7.42. `ElementTree_parse_opml.py`

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

print(tree)
```

Метод `parse()` читает данные, анализирует XML-документ и возвращает объект `ElementTree`.

```
$ python3 ElementTree_parse_opml.py

<xml.etree.ElementTree.ElementTree object at 0x1013e5630>
```

7.5.2. Обход дерева узлов

Для поочередного обхода всех узлов иерархического дерева элементов документа используем метод `iter()`, который возвращает генератор, обеспечивающий итерирование по экземпляру `ElementTree`.

Листинг 7.43. `ElementTree_dump_opml.py`

```
from xml.etree import ElementTree
import pprint

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.iter():
    print(node.tag)
```

В этом примере все дерево документа выводится по одному дескриптору за раз.

```
$ python3 ElementTree_dump_opml.py

opml
head
title
dateCreated
dateModified
body
outline
outline
```

```
outline
outline
outline
```

Чтобы вывести на печать лишь группы имен и URL-адреса подкастов, пропустим все данные в разделе заголовка и выведем лишь атрибуты `text` и `xmlUrl`, выполнив поиск соответствующих значений в словаре `attrib`.

Листинг 7.44. `ElementTree_show_feed_urls.py`

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.iter('outline'):
    name = node.attrib.get('text')
    url = node.attrib.get('xmlUrl')
    if name and url:
        print('  %s' % name)
        print('    %s' % url)
    else:
        print(name)
```

Аргумент `'outline'` метода `iter()` задает обработку лишь тех узлов, которые соответствуют дескрипторам `'outline'`.

```
$ python3 ElementTree_show_feed_urls.py

Non-tech
  99% Invisible
    http://feeds.99percentinvisible.org/99percentinvisible
Python
  Talk Python to Me
    https://talkpython.fm/episodes/rss
  Podcast.__init__
    http://podcastinit.podbean.com/feed/
```

7.5.3. Поиск узлов в документе

Обход всего дерева для поиска нужных узлов чреват ошибками. В предыдущем примере мы должны были проверять для каждого узла `outline`, соответствует ли он группе (узлы, содержащие только атрибут `text`) или подкасту (узлы, содержащие атрибуты `text` и `xmlUrl`). Для получения простого списка URL-адресов потоков без имен и групп логику можно упростить за счет привлечения метода `findall()` для поиска узлов с использованием более описательных характеристик.

На первом этапе преобразования первоначальной версии сценария используем для поиска всех узлов `outline` аргумент в виде пути XPath.

Листинг 7.45. ElementTree_find_feeds_by_tag.py

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.findall('.//outline'):
    url = node.attrib.get('xmlUrl')
    if url:
        print(url)
```

Логика этой версии почти не отличается от той, которая использовалась в версии с методом `iter()`. В ней по-прежнему приходится проверять наличие URL-адреса в элементе, за исключением того, что в случае отсутствия URL-адреса групповое имя не выводится.

```
$ python3 ElementTree_find_feeds_by_tag.py

http://feeds.99percentinvisible.org/99percentinvisible
https://talkpython.fm/episodes/rss
http://podcastinit.podbean.com/feed/
```

Можно воспользоваться тем фактом, что узлы `outline` имеют только два уровня вложенности. Изменение пути поиска на `.//outline/outline` означает, что цикл будет обрабатывать только узлы `outline` второго уровня.

Листинг 7.46. ElementTree_find_feeds_by_structure.py

```
from xml.etree import ElementTree

with open('podcasts.opml', 'rt') as f:
    tree = ElementTree.parse(f)

for node in tree.findall('.//outline/outline'):
    url = node.attrib.get('xmlUrl')
    print(url)
```

Предполагается, что в исходном документе все узлы `outline` второго уровня содержат атрибут `xmlURLattribute`, ссылающийся на канал подкаста, поэтому проверку его наличия можно опустить.

```
$ python3 ElementTree_find_feeds_by_structure.py

http://feeds.99percentinvisible.org/99percentinvisible
https://talkpython.fm/episodes/rss
http://podcastinit.podbean.com/feed/
```

Вместе с тем эта версия сценария привязана к существующей структуре и потому не будет работать, если структура вложения узлов `outline` по каким-либо причинам впоследствии изменится.

7.5.4. Атрибуты узлов

Элементами, возвращаемыми функциями `findall()` и `iter()`, являются объекты `Element`, каждый из которых представляет узел в иерархическом представлении XML-документа. Каждый объект `Element` имеет атрибуты для доступа к данным, извлеченным из XML. Это поведение можно проиллюстрировать на чуть более сложном примере с помощью файла *data.xml*.

Листинг 7.47. `data.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<top>
  <child>Regular text.</child>
  <child_with_tail>Regular text.</child_with_tail>"Tail" text.
  <with_attributes name="value" foo="bar" />
  <entity_expansion attribute="This &#38; That">
    That &#38; This
  </entity_expansion>
</top>
```

XML-атрибуты узла доступны через свойство `attrib`, которое ведет себя подобно словарю.

Листинг 7.48. `ElementTree_node_attributes.py`

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

node = tree.find('./with_attributes')
print(node.tag)
for name, value in sorted(node.attrib.items()):
    print('  %-4s = "%s"' % (name, value))
```

Узел в строке 5 входного файла имеет два атрибута: `name` и `foo`.

```
$ python3 ElementTree_node_attributes.py

with_attributes
  foo = "bar"
  name = "value"
```

Текстовое содержимое узлов доступно вместе с прикрепленным к ним (`tail`) текстом, следующим после закрывающего дескриптора.

Листинг 7.49. `ElementTree_node_text.py`

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

for path in ['./child', './child_with_tail']:
    node = tree.find(path)
```

```
    print(node.tag)
    print('  child node text:', node.text)
    print('  and tail text  :', node.tail)
```

Дочерний узел в строке 3 содержит встроенный текст, а узел в строке 4 содержит текст с прикрепленной текстовой частью (включающий пробел).

```
$ python3 ElementTree_node_text.py

child
  child node text: Regular text.
  and tail text  :

child_with_tail
  child node text: Regular text.
  and tail text  : "Tail" text.
```

Ссылки на XML-сущности, встроенные в документ, преобразуются в соответствующие символы до возврата значений.

Листинг 7.50. `ElementTree_entity_references.py`

```
from xml.etree import ElementTree

with open('data.xml', 'rt') as f:
    tree = ElementTree.parse(f)

node = tree.find('entity_expansion')
print(node.tag)
print('  in attribute:', node.attrib['attribute'])
print('  in text     :', node.text.strip())
```

Автоматический характер преобразования означает, что детали реализации представлений, определенных в XML-документе, могут игнорироваться.

```
entity_expansion
  in attribute: This & That
  in text     : That & This
```

7.5.5. Отслеживание событий в процессе анализа документа

Второй API, предназначенный для обработки XML-документов, основан на обработке событий. Когда встречается открывающий дескриптор, анализатор генерирует событие `start`, а когда закрывающий — событие `end`. На этапе синтаксического анализа данные могут извлекаться путем итерирования по потоку событий, что удобно, если после этого не требуется манипулировать сразу всем документом, а проанализированный документ не должен храниться в памяти целиком.

Возможные типы событий перечислены ниже.

`start`
: Встретился новый дескриптор. Обрабатывается закрывающая угловая скобка дескриптора, но не его содержимое.

`end`
: Обрабатывается закрывающая угловая скобка закрывающего дескриптора. К этому моменту все дочерние узлы уже были обработаны.

`start-ns`
: Начало объявления пространства имен.

`end-ns`
: Конец объявления пространства имен.

Метод `iterparse()` возвращает итерируемый объект, который вырабатывает кортежи, содержащие имя события и узел, запустивший это событие.

Листинг 7.51. `ElementTree_show_all_events.py`

```
from xml.etree.ElementTree import iterparse

depth = 0
prefix_width = 8
prefix_dots = '.' * prefix_width
line_template = ''.join([
    '{prefix:<0.{prefix_len}}',
    '{event:<8}',
    '{suffix:<{suffix_len}} ',
    '{node.tag:<12} ',
    '{node_id}',
])

EVENT_NAMES = ['start', 'end', 'start-ns', 'end-ns']

for (event, node) in iterparse('podcasts.opml', EVENT_NAMES):
    if event == 'end':
        depth -= 1

    prefix_len = depth * 2

    print(line_template.format(
        prefix=prefix_dots,
        prefix_len=prefix_len,
        suffix='',
        suffix_len=(prefix_width - prefix_len),
        node=node,
        node_id=id(node),
        event=event,
    ))

    if event == 'start':
        depth += 1
```

По умолчанию генерируются лишь события `end`. Чтобы увидеть другие события, передайте список имен желаемых событий методу `iterparse()`, как показано в этом примере.

```
$ python3 ElementTree_show_all_events.py
start           opml         4312612200
..start         head         4316174520
....start       title        4316254440
....end         title        4316254440
....start       dateCreated  4316254520
....end         dateCreated  4316254520
....start       dateModified 4316254680
....end         dateModified 4316254680
..end           head         4316174520
..start         body         4316254840
....start       outline      4316254920
......start     outline      4316255080
......end       outline      4316255080
....end         outline      4316254920
....start       outline      4316255160
......start     outline      4316255240
......end       outline      4316255240
......start     outline      4316255320
......end       outline      4316255320
....end         outline      4316255160
..end           body         4316254840
end             opml         4312612200
```

Основанный на обработке событий подход более естествен для таких операций, как преобразование XML-данных в другой формат. Эту технику можно использовать для преобразования списка подкастов из предыдущих примеров в CSV-файл, которые затем могут быть загружены в электронную таблицу или приложение базы данных.

Листинг 7.52. `ElementTree_write_podcast_csv.py`

```
import csv
from xml.etree.ElementTree import iterparse
import sys

writer = csv.writer(sys.stdout, quoting=csv.QUOTE_NONNUMERIC)

group_name = ''

parsing = iterparse('podcasts.opml', events=['start'])

for (event, node) in parsing:
    if node.tag != 'outline':
        # Игнорировать все, что не входит в дескриптор outline
        continue
    if not node.attrib.get('xmlUrl'):
        # Запомнить текущую группу
        group_name = node.attrib['text']
    else:
        # Вывести запись, соответствующую подкасту
        writer.writerow(
            (group_name, node.attrib['text'],
```

```
            node.attrib['xmlUrl'],
            node.attrib.get('htmlUrl', ''))
        )
```

В этой программе отсутствует необходимость в сохранении полной объектной модели документа в памяти машины, и поэтому поочередная обработка узлов по мере того, как они встречаются во входном документе, оказывается более эффективной.

```
$ python3 ElementTree_write_podcast_csv.py

"Non-tech","99% Invisible","http://feeds.99percentinvisible.org/\
99percentinvisible","http://99percentinvisible.org"
"Python","Talk Python to Me","https://talkpython.fm/episodes/rss\
","https://talkpython.fm"
"Python","Podcast.__init__","http://podcastinit.podbean.com/feed\
/","http://podcastinit.com"
```

Примечание

Вывод сценария `ElementTree_write_podcast_csv.py` переформатирован в соответствии с шириной печатной страницы. Символы \ указывают на искусственные разрывы строк.

7.5.6. Создание построителя пользовательского дерева

Потенциально более широкие возможности обработки событий парсинга открывает замена стандартного построителя иерархического дерева узлов документа его пользовательской версией. Анализатор `XMLParser` использует объект `TreeBuilder` для обработки XML-документа и вызова методов для целевого класса с целью сохранения результатов. Обычным выводом является экземпляр `ElementTree`, создаваемый заданным по умолчанию классом `TreeBuilder`. Замена класса `TreeBuilder` другим классом позволяет получать события до инстанциализации узлов `Element`, что приводит к экономии части ресурсов.

Конвертер XML-CSV из предыдущего примера нетрудно переделать в построитель иерархического дерева узлов.

Листинг 7.53. `ElementTree_podcast_csv_treebuilder.py`

```
import csv
import io
from xml.etree.ElementTree import XMLParser
import sys


class PodcastListToCSV(object):

    def __init__(self, outputFile):
        self.writer = csv.writer(
            outputFile,
            quoting=csv.QUOTE_NONNUMERIC,
        )
        self.group_name = ''
```

```
    def start(self, tag, attrib):
        if tag != 'outline':
            # Игнорировать все, что не входит в дескриптор outline
            return
        if not attrib.get('xmlUrl'):
            # Запомнить текущую группу
            self.group_name = attrib['text']
        else:
            # Вывести запись, соответствующую подкасту
            self.writer.writerow(
                (self.group_name,
                 attrib['text'],
                 attrib['xmlUrl'],
                 attrib.get('htmlUrl', ''))
            )

    def end(self, tag):
        "Ignore closing tags"

    def data(self, data):
        "Ignore data inside nodes"

    def close(self):
        "Nothing special to do here"


target = PodcastListToCSV(sys.stdout)
parser = XMLParser(target=target)
with open('podcasts.opml', 'rt') as f:
    for line in f:
        parser.feed(line)
parser.close()
```

Класс `PodcastListToCSV` реализует протокол `TreeBuilder`. Каждый раз, когда встречается новый XML-дескриптор, вызывается метод `start()`, которому передаются имя дескриптора и атрибуты. Когда встречается соответствующий закрывающий дескриптор, вызывается метод `end()`, которому передается имя дескриптора. В промежутках между этими двумя событиями для каждого встретившегося узла с содержимым вызывается метод `data()` (предполагается, что объект построения дерева отслеживает текущий узел). После обработки всех входных данных вызывается метод `close()`. Этот метод может вернуть значение, которое будет возвращено пользователю объекта `TreeBuilder`.

```
$ python3 ElementTree_podcast_csv_treebuilder.py

"Non-tech","99% Invisible","http://feeds.99percentinvisible.org/\
99percentinvisible","http://99percentinvisible.org"
"Python","Talk Python to Me","https://talkpython.fm/episodes/rss\
","https://talkpython.fm"
"Python","Podcast.__init__","http://podcastinit.podbean.com/feed\
/","http://podcastinit.com"
```

Примечание

Вывод сценария `ElementTree_podcast_csv_treebuidler.py` переформатирован в соответствии с шириной печатной страницы. Символы \ указывают на искусственные разрывы строк.

7.5.7. Синтаксический анализ строк

При работе с небольшими порциями текста в формате XML, особенно со строковыми литералами, которые могут внедряться в исходный код программы, используйте функцию `XML()`, передавая ей строку с XML-текстом, подлежащим анализу, в качестве единственного аргумента.

Листинг 7.54. `ElementTree_XML.py`

```
from xml.etree.ElementTree import XML


def show_node(node):
    print(node.tag)
    if node.text is not None and node.text.strip():
        print('  text: "%s"' % node.text)
    if node.tail is not None and node.tail.strip():
        print('  tail: "%s"' % node.tail)
    for name, value in sorted(node.attrib.items()):
        print('  %-4s = "%s"' % (name, value))
    for child in node:
        show_node(child)


parsed = XML('''
<root>
  <group>
    <child id="a">This is child "a".</child>
    <child id="b">This is child "b".</child>
  </group>
  <group>
    <child id="c">This is child "c".</child>
  </group>
</root>
''')

print('parsed =', parsed)

for elem in parsed:
    show_node(elem)
```

В отличие от метода `parse()`, в данном случае возвращаемым значением является экземпляр `Element`, а не `ElementTree`.

Класс `Element` непосредственно поддерживает протокол итератора, что избавляет от необходимости вызывать метод `getiterator()`.

```
$ python3 ElementTree_XML.py

parsed = <Element 'root' at 0x10079eef8>
group
child
  text: "This is child "a"."
  id   = "a"
child
  text: "This is child "b"."
  id   = "b"
group
child
  text: "This is child "c"."
  id   = "c"
```

В случае структурированных XML-документов, в которых для идентификации уникальных узлов используются атрибуты `id`, удобный способ доступа к результатам анализа документа обеспечивает функция `XMLID()`.

Листинг 7.55. `ElementTree_XMLID.py`

```
from xml.etree.ElementTree import XMLID

tree, id_map = XMLID('''
<root>
  <group>
    <child id="a">This is child "a".</child>
    <child id="b">This is child "b".</child>
  </group>
  <group>
    <child id="c">This is child "c".</child>
  </group>
</root>
''')

for key, value in sorted(id_map.items()):
    print('%s = %s' % (key, value))
```

Функция `XMLID()` возвращает дерево в виде объекта `Element` вместе со словарем, сопоставляющим строки атрибутов `id` с индивидуальными узлами.

```
$ python3 ElementTree_XMLID.py

a = <Element 'child' at 0x10133aea8>
b = <Element 'child' at 0x10133aef8>
c = <Element 'child' at 0x10133af98>
```

7.5.8. Создание документов с помощью узлов `Element`

В дополнение к возможностям синтаксического анализа модуль `xml.etree.ElementTree` поддерживает создание корректно сформированных XML-документов из объектов `Element`, конструируемых в приложении. Классу `Element`, ис-

пользуемому при анализе документа, также известно, как генерировать сериализованную форму своего содержимого, которое затем может быть записано в файл или другой поток данных.

Для создания иерархии узлов `Element` используются три вспомогательные функции. Функция `Element()` создает стандартный узел, функция `SubElement()` присоединяет новый узел к родительскому узлу, а функция `Comment()` создает узел, который сериализует данные, используя синтаксис комментариев XML.

Листинг 7.56. `ElementTree_create.py`

```
from xml.etree.ElementTree import (
    Element, SubElement, Comment, tostring,
)

top = Element('top')

comment = Comment('Generated for PyMOTW')
top.append(comment)

child = SubElement(top, 'child')
child.text = 'This child contains text.'

child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has text.'
child_with_tail.tail = 'And "tail" text.'

child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'

print(tostring(top))
```

Вывод содержит лишь XML-узлы дерева — он не включает объявление версии и кодировки.

```
$ python3 ElementTree_create.py

b'<top><!--Generated for PyMOTW--><child>This child contains text.
</child><child_with_tail>This child has text.</child_with_tail>And
"tail" text.<child_with_entity_ref>This &amp; that</child_with_ent
ity_ref></top>'
```

Символ `&`, содержащийся в тексте узла `child_with_entity_ref`, автоматически преобразуется в ссылку на сущность `&`.

7.5.9. Красивая печать XML

Класс `ElementTree` не делает никаких попыток форматировать вывод функции `tostring()` для улучшения его удобочитаемости, поскольку добавление дополнительных пробелов изменяет содержимое документа. Чтобы сделать вывод более удобным для чтения, в оставшихся примерах XML-документ предварительно преобразуется с помощью модуля `xml.dom.minidom` перед применением к нему метода `toprettyxml()`.

Листинг 7.57. ElementTree_pretty.py

```
from xml.etree import ElementTree
from xml.dom import minidom


def prettify(elem):
    """Вернуть красиво оформленную XML-строку для объекта Element.
    """
    rough_string = ElementTree.tostring(elem, 'utf-8')
    reparsed = minidom.parseString(rough_string)
    return reparsed.toprettyxml(indent="  ")
```

Обновленный пример представлен в следующем листинге.

Листинг 7.58. ElementTree_create_pretty.py

```
from xml.etree.ElementTree import Element, SubElement, Comment
from ElementTree_pretty import prettify

top = Element('top')

comment = Comment('Generated for PyMOTW')
top.append(comment)

child = SubElement(top, 'child')
child.text = 'This child contains text.'

child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has text.'
child_with_tail.tail = 'And "tail" text.'

child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'

print(prettify(top))
```

Читать вывод, создаваемый этим примером, значительно легче.

```
$ python3 ElementTree_create_pretty.py

<?xml version="1.0" ?>
<top>
  <!--Generated for PyMOTW-->
  <child>This child contains text.</child>
  <child_with_tail>This child has text.</child_with_tail>
  And &quot;tail&quot; text.
  <child_with_entity_ref>This &amp; that</child_with_entity_ref>
</top>
```

Кроме дополнительных пробелов, добавленных с целью форматирования, модуль `xml.dom.minidom` включает в вывод объявление XML.

7.5.10. Установка свойств объектов `Element`

В предыдущем примере создавались узлы с дескрипторами и текстовым содержимым, но их атрибуты не устанавливались. В разделе 7.5.1 многие примеры работали с OPML-файлом, содержащим список подкастов и их потоков. Для представления имен групп и свойств подкастов в узлах `outline` иерархического дерева использовались атрибуты. Класс `ElementTree` позволяет создать аналогичный XML-файл из входного CSV-файла, установив все атрибуты элементов в процессе построения дерева.

Листинг 7.59. `ElementTree_csv_to_xml.py`

```
import csv
from xml.etree.ElementTree import (
    Element, SubElement, Comment, tostring,
)
import datetime
from ElementTree_pretty import prettify

generated_on = str(datetime.datetime.now())

# Конфигурирование одного атрибута с помощью метода set()
root = Element('opml')
root.set('version', '1.0')

root.append(
    Comment('Generated by ElementTree_csv_to_xml.py for PyMOTW')
)

head = SubElement(root, 'head')
title = SubElement(head, 'title')
title.text = 'My Podcasts'
dc = SubElement(head, 'dateCreated')
dc.text = generated_on
dm = SubElement(head, 'dateModified')
dm.text = generated_on

body = SubElement(root, 'body')

with open('podcasts.csv', 'rt') as f:
    current_group = None
    reader = csv.reader(f)
    for row in reader:
        group_name, podcast_name, xml_url, html_url = row
        if (current_group is None or
                group_name != current_group.text):
            # Начать новую группу
            current_group = SubElement(
                body, 'outline',
                {'text': group_name},
            )
        # Добавить подкаст в группу,
        # одновременно установив значения
```

```
        # всех его атрибутов
        podcast = SubElement(
            current_group, 'outline',
            {'text': podcast_name,
             'xmlUrl': xml_url,
             'htmlUrl': html_url},
        )

print(prettify(root))
```

В этом примере для задания значений атрибутов новых узлов используются два приема. Корневой узел конфигурируется с помощью метода `set()` с изменением по одному атрибуту за раз. Атрибуты узлов подкастов задаются сразу за счет передачи словаря фабрике узлов.

```
$ python3 ElementTree_csv_to_xml.py

<?xml version="1.0" ?>
<opml version="1.0">
  <!--Generated by ElementTree_csv_to_xml.py for PyMOTW-->
  <head>
    <title>My Podcasts</title>
    <dateCreated>2017-09-05 00:38:57.109257</dateCreated>
    <dateModified>2017-09-05 00:38:57.109257</dateModified>
  </head>
  <body>
    <outline text="Non-tech">
      <outline htmlUrl="http://99percentinvisible.org" text="99%\
 Invisible" xmlUrl="http://feeds.99percentinvisible.org/99percen\
tinvisible"/>
    </outline>
    <outline text="Python">
      <outline htmlUrl="https://talkpython.fm" text="Talk Python\
to Me" xmlUrl="https://talkpython.fm/episodes/rss"/>
    </outline>
    <outline text="Python">
      <outline htmlUrl="http://podcastinit.com" text="Podcast.__\
init__" xmlUrl="http://podcastinit.podbean.com/feed/"/>
    </outline>
  </body>
</opml>
```

7.5.11. Создание деревьев из списков узлов

Метод `extend()` позволяет добавить в экземпляр `Element` сразу несколько узлов.

Листинг 7.60. `ElementTree_extend.py`

```
from xml.etree.ElementTree import Element, tostring
from ElementTree_pretty import prettify

top = Element('top')
```

```
children = [
    Element('child', num=str(i))
    for i in range(3)
]

top.extend(children)

print(prettify(top))
```

Предоставленные в виде списка узлы добавляются непосредственно в новый родительский узел.

```
$ python3 ElementTree_extend.py

<?xml version="1.0" ?>
<top>
  <child num="0"/>
  <child num="1"/>
  <child num="2"/>
</top>
```

Если предоставляется другой экземпляр `Element`, то в новый родительский узел добавляются дочерние узлы данного узла.

Листинг 7.61. `ElementTree_extend_node.py`

```
from xml.etree.ElementTree import (
    Element, SubElement, tostring, XML,
)
from ElementTree_pretty import prettify

top = Element('top')

parent = SubElement(top, 'parent')

children = XML(
    '<root><child num="0" /><child num="1" />'
    '<child num="2" /></root>'
)
parent.extend(children)

print(prettify(top))
```

В данном случае узел с дескриптором `root`, созданный на основе XML-строки, имеет три дочерних узла, которые добавляются в дочерний узел. Узел `root` не включается в результирующее дерево.

```
$ python3 ElementTree_extend_node.py

<?xml version="1.0" ?>
<top>
  <parent>
    <child num="0"/>
    <child num="1"/>
```

```
    <child num="2"/>
  </parent>
</top>
```

Важно понимать, что метод `extend()` не изменяет существующие отношения между родительскими и дочерними узлами. Если значения, переданные методу `extend()`, уже существуют в иерархическом дереве, то они остаются и будут воспроизведены в выходном результате.

Листинг 7.62. `ElementTree_extend_node_copy.py`

```
from xml.etree.ElementTree import (
    Element, SubElement, tostring, XML,
)
from ElementTree_pretty import prettify

top = Element('top')

parent_a = SubElement(top, 'parent', id='A')
parent_b = SubElement(top, 'parent', id='B')

# Создать дочерние узлы
children = XML(
    '<root><child num="0" /><child num="1" />'
    '<child num="2" /></root>'
)

# Установка атрибутов id со значениями id объектов узлов
# для более отчетливой демонстрации наличия дубликатов
for c in children:
    c.set('id', str(id(c)))

# Добавление узла в первый родительский узел
parent_a.extend(children)

print('A:')
print(prettify(top))
print()

# Копирование узлов во второй родительский узел
parent_b.extend(children)

print('B:')
print(prettify(top))
print()
```

Установка уникальных идентификаторов объектов Python в качестве значений атрибутов этих дочерних узлов позволяет убедительно продемонстрировать тот факт, что одни и те же объекты узлов появляются в результирующем дереве более одного раза.

```
$ python3 ElementTree_extend_node_copy.py

A:
```

```
<?xml version="1.0" ?>
<top>
  <parent id="A">
    <child id="4316789880" num="0"/>
    <child id="4316789960" num="1"/>
    <child id="4316790040" num="2"/>
  </parent>
  <parent id="B"/>
</top>


B:
<?xml version="1.0" ?>
<top>
  <parent id="A">
    <child id="4316789880" num="0"/>
    <child id="4316789960" num="1"/>
    <child id="4316790040" num="2"/>
  </parent>
  <parent id="B">
    <child id="4316789880" num="0"/>
    <child id="4316789960" num="1"/>
    <child id="4316790040" num="2"/>
  </parent>
</top>
```

7.5.12. Сериализация XML-разметки в поток

Реализация метода `tostring()` предусматривает запись данных в оперативную память в виде файлового объекта и возврат строки, представляющей все дерево элементов. Такой подход позволяет уменьшить потребление памяти при работе с большими объемами данных и повысить эффективность использования библиотек ввода-вывода за счет осуществления записи непосредственно с использованием дескриптора файла при помощи метода `write()` объекта `ElementTree`.

Листинг 7.63. `ElementTree_write.py`

```
import io
import sys
from xml.etree.ElementTree import (
    Element, SubElement, Comment, ElementTree,
)

top = Element('top')

comment = Comment('Generated for PyMOTW')
top.append(comment)

child = SubElement(top, 'child')
child.text = 'This child contains text.'

child_with_tail = SubElement(top, 'child_with_tail')
child_with_tail.text = 'This child has regular text.'
```

```
child_with_tail.tail = 'And "tail" text.'

child_with_entity_ref = SubElement(top, 'child_with_entity_ref')
child_with_entity_ref.text = 'This & that'

empty_child = SubElement(top, 'empty_child')

ElementTree(top).write(sys.stdout.buffer)
```

Для вывода на консоль в этом примере используется буферизованный поток `sys.stdout.buffer` вместо обычного потока `sys.stdout`, поскольку объект `ElementTree` вырабатывает не строки Unicode, а закодированные байты. Выходные результаты также можно было бы записать в файл, открытый в двоичном режиме, или сокет.

```
$ python3 ElementTree_write.py

<top><!--Generated for PyMOTW--><child>This child contains text.</
child><child_with_tail>This child has regular text.</child_with_ta
il>And "tail" text.<child_with_entity_ref>This &amp;amp; that</child_w
ith_entity_ref><empty_child /></top>
```

Последний узел в дереве не содержит ни текста, ни других узлов и поэтому записывается в виде пустого дескриптора, `<empty_child />`. Метод `write()` получает аргумент `method`, позволяющий управлять обработкой пустых узлов.

Листинг 7.64. `ElementTree_write_method.py`

```
import io
import sys
from xml.etree.ElementTree import (
    Element, SubElement, ElementTree,
)

top = Element('top')

child = SubElement(top, 'child')
child.text = 'Contains text.'

empty_child = SubElement(top, 'empty_child')

for method in ['xml', 'html', 'text']:
    print(method)
    sys.stdout.flush()
    ElementTree(top).write(sys.stdout.buffer, method=method)
    print('\n')
```

Поддерживаются три метода.

- **`xml`.** Используется по умолчанию, возвращает `<empty_child />`.
- **`html`.** Создает пару дескрипторов, как это требуется в HTML-документах (`<empty_child></empty_child>`).
- **`text`.** Выводит только текст узлов, полностью опуская пустые дескрипторы.

```
$ python3 ElementTree_write_method.py

xml
<top><child>Contains text.</child><empty_child /></top>

html
<top><child>Contains text.</child><empty_child></empty_child></t
op>

text
Contains text.
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `xml.etree.ElementTree`[14].
- `csv` (раздел 7.6). Чтение и запись файлов со значениями, разделенными запятыми.
- `defusedxml`[15]. Пакет, полезный при работе с XML-данными из ненадежных источников, который устраняет уязвимости, связанные с возможными атаками типа DoS (отказ в обслуживании).
- *Pretty print xml with python — indenting xml*[16] (Rene Dudfield). Советы, касающиеся организации красивой печати XML-документов в Python.
- *ElementTree Overview*[17] (Fredrick Lundh). Оригинальная документация и ссылки на версии разработки библиотеки ElementTree.
- *Process XML in Python with ElementTree*[18] (David Mertz). Статья на сайте IBM DeveloperWorks.
- *Outline Processor Markup Language (OPML)*[19] (Dave Winer). Спецификация и документация OPML.
- *XML Path Language (XPath)*[20]. Синтаксис языка запросов к элементам XML-документа.
- *XPath Support in ElementTree*[21] (Fredrick Lundh). Часть оригинальной документации библиотеки ElementTree.

7.6. csv: файлы с данными, разделенными запятыми

Модуль `csv` используется для работы с данными, экспортированными из электронных таблиц и баз данных в текстовые файлы, содержащие поля и записи в формате CSV, в котором данные разделяются запятыми.

[14] https://docs.python.org/3.5/library/xml.etree.elementtree.html
[15] https://pypi.python.org/pypi/defusedxml
[16] http://renesd.blogspot.com/2007/05/pretty-print-xml-with-python.html
[17] http://effbot.org/zone/element-index.htm
[18] www.ibm.com/developerworks/library/x-matters28/
[19] www.opml.org
[20] www.w3.org/TR/xpath/
[21] http://effbot.org/zone/element-xpath.htm

7.6.1. Чтение

Для чтения данных из CSV-файла используют функцию `reader()`. Создаваемый ею объект чтения можно использовать в качестве итератора для поочередной обработки строк файла.

Листинг 7.65. csv_reader.py

```
import csv
import sys

with open(sys.argv[1], 'rt') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Первым аргументом функции `reader()` является источник текстовых строк. В данном случае — это файл, но допускается использование любого итерируемого объекта (например, экземпляра `StringIO` или списка). Также можно передать необязательные аргументы, позволяющие управлять анализом входных данных.

```
# testdata.csv
"Title 1","Title 2","Title 3","Title 4"
1,"a",08/18/07,"å"
2,"bſ",08/19/07,""
3,"c",08/20/07,"ç"
```

При чтении файла каждая строка анализируемых входных данных преобразуется в список строк.

```
$ python3 csv_reader.py testdata.csv

['Title 1', 'Title 2', 'Title 3', 'Title 4']
['1', 'a', '08/18/07', 'å']
['2', 'b', '08/19/07', 'ſ']
['3', 'c', '08/20/07', 'ç']
```

Анализатор обрабатывает разрывы строк, внедренные в строки таблицы, поэтому входные строки — это не то же самое, что прочитанные строки.

```
# testlinebreak.csv
"Title 1","Title 2","Title 3"
1,"first line
second line",08/18/07
```

Поля с разрывами строк во входном файле сохраняют внутренние разрывы строк, когда возвращаются анализатором.

```
$ python3 csv_reader.py testlinebreak.csv

['Title 1', 'Title 2', 'Title 3']
['1', 'first line\nsecond line', '08/18/07']
```

7.6.2. Запись

Запись CSV-файлов ничуть не труднее их чтения. Сначала следует создать объект записи с помощью функции `writer()`, а затем итерировать по строкам, используя метод `writerow()` для их вывода.

Листинг 7.66. `csv_writer.py`

```
import csv
import sys

unicode_chars = 'a∫ç'

with open(sys.argv[1], 'wt') as f:
    writer = csv.writer(f)
    writer.writerow(('Title 1', 'Title 2', 'Title 3', 'Title 4'))
    for i in range(3):
        row = (
            i + 1,
            chr(ord('a') + i),
            '08/{:02d}/07'.format(i + 1),
            unicode_chars[i],
        )
        writer.writerow(row)

print(open(sys.argv[1], 'rt').read())
```

Вывод будет выглядеть не так, как экспортированные данные, использованные в примере с методом `reader`, поскольку в нем отсутствуют кавычки.

```
$ python3 csv_writer.py testout.csv

Title 1,Title 2,Title 3,Title 4
1,a,08/01/07,a
2,b∫,08/02/07,
3,c,08/03/07,ç
```

7.6.2.1. Заключение в кавычки

Используемое по умолчанию поведение кавычек отличается для метода `writer()`, что и объясняет их отсутствие в выводе предыдущего примера. Чтобы добавить кавычки, следует изменить их поведение с помощью аргумента `quoting`.

```
writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
```

Используемому в данном случае режиму `QUOTE_NONNUMERIC` соответствует заключение в кавычки всех столбцов, содержащиеся в которых значения не являются числами.

```
$ python3 csv_writer_quoted.py testout_quoted.csv

"Title 1","Title 2","Title 3","Title 4"
1,"a","08/01/07","a"
```

```
2,"bƒ","08/02/07",""
3,"c","08/03/07","ç"
```

Доступны четыре опции управления режимом использования кавычек, определенные в виде констант в модуле csv.

- QUOTE_ALL. В кавычки заключаются все значения, независимо от типа.
- QUOTE_MINIMAL. В кавычки заключаются только поля, содержащие специальные символы (символы, которые могли бы сбить с толку анализатор, такие как символ-разделитель, заданный параметрами диалекта или опциями). Это поведение задано по умолчанию.
- QUOTE_NONNUMERIC. В кавычки заключаются все нечисловые поля. При использовании с методом reader входные поля, не заключенные в кавычки, преобразуются в числа с плавающей точкой.
- QUOTE_NONE. Поля не заключаются в кавычки. При использовании с методом reader символы кавычек включаются в значения полей (обычно они интерпретируются как разделители и отбрасываются).

7.6.3. Диалекты

Ввиду отсутствия строгого стандарта для файлов CSV анализатор должен проявлять определенную гибкость. Для обеспечения этой гибкости предусмотрено множество параметров, позволяющих управлять способом чтения и записи данных. Вместо того чтобы передавать каждый из этих параметров по отдельности методам reader() и writer(), они группируются в объект *диалекта*.

Классы диалектов можно регистрировать по их именам, чтобы вызывающему коду модуля csv не нужно было заранее знать, какие параметры установлены. Полный список зарегистрированных диалектов можно получить с помощью функции list_dialects().

Листинг 7.67. csv_list_dialects.py

```
import csv

print(csv.list_dialects())
```

Стандартная библиотека включает три диалекта: excel, excel-tabs и unix. Диалект excel предназначен для работы с данными в формате, используемом по умолчанию приложением Microsoft Excel; он работает также с приложением LibreOffice[22]. В диалекте unix в кавычки заключаются все поля с двойными кавычками, а в качестве разделителя записей используется символ \n.

```
$ python3 csv_list_dialects.py

['excel', 'excel-tab', 'unix']
```

[22] www.libreoffice.org

7.6.3.1. Создание диалекта

Если во входном файле вместо запятых в качестве разделителей полей используются символы канала (|), то можно зарегистрировать соответствующий диалект.

```
# testdata.pipes
"Title 1"|"Title 2"|"Title 3"
1|"first line
second line"|08/18/07
```

Листинг 7.68. `csv_dialect.py`

```
import csv

csv.register_dialect('pipes', delimiter='|')

with open('testdata.pipes', 'r') as f:
    reader = csv.reader(f, dialect='pipes')
    for row in reader:
        print(row)
```

Используя диалект "pipes", этот файл можно читать так же, как и файл с разделителем-запятой.

```
$ python3 csv_dialect.py

['Title 1', 'Title 2', 'Title 3']
['1', 'first line\nsecond line', '08/18/07']
```

7.6.3.2. Параметры диалекта

Диалект определяет все лексемы, используемые при парсинге или записи файла данных. Параметры формата файла, которые можно определить, приведены в табл. 7.3.

Таблица 7.3. Параметры диалекта CSV

Атрибут	По умолчанию	Описание
`delimiter`	`,`	Разделитель полей (одиночный символ)
`doublequote`	`True`	Флаг, определяющий, будут ли дублироваться экземпляры `quotechar`
`escapechar`	`None`	Последовательность символов, используемая в качестве Escape-последовательности
`lineterminator`	`\r\n`	Последовательность символов, используемая объектом `writer` для завершения строки
`quotechar`	`"`	Строка, используемая в роли кавычек для обрамления полей, содержащих специальные значения (одиночный символ)
`quoting`	`QUOTE_MINIMAL`	Управляет описанными ранее режимами заключения в кавычки

Окончание табл. 7.3

Атрибут	По умолчанию	Описание
skipinitialspace	False	Управляет игнорированием пробелов, следующих за разделителем полей

Листинг 7.69. csv_dialect_variations.py

```
import csv
import sys

csv.register_dialect('escaped',
                     escapechar='\\',
                     doublequote=False,
                     quoting=csv.QUOTE_NONE,
                     )
csv.register_dialect('singlequote',
                     quotechar="'",
                     quoting=csv.QUOTE_ALL,
                     )

quoting_modes = {
    getattr(csv, n): n
    for n in dir(csv)
    if n.startswith('QUOTE_')
}

TEMPLATE = '''\
Dialect: "{name}"

  delimiter    = {dl!r:<6}    skipinitialspace = {si!r}
  doublequote  = {dq!r:<6}    quoting          = {qu}
  quotechar    = {qc!r:<6}    lineterminator   = {lt!r}
  escapechar   = {ec!r:<6}
'''

for name in sorted(csv.list_dialects()):
    dialect = csv.get_dialect(name)

    print(TEMPLATE.format(
        name=name,
        dl=dialect.delimiter,
        si=dialect.skipinitialspace,
        dq=dialect.doublequote,
        qu=quoting_modes[dialect.quoting],
        qc=dialect.quotechar,
        lt=dialect.lineterminator,
        ec=dialect.escapechar,
    ))

    writer = csv.writer(sys.stdout, dialect=dialect)
    writer.writerow(
        ('col1', 1, '10/01/2010',
```

```
        'Special chars: " \' {} to parse'.format(
            dialect.delimiter))
    )
    print()
```

Данная программа демонстрирует различия в отображении одних и тех же данных с использованием разных диалектов.

```
$ python3 csv_dialect_variations.py

Dialect: "escaped"

  delimiter    = ','       skipinitialspace = 0
  doublequote = 0          quoting          = QUOTE_NONE
  quotechar    = '"'       lineterminator   = '\r\n'
  escapechar   = '\\'

col1,1,10/01/2010,Special chars: \" ' \, to parse


Dialect: "excel"

  delimiter    = ','       skipinitialspace = 0
  doublequote = 1          quoting          = QUOTE_MINIMAL
  quotechar    = '"'       lineterminator   = '\r\n'
  escapechar   = None

col1,1,10/01/2010,"Special chars: "" ' , to parse"


Dialect: "excel-tab"

  delimiter    = '\t'      skipinitialspace = 0
  doublequote = 1          quoting          = QUOTE_MINIMAL
  quotechar    = '"'       lineterminator   = '\r\n'
  escapechar   = None

col1    1       10/01/2010      "Special chars: "" '    to parse"


Dialect: "singlequote"

  delimiter    = ','       skipinitialspace = 0
  doublequote = 1          quoting          = QUOTE_ALL
  quotechar    = "'"       lineterminator   = '\r\n'
  escapechar   = None

'col1','1','10/01/2010','Special chars: " '' , to parse'


Dialect: "unix"

  delimiter    = ','       skipinitialspace = 0
```

```
  doublequote = 1          quoting           = QUOTE_ALL
  quotechar   = '"'        lineterminator    = '\n'
  escapechar  = None

"col1","1","10/01/2010","Special chars: "" ' , to parse"
```

7.6.3.3. Автоматическое распознавание диалектов

Наилучшим вариантом настройки параметров диалекта для анализа входного файла является тот, при котором корректные настройки заранее известны. В случае данных с неизвестными параметрами диалекта может пригодиться класс `Sniffer`, который поможет выдвинуть обоснованные предположения относительно этого. Методу `sniff()` передается образец входных данных и необязательный аргумент, задающий набор возможных символов-разделителей.

Листинг 7.70. `csv_dialect_sniffer.py`

```
import csv
from io import StringIO
import textwrap

csv.register_dialect('escaped',
                     escapechar='\\',
                     doublequote=False,
                     quoting=csv.QUOTE_NONE)
csv.register_dialect('singlequote',
                     quotechar="'",
                     quoting=csv.QUOTE_ALL)

# Сгенерировать пробные данные для всех известных диалектов
samples = []
for name in sorted(csv.list_dialects()):
    buffer = StringIO()
    dialect = csv.get_dialect(name)
    writer = csv.writer(buffer, dialect=dialect)
    writer.writerow(
        ('col1', 1, '10/01/2010',
         'Special chars " \' {} to parse'.format(
             dialect.delimiter))
    )
    samples.append((name, dialect, buffer.getvalue()))

# Определить диалект для данного образца, а затем использовать
# результат для парсинга данных
sniffer = csv.Sniffer()
for name, expected, sample in samples:
    print('Dialect: "{}"'.format(name))
    print('In: {}'.format(sample.rstrip()))
    dialect = sniffer.sniff(sample, delimiters=',\t')
    reader = csv.reader(StringIO(sample), dialect=dialect)
    print('Parsed:\n  {}\n'.format(
          '\n  '.join(repr(r) for r in next(reader))))
```

Метод sniff() возвращает экземпляр Dialect с параметрами, которые рекомендуется использовать для анализа данных. Конечный результат не всегда оказывается идеальным, как показано ниже на примере диалекта escaped.

```
$ python3 csv_dialect_sniffer.py

Dialect: "escaped"
In: col1,1,10/01/2010,Special chars \" ' \, to parse
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars \\" \' \\'
  ' to parse'

Dialect: "excel"
In: col1,1,10/01/2010,"Special chars "" ' , to parse"
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' , to parse'

Dialect: "excel-tab"
In: col1      1     10/01/2010    "Special chars "" '    to parse"
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' \t to parse'

Dialect: "singlequote"
In: 'col1','1','10/01/2010','Special chars " '' , to parse'
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' , to parse'

Dialect: "unix"
In: "col1","1","10/01/2010","Special chars "" ' , to parse"
Parsed:
  'col1'
  '1'
  '10/01/2010'
  'Special chars " \' , to parse'
```

7.6.4. Использование имен полей

Помимо средств для работы с последовательностями данных модуль csv включает классы, предназначенные для работы со строками как словарями, что позволяет использовать именованные поля. Классы DictReader и DictWriter преобра-

зуют строки в словари вместо списков. Ключи могут передаваться словарю или же будут браться из первой строки входного файла (если она содержит заголовки).

Листинг 7.71. csv_dictreader.py

```
import csv
import sys

with open(sys.argv[1], 'rt') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row)
```

Классы, предназначенные для чтения и записи данных, реализованы в виде оболочек, обертывающих классы на основе последовательностей, и используют те же методы и аргументы. Единственным отличием API чтения является то, что строки возвращаются в виде словарей, а не списков или кортежей.

```
$ python3 csv_dictreader.py testdata.csv

{'Title 2': 'a', 'Title 3': '08/18/07', 'Title 4': 'å', 'Title 1
': '1'}
{'Title 2': 'b', 'Title 3': '08/19/07', 'Title 4': '∫', 'Title 1
': '2'}
{'Title 2': 'c', 'Title 3': '08/20/07', 'Title 4': 'ç', 'Title 1
': '3'}
```

Объекту на основе класса `DictWriter` должен предоставляться список имен полей, чтобы он мог расположить столбцы в правильном порядке при их выводе.

Листинг 7.72. csv_dictwriter.py

```
import csvimport csv
import sys

fieldnames = ('Title 1', 'Title 2', 'Title 3', 'Title 4')
headers = {
    n: n
    for n in fieldnames
}
unicode_chars = 'å∫ç'

with open(sys.argv[1], 'wt') as f:

    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()

    for i in range(3):
        writer.writerow({
            'Title 1': i + 1,
            'Title 2': chr(ord('a') + i),
            'Title 3': '08/{:02d}/07'.format(i + 1),
            'Title 4': unicode_chars[i],
```

```
        })

print(open(sys.argv[1], 'rt').read())
```

Имена полей не записываются автоматически в файл, но их можно записать явным образом, используя метод `writeheader()`.

```
$ python3 csv_dictwriter.py testout.csv

Title 1,Title 2,Title 3,Title 4
1,a,08/01/07,à
2,bʃ,08/02/07,
3,c,08/03/07,ç
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `csv`[23].
- **PEP 305**[24]. *CSV File API*.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `csv` (раздел А.6.12).

[23] https://docs.python.org/3.5/library/csv.html
[24] www.python.org/dev/peps/pep-0305

Глава 8

Сжатие и архивирование данных

Несмотря на непрерывное увеличение емкости запоминающих устройств в современных компьютерных системах, им трудно угнаться за безудержным ростом объема данных, которые приходится обрабатывать. Алгоритмы сжатия без потерь частично заполняют эту брешь, снижая требования к объему памяти, необходимому для хранения данных, за счет дополнительных затрат времени на их сжатие и распаковку. Python включает интерфейсы к большинству популярных библиотек сжатия, обеспечивающих чтение и запись сжатых данных.

Модули `zlib` (раздел 8.1) и `gzip` (раздел 8.2) предоставляют доступ к библиотеке GNU zip, тогда как модуль `bz2` (раздел 8.3) позволяет работать с более новым форматом сжатия bzip2. Оба модуля работают с потоками данных, каков бы ни был их формат, и предоставляют интерфейсы для прозрачного чтения и записи сжатых файлов. Используйте эти модули для сжатия одиночного файла или источника данных.

Стандартная библиотека также включает модули для работы с *архивными* форматами, позволяющими объединять несколько файлов в один, с которым можно обращаться как с отдельной единицей. Модуль `tarfile` (раздел 8.4) предназначен для чтения и записи архивов *tar* в Unix – старого стандарта, который все еще широко используется в силу его гибкости. Модуль `zipfile` (раздел 8.5) работает с архивами, основанными на формате, получившем популярность благодаря программе PKZIP для ПК, которая первоначально работала под управлением MS-DOS и Windows, но в настоящее время используется также на других платформах из-за простоты ее интерфейса и переносимости формата.

8.1. `zlib`: сжатие данных средствами библиотеки GNU `zlib`

Модуль `zlib` предоставляет низкоуровневый интерфейс ко многим функциям библиотеки сжатия из проекта GNU.

8.1.1. Работа с данными в памяти

Простейший способ работы с модулем `zlib` требует хранения всех данных, подлежащих сжатию или восстановлению, в памяти компьютера.

Листинг 8.1. `zlib_memory.py`

```
import zlib
import binascii

original_data = b'This is the original text.'
print('Original     :', len(original_data), original_data)
```

```
compressed = zlib.compress(original_data)
print('Compressed   :', len(compressed),
      binascii.hexlify(compressed))

decompressed = zlib.decompress(compressed)
print('Decompressed :', len(decompressed), decompressed)
```

Каждая из функций `compress()` и `decompress()` получает аргумент в виде байтовой последовательности и возвращает значение этого же типа.

```
$ python3 zlib_memory.py

Original     : 26 b'This is the original text.'
Compressed   : 32 b'789c0bc9c82c5600a2928c5485fca2ccf4ccbcc41c85
92d48a123d007f2f097e'
Decompressed : 26 b'This is the original text.'
```

Как следует из предыдущего примера, в случае небольших наборов данных размер сжатой версии может превышать размер несжатой. И хотя фактический результат зависит от природы конкретных данных, будет интересно получить некоторое представление о величине соответствующих накладных расходов для входных данных небольшого объема.

Листинг 8.2. `zlib_lengths.py`

```
import zlib

original_data = b'This is the original text.'

template = '{:>15} {:>15}'
print(template.format('len(data)', 'len(compressed)'))
print(template.format('-' * 15, '-' * 15))

for i in range(5):
    data = original_data * i
    compressed = zlib.compress(data)
    highlight = '*' if len(data) < len(compressed) else ''
    print(template.format(len(data), len(compressed)), highlight)
```

В выводе символами * отмечены строки, в которых сжатые данные занимают больше места в памяти, чем их несжатая версия.

```
$ python3 zlib_lengths.py
      len(data)   len(compressed)
---------------  ---------------
              0                8 *
             26               32 *
             52               35
             78               35
            104               36
```

Модуль `zlib` поддерживает несколько уровней сжатия, позволяющих регулировать баланс между затратами процессорного времени и степенью сжатия данных. По умолчанию используется уровень сжатия `zlib.Z_DEFAULT_COMPRESSION`

(-1), которому соответствует некоторое значение уровня сжатия, заданное в коде модуля и представляющее разумный компромисс между производительностью и степенью сжатия. В настоящее время ему соответствует уровень 6.

Листинг 8.3. zlib_compresslevel.py

```
import zlib

input_data = b'Some repeated text.\n' * 1024
template = '{:>5} {:>5}'

print(template.format('Level', 'Size'))
print(template.format('-----', '----'))

for i in range(0, 10):
    data = zlib.compress(input_data, i)
    print(template.format(i, len(data)))
```

Если задан уровень 0, то сжатие не выполняется. Уровень 9 требует наибольшего объема вычислений и обеспечивает наибольший коэффициент сжатия. Как показывает этот пример, для заданного набора несколько разных уровней сжатия могут приводить к одному и тому же результату.

```
$ python3 zlib_compresslevel.py

Level   Size
-----   ----
    0  20491
    1    172
    2    172
    3    172
    4     98
    5     98
    6     98
    7     98
    8     98
    9     98
```

8.1.2. Инкрементное сжатие и восстановление данных

В типичных случаях реального применения сжатия и восстановления данных подход, основанный на использовании для этих целей исключительно оперативной памяти компьютера, оказывается непрактичным. Его основным недостатком является то, что система должна иметь запас памяти, достаточный для одновременного хранения несжатой и сжатой версии исходного набора. Альтернативный подход заключается в использовании объектов Compress и Decompress для инкрементного манипулирования данными, что избавляет от необходимости хранить в памяти весь набор целиком.

Листинг 8.4. zlib_incremental.py

```
import zlib
import binascii
```

```
compressor = zlib.compressobj(1)

with open('lorem.txt', 'rb') as input:
    while True:
        block = input.read(64)
        if not block:
            break
        compressed = compressor.compress(block)
        if compressed:
            print('Compressed: {}'.format(
                binascii.hexlify(compressed)))
        else:
            print('buffering...')
    remaining = compressor.flush()
    print('Flushed: {}'.format(binascii.hexlify(remaining)))
```

В этом примере данные читаются небольшими блоками из простого текстового файла и передаются методу `compress()`. Объект-упаковщик поддерживает внутренний буфер сжатых данных. Поскольку алгоритм сжатия зависит от контрольных сумм и минимальных размеров блоков, объект-упаковщик не всегда может быть готов к тому, чтобы возвращать данные сразу же после получения очередной их порции. Если блок сжатых данных не заполнен полностью, возвращается пустая строка. После передачи всех данных метод `flush()` вынуждает объект-упаковщик закрыть завершающий блок и вернуть остаток сжатых данных.

```
$ python3 zlib_incremental.py

Compressed: b'7801'
buffering...
buffering...
buffering...
buffering...
buffering...
Flushed: b'55904b6ac4400c44f73e451da0f129b20c2110c85e696b8c40dde
dd167ce1f7915025a087daa9ef4be8c07e4f21c38962e834b800647435fd3b90
747b2810eb9c4bbcc13ac123bded6e4bef1c91ee40d3c6580e3ff52aad2e8cb2
eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08967d228a
f1447c8ec72e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c4439ec96
05b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a312eebd22
0d4b32441bdc1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32279de'
```

8.1.3. Потоки смешанного содержимого

Объект `Decompress`, возвращаемый методом `decompressobj()`, можно также использовать в тех ситуациях, когда сжатые данные смешиваются с несжатыми.

Листинг 8.5. `zlib_mixed.py`

```
import zlib

lorem = open('lorem.txt', 'rb').read()
compressed = zlib.compress(lorem)
combined = compressed + lorem
```

```
decompressor = zlib.decompressobj()
decompressed = decompressor.decompress(combined)

decompressed_matches = decompressed == lorem
print('Decompressed matches lorem:', decompressed_matches)

unused_matches = decompressor.unused_data == lorem
print('Unused data matches lorem :', unused_matches)
```

После восстановления всех данных атрибут `unused_data` содержит данные, оставшиеся неиспользованными.

```
$ python3 zlib_mixed.py

Decompressed matches lorem: True
Unused data matches lorem : True
```

8.1.4. Контрольные суммы

Помимо функций, обеспечивающих сжатие и распаковку данных, модуль `zlib` включает две функции, предназначенные для вычисления контрольных сумм данных, `adler32()` и `crc32()`. Ни одна контрольная сумма не является криптографически безопасной, и они предназначены исключительно для проверки целостности данных.

Листинг 8.6. `zlib_checksums.py`

```
import zlib

data = open('lorem.txt', 'rb').read()

cksum = zlib.adler32(data)
print('Adler32: {:12d}'.format(cksum))
print('       : {:12d}'.format(zlib.adler32(data, cksum)))

cksum = zlib.crc32(data)
print('CRC-32 : {:12d}'.format(cksum))
print('       : {:12d}'.format(zlib.crc32(data, cksum)))
```

Обе функции получают одинаковые аргументы: байтовую строку, содержащую данные, и оптимальное значение, используемое в качестве начальной точки для контрольной суммы. Каждая из функций возвращает 32-разрядное целое значение, которое может быть передано обратно при последующем вызове в качестве новой начальной точки для получения текущей контрольной суммы.

```
$ python3 zlib_checksums.py

Adler32:   3542251998
       :    669447099
CRC-32 :   3038370516
       :   2870078631
```

8.1.5. Сжатие сетевых данных

Сервер, код которого приведен в следующем листинге, использует объект-упаковщик потока для формирования ответа на запрос, состоящий из имен файлов, посредством записи сжатой версии файла в сокет, используемый для связи с клиентом.

Листинг 8.7. `zlib_server.py`

```
import zlib
import logging
import socketserver
import binascii

BLOCK_SIZE = 64


class ZlibRequestHandler(socketserver.BaseRequestHandler):

    logger = logging.getLogger('Server')

    def handle(self):
        compressor = zlib.compressobj(1)

        # Определение файла, запрашиваемого клиентом
        filename = self.request.recv(1024).decode('utf-8')
        self.logger.debug('client asked for: %r', filename)

        # Отправка порций данных по мере их сжатия
        with open(filename, 'rb') as input:
            while True:
                block = input.read(BLOCK_SIZE)
                if not block:
                    break
                self.logger.debug('RAW %r', block)
                compressed = compressor.compress(block)
                if compressed:
                    self.logger.debug(
                        'SENDING %r',
                        binascii.hexlify(compressed))
                    self.request.send(compressed)
                else:
                    self.logger.debug('BUFFERING')

        # Отправка данных, буферизуемых упаковщиком
        remaining = compressor.flush()
        while remaining:
            to_send = remaining[:BLOCK_SIZE]
            remaining = remaining[BLOCK_SIZE:]
            self.logger.debug('FLUSHING %r',
                              binascii.hexlify(to_send))
            self.request.send(to_send)
        return
```

```
if __name__ == '__main__':
    import socket
    import threading
    from io import BytesIO

    logging.basicConfig(
        level=logging.DEBUG,
        format='%(name)s: %(message)s',
    )
    logger = logging.getLogger('Client')

    # Настроить сервер, выполняющийся в отдельном потоке
    address = ('localhost', 0)  # позволить ядру назначить порт
    server = socketserver.TCPServer(address, ZlibRequestHandler)
    ip, port = server.server_address  # Какой порт назначен?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)
    t.start()

    # Подключиться к серверу в качестве клиента
    logger.info('Contacting server on %s:%s', ip, port)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Запросить файл
    requested_file = 'lorem.txt'
    logger.debug('sending filename: %r', requested_file)
    len_sent = s.send(requested_file.encode('utf-8'))

    # Получить ответ
    buffer = BytesIO()
    decompressor = zlib.decompressobj()
    while True:
        response = s.recv(BLOCK_SIZE)
        if not response:
            break
        logger.debug('READ %r', binascii.hexlify(response))

        # Включить неиспользованные данные
        # при передаче данных распаковщику
        to_decompress = decompressor.unconsumed_tail + response
        while to_decompress:
            decompressed = decompressor.decompress(to_decompress)
            if decompressed:
                logger.debug('DECOMPRESSED %r', decompressed)
                buffer.write(decompressed)
                # Поиск данных, не использованных из-за
                # переполнения буфера
                to_decompress = decompressor.unconsumed_tail
            else:
                logger.debug('BUFFERING')
                to_decompress = None
```

```
    # Обработка данных, оставшихся в буфере распаковщика
    remainder = decompressor.flush()
    if remainder:
        logger.debug('FLUSHED %r', remainder)
        buffer.write(remainder)

    full_response = buffer.getvalue()
    lorem = open('lorem.txt', 'rb').read()
    logger.debug('response matches file contents: %s',
                 full_response == lorem)

    # Освобождение ресурсов
    s.close()
    server.socket.close()
```

Этот листинг включает искусственное разбиение данных на порции, чтобы проиллюстрировать буферизацию, которая выполняется в тех случаях, когда передача данных методу `compress()` или `decompress()` не приводит к заполнению блока сжатых или распакованных выходных данных.

Клиент подключается к сокету и запрашивает файл, после чего получает в цикле блоки сжатых данных. Поскольку блок не всегда может содержать информацию, необходимую для его полной распаковки, остаток полученных ранее данных объединяется с новыми данными и передается распаковщику. После распаковки данных они присоединяются в конец буфера, который сравнивается с содержимым файла в конце цикла обработки.

Предупреждение

В этом сервере не обеспечена надлежащая безопасность. Не выполняйте его в системе, подключенной к Интернету, или в любой незащищенной среде.

```
$ python3 zlib_server.py

Client: Contacting server on 127.0.0.1:53658
Client: sending filename: 'lorem.txt'
Server: client asked for: 'lorem.txt'
Server: RAW b'Lorem ipsum dolor sit amet, consectetuer adipiscin
g elit. Donec\n'
Server: SENDING b'7801'
Server: RAW b'egestas, enim et consectetuer ullamcorper, lectus
ligula rutrum '
Server: BUFFERING
Server: RAW b'leo, a\nelementum elit tortor eu quam. Duis tincid
unt nisi ut ant'
Server: BUFFERING
Server: RAW b'e. Nulla\nfacilisi. Sed tristique eros eu libero.
Pellentesque ve'
Server: BUFFERING
Server: RAW b'l arcu. Vivamus\npurus orci, iaculis ac, suscipit
sit amet, pulvi'
Client: READ b'7801'
Client: BUFFERING
Server: BUFFERING
```

```
Server: RAW b'nar eu,\nlacus.\n'
Server: BUFFERING
Server: FLUSHING b'55904b6ac4400c44f73e451da0f129b20c2110c85e696
b8c40ddedd167ce1f7915025a087daa9ef4be8c07e4f21c38962e834b8006474
35fd3b90747b2810eb9'
Server: FLUSHING b'c4bbcc13ac123bded6e4bef1c91ee40d3c6580e3ff52a
ad2e8cb2eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08
967d228af1447c8ec72'
Client: READ b'55904b6ac4400c44f73e451da0f129b20c2110c85e696b8c4
0ddedd167ce1f7915025a087daa9ef4be8c07e4f21c38962e834b800647435fd
3b90747b2810eb9'
Server: FLUSHING b'e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c
4439ec9605b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a3
12eebd220d4b32441bd'
Client: DECOMPRESSED b'Lorem ipsum dolor sit amet, consectetuer
adi'
Client: READ b'c4bbcc13ac123bded6e4bef1c91ee40d3c6580e3ff52aad2e
8cb2eb6062dad74a89ca904cbb0f2545e0db4b1f2e01955b8c511cb2ac08967d
228af1447c8ec72'
Client: DECOMPRESSED b'piscing elit. Donec\negestas, enim et con
sectetuer ullamcorper, lectus ligula rutrum leo, a\nelementum el
it tortor eu quam. Duis tinci'
Client: READ b'e40c4c714116e60cdef171bb6c0feaa255dff1c507c2c4439
ec9605b7e0ba9fc54bae39355cb89fd6ebe5841d673c7b7bc68a46f575a312ee
bd220d4b32441bd'
Client: DECOMPRESSED b'dunt nisi ut ante. Nulla\nfacilisi. Sed t
ristique eros eu libero. Pellentesque vel arcu. Vivamus\npurus o
rci, iaculis ac'
Server: FLUSHING b'c1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32
279de'
Client: READ b'c1b36ebf0aedef3d57ea4b26dd986dd39af57dfb05d32279d
e'
Client: DECOMPRESSED b', suscipit sit amet, pulvinar eu,\nlacus.
\n'
Client: response matches file contents: True
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `zlib`[1].
- `gzip` (раздел 8.2). Модуль `gzip` включает высокоуровневый (основанный на файлах) интерфейс к библиотеке `zlib`.
- *A Massively Spiffy Yet Delicately Unobtrusive Compression Library*[2] (домашняя страница библиотеки `zlib`).
- *zlib 1.2.11 Manual*[3]. Полная документация библиотеки `zlib`.
- `bz2` (раздел 8.3). Модуль `bz2` предоставляет аналогичный интерфейс к библиотеке сжатия `bzip2`.

[1] `https://docs.python.org/3.5/library/zlib.html`

[2] `www.zlib.net`

[3] `www.zlib.net/manual.html`

8.2. gzip: чтение и запись файлов GNU zip

Модуль gzip предоставляет интерфейс к файлам GNU zip, используя модуль zlib (раздел 8.1) для сжатия и распаковки данных.

8.2.1. Запись сжатых файлов

Функция open() уровня модуля создаст экземпляр класса GzipFile, действующий как обычный объект файла. Предоставляются обычные методы, обеспечивающие запись и чтение байтов.

Листинг 8.8. gzip_write.py

```
import gzip
import io
import os

outfilename = 'example.txt.gz'
with gzip.open(outfilename, 'wb') as output:
    with io.TextIOWrapper(output, encoding='utf-8') as enc:
        enc.write('Contents of the example file go here.\n')

print(outfilename, 'contains', os.stat(outfilename).st_size,
      'bytes')
os.system('file -b --mime {}'.format(outfilename))
```

Чтобы записать данные в сжатый файл, откройте его в режиме 'wb'. В этом примере класс GzipFile обертывается классом TextIOWrapper из модуля io (см. раздел 6.11) для преобразования текста Unicode в байты с целью сжатия.

```
$ python3 gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 75 bytes
```

Степень сжатия файлов можно регулировать с помощью аргумента compresslevel, который может принимать значения в диапазоне от 0 до 9, включая граничные. Более низким значениям соответствует более высокая скорость обработки, но меньшая степень сжатия. Более высоким значениям соответствует меньшая скорость обработки, но большая степень сжатия.

Листинг 8.9. gzip_compresslevel.py

```
import gzip
import io
import os
import hashlib


def get_hash(data):
    return hashlib.md5(data).hexdigest()


data = open('lorem.txt', 'r').read() * 1024
```

```
cksum = get_hash(data.encode('utf-8'))


print('Level  Size        Checksum')
print('-----  ----------  ---------------------------------')
print('data   {:>10}  {}'.format(len(data), cksum))

for i in range(0, 10):
    filename = 'compress-level-{}.gz'.format(i)
    with gzip.open(filename, 'wb', compresslevel=i) as output:
        with io.TextIOWrapper(output, encoding='utf-8') as enc:
            enc.write(data)
    size = os.stat(filename).st_size
    cksum = get_hash(open(filename, 'rb').read())
    print('{:>5d}  {:>10d}  {}'.format(i, size, cksum))
```

В центральном столбце вывода отображаются размеры (в байтах) файлов, полученных в результате сжатия входных данных. В данном конкретном случае повышение степени сжатия не всегда приводит к уменьшению размера выходного файла. Конечные результаты зависят от природы входных данных.

```
$ python3 gzip_compresslevel.py

Level  Size        Checksum
----- ---------- ---------------------------------
data       754688  e4c0f9433723971563f08a458715119c
    0      754848  7f050dafb281c7b9d30e5fccf4e0cf19
    1        9846  3b1708684b3655d136b8dca292f5bbba
    2        8267  48ceb436bf10bc6bbd60489eb285de27
    3        8227  4217663bf275f4241a8b73b1a1cfd734
    4        4167  1a5d9b968520d64ed10a4c125735d8b4
    5        4167  90d85bf6457c2eaf20307deb90d071c6
    6        4167  1798ac0cbd77d79973efd8e222bf85d8
    7        4167  7fe834b01c164a14c2d2d8e5560402e6
    8        4167  03795b47b899384cdb95f99c1b7f9f71
    9        4167  a33be56e455f8c787860f23c3b47b6f1
```

Экземпляр `GzipFile` включает также метод `writelines()`, который можно использовать для записи последовательности строк.

Листинг 8.10. `gzip_writelines.py`

```
import gzip
import io
import itertools
import os

with gzip.open('example_lines.txt.gz', 'wb') as output:
    with io.TextIOWrapper(output, encoding='utf-8') as enc:
        enc.writelines(
            itertools.repeat('The same line, over and over.\n', 10)
        )

os.system('gzcat example_lines.txt.gz')
```

Как и в случае обычных файлов, входные строки должны заканчиваться символом перевода строки.

```
$ python3 gzip_writelines.py

The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
```

8.2.2. Чтение сжатых данных

Чтобы прочитать обратно данные из ранее сжатых файлов, откройте файл в режиме чтения двоичных данных (`'rb'`), предотвращающем преобразование байтов в текст Unicode.

Листинг 8.11. `gzip_read.py`

```
import gzip
import io

with gzip.open('example.txt.gz', 'rb') as input_file:
    with io.TextIOWrapper(input_file, encoding='utf-8') as dec:
        print(dec.read())
```

В этом примере файл, записанный с помощью сценария `gzip_write.py` из предыдущего примера, читается с использованием функции `TextIOWrapper()` для декодирования текста после его распаковки.

```
$ python3 gzip_read.py

Contents of the example file go here
```

Также существует возможность выполнять поиск и чтение лишь определенной части данных в процессе чтения файла.

Листинг 8.12. `gzip_seek.py`

```
import gzip

with gzip.open('example.txt.gz', 'rb') as input_file:
    print('Entire file:')
    all_data = input_file.read()
    print(all_data)

    expected = all_data[5:15]
```

```
    # Вернуться в начало
    input_file.seek(0)

    # Переместиться вперед на 5 байт
    input_file.seek(5)
    print('Starting at position 5 for 10 bytes:')
    partial = input_file.read(10)
    print(partial)

    print()
    print(expected == partial)
```

В методе `seek()` позиция отсчитывается относительно несжатых данных, поэтому вызывающему коду не обязательно знать о том, что файл сжат.

```
$ python3 gzip_seek.py

Entire file:
b'Contents of the example file go here.\n'
Starting at position 5 for 10 bytes:
b'nts of the'

True
```

8.2.3. Работа с потоками

Класс `GzipFile` можно использовать для обертывания других типов потоков данных, тем самым обеспечивая возможность их сжатия. Такой подход полезен в тех случаях, когда данные передаются с использованием сокета или существующего (и уже открытого) дескриптора файла. Кроме того, для выполнения операций над данными в памяти совместно с классом `GzipFile` можно использовать буфер `BytesIO`.

Листинг 8.13. `gzip_BytesIO.py`

```
import gzip
from io import BytesIO
import binascii

uncompressed_data = b'The same line, over and over.\n' * 10
print('UNCOMPRESSED:', len(uncompressed_data))
print(uncompressed_data)

buf = BytesIO()
with gzip.GzipFile(mode='wb', fileobj=buf) as f:
    f.write(uncompressed_data)

compressed_data = buf.getvalue()
print('COMPRESSED:', len(compressed_data))
print(binascii.hexlify(compressed_data))

inbuffer = BytesIO(compressed_data)
with gzip.GzipFile(mode='rb', fileobj=inbuffer) as f:
```

```
    reread_data = f.read(len(uncompressed_data))

print('\nREREAD:', len(reread_data))
print(reread_data)
```

Одним из преимуществ использования класса `GzipFile` поверх модуля `zlib` (раздел 8.1) является то, что он поддерживает файловый API. Однако при чтении ранее сжатых данных методу `read()` необходимо передать размер несжатого файла, иначе возникнет ошибка CRC – возможно, по той причине, что функция `BytesIO()` вернет пустую строку, прежде чем сообщит о достижении конца файла. Работая с потоками сжатых данных, либо добавляйте к ним префикс в виде целого числа, представляющего фактический размер данных, подлежащих чтению, либо используйте API инкрементной распаковки, предлагаемый модулем `zlib`.

```
$ python3 gzip_BytesIO.py

UNCOMPRESSED: 300
b'The same line, over and over.\nThe same line, over and over.\nT
he same line, over and over.\nThe same line, over and over.\nThe
same line, over and over.\nThe same line, over and over.\nThe sam
e line, over and over.\nThe same line, over and over.\nThe same l
ine, over and over.\nThe same line, over and over.\n'
COMPRESSED: 51
b'1f8b08006149aa5702ff0bc94855284ecc4d55c8c9cc4bd551c82f4b2d5248c
c4b0133f4b8424665916401d3e717802c010000'

REREAD: 300
b'The same line, over and over.\nThe same line, over and over.\nT
he same line, over and over.\nThe same line, over and over.\nThe
same line, over and over.\nThe same line, over and over.\nThe sam
e line, over and over.\nThe same line, over and over.\nThe same l
ine, over and over.\nThe same line, over and over.\n'
```

Совет

- Раздел стандартной библиотеки, посвященный модулю `gzip`[4].
- `zlib` (раздел 8.1). Модуль `zlib` предоставляет низкоуровневый интерфейс к средствам сжатия `gzip`.
- `zipfile` (раздел 8.5). Модуль `zipfile` предоставляет доступ к ZIP-архивам.
- `bz2` (раздел 8.3). Модуль `bz2` использует формат сжатия `bzip2`.
- `tarfile` (раздел 8.4). Модуль `tarfile` включает встроенную поддержку чтения сжатых *tar*-архивов.
- `io` (раздел 6.11). Строительные блоки для создания входных и выходных каналов.

8.3. bz2: формат сжатия bzip2

Модуль `bz2` – это интерфейс к библиотеке *bzip2*, используемой для сжатия данных с целью их хранения или передачи. Для этого предоставляются три вида API:

[4] https://docs.python.org/3.5/library/gzip.html

- "одноразовые" функции сжатия/распаковки для работы с большими объемами двоичных данных;
- итеративные объекты сжатия/распаковки для работы с потоками данных;
- объект наподобие файла, позволяющий читать и записывать данные так, как если бы они представляли собой несжатый файл.

8.3.1. Обработка всего набора данных в памяти

Простейший способ работы с модулем `bz2` — загрузить в память сразу все данные, подлежащие сжатию или распаковке, и преобразовать их с помощью функции `compress()` или `decompress()` соответственно.

Листинг 8.14. `bz2_memory.py`

```
import bz2
import binascii

original_data = b'This is the original text.'
print('Original     : {} bytes'.format(len(original_data)))
print(original_data)

print()
compressed = bz2.compress(original_data)
print('Compressed   : {} bytes'.format(len(compressed)))
hex_version = binascii.hexlify(compressed)
for i in range(len(hex_version) // 40 + 1):
    print(hex_version[i * 40:(i + 1) * 40])

print()
decompressed = bz2.decompress(compressed)
print('Decompressed : {} bytes'.format(len(decompressed)))
print(decompressed)
```

Сжатые данные содержат символы, не являющиеся символами ASCII, поэтому перед выводом на печать они должны быть преобразованы в шестнадцатеричное представление. Результаты, выводимые в следующих примерах, переформатированы таким образом, чтобы каждая строка вывода содержала не более 40 символов.

```
$ python3 bz2_memory.py

Original     : 26 bytes
b'This is the original text.'
Compressed   : 62 bytes
b'425a68393141592653591be35a6000002938040'
b'01040022e59c402000314c000111e93d434da223'
b'028cf9e73148cae0a0d6ed7f17724538509016be'
b'35a6'

Decompressed : 26 bytes
b'This is the original text.'
```

В случае коротких текстов размер сжатой версии может значительно превосходить размер оригинала. И хотя фактический результат зависит от природы конкретных данных, будет интересно получить некоторое представление о величине накладных расходов, обусловленных выполнением сжатия.

Листинг 8.15. bz2_lengths.py

```
import bz2

original_data = b'This is the original text.'

fmt = '{:>15}  {:>15}'
print(fmt.format('len(data)', 'len(compressed)'))
print(fmt.format('-' * 15, '-' * 15))

for i in range(5):
    data = original_data * i
    compressed = bz2.compress(data)
    print(fmt.format(len(data), len(compressed)), end='')
    print('*' if len(data) < len(compressed) else '')
```

Строкам вывода, заканчивающимся символами *, соответствуют ситуации, в которых размер сжатых данных превышает размер соответствующих исходных данных.

```
$ python3 bz2_lengths.py

      len(data)  len(compressed)
---------------  ---------------
              0               14*
             26               62*
             52               68*
             78               70
            104               72
```

8.3.2. Инкрементное сжатие и восстановление данных

Основным недостатком этого способа является то, что система должна иметь достаточный запас памяти, чтобы в ней могли одновременно храниться несжатая и сжатая версии исходного набора. Альтернативный подход заключается в использовании объектов `BZ2Compressor` и `BZ2Decompressor` для инкрементного манипулирования данными, что избавляет от необходимости хранить в памяти весь набор целиком.

Листинг 8.16. bz2_incremental.py

```
import bz2
import binascii
import io

compressor = bz2.BZ2Compressor()

with open('lorem.txt', 'rb') as input:
```

```
    while True:
        block = input.read(64)
        if not block:
            break
        compressed = compressor.compress(block)
        if compressed:
            print('Compressed: {}'.format(
                binascii.hexlify(compressed)))
        else:
            print('buffering...')
    remaining = compressor.flush()
    print('Flushed: {}'.format(binascii.hexlify(remaining)))
```

В этом примере данные читаются небольшими блоками из простого текстового файла и передаются методу `compress()`. Объект-упаковщик поддерживает внутренний буфер сжатых данных. Поскольку алгоритм сжатия зависит от контрольных сумм и минимальных размеров блоков, объект-упаковщик не всегда может быть готов к тому, чтобы возвращать данные сразу же после получения очередной их порции. Если блок сжатых данных не заполнен целиком, возвращается пустая строка. После передачи всех данных метод `flush()` вынуждает объект-упаковщик закрыть завершающий блок и вернуть остаток сжатых данных.

```
$ python3 bz2_incremental.py

buffering...
buffering...
buffering...
buffering...
Flushed: b'425a6839314159265359ba83a48c000014d5800010400504052fa
7fe003000ba9112793d4ca789068698a0d1a341901a0d53f4d1119a8d4c9e812
d755a67c10798387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6faf209c5
2a90aaa4d16a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b11660e392
126fc18a72a1088716cc8dedda5d489da410748531278043d70a8a131c2b8adc
d6a221bdb8c7ff76b88c1d5342ee48a70a12175074918'
```

8.3.3. Потоки смешанного содержимого

Класс `BZ2Decompressor` можно использовать также в тех ситуациях, когда сжатые данные смешиваются с несжатыми.

Листинг 8.17. `bz2_mixed.py`

```
import bz2

lorem = open('lorem.txt', 'rt').read().encode('utf-8')
compressed = bz2.compress(lorem)
combined = compressed + lorem

decompressor = bz2.BZ2Decompressor()
decompressed = decompressor.decompress(combined)

decompressed_matches = decompressed == lorem
```

```
print('Decompressed matches lorem:', decompressed_matches)

unused_matches = decompressor.unused_data == lorem
print('Unused data matches lorem :', unused_matches)
```

После восстановления всех данных атрибут `unused_data` содержит данные, оставшиеся неиспользованными.

```
$ python3 bz2_mixed.py

Decompressed matches lorem: True
Unused data matches lorem : True
```

8.3.4. Запись сжатых файлов

Класс `BZ2File` можно использовать для чтения и записи сжатых файлов в формате `bzip2`, применяя обычные методы, предназначенные для чтения и записи данных.

Листинг 8.18. `bz2_file_write.py`

```
import bz2
import io
import os

data = 'Contents of the example file go here.\n'

with bz2.BZ2File('example.bz2', 'wb') as output:
    with io.TextIOWrapper(output, encoding='utf-8') as enc:
        enc.write(data)

os.system('file example.bz2')
```

Чтобы записать данные в сжатый файл, откройте его в режиме `'wb'`. В этом примере класс `BZ2File` обертывается классом `TextIOWrapper` из модуля `io` (раздел 6.11) для преобразования текста Unicode в байты с целью сжатия.

```
$ python3 bz2_file_write.py

example.bz2: bzip2 compressed data, block size = 900k
```

Степень сжатия файлов можно регулировать с помощью аргумента `compresslevel`, который может принимать значения в диапазоне от 1 до 9. Более низким значениям соответствует более высокая скорость обработки, но меньшая степень сжатия. Более высоким значениям соответствует меньшая скорость обработки, но большая степень сжатия.

Листинг 8.19. `bz2_file_compresslevel.py`

```
import bz2
import io
import os
```

```
data = open('lorem.txt', 'r', encoding='utf-8').read() * 1024
print('Input contains {} bytes'.format(
    len(data.encode('utf-8'))))

for i in range(1, 10):
    filename = 'compress-level-{}.bz2'.format(i)
    with bz2.BZ2File(filename, 'wb', compresslevel=i) as output:
        with io.TextIOWrapper(output, encoding='utf-8') as enc:
            enc.write(data)
    os.system('cksum {}'.format(filename))
```

В центральном столбце вывода отображаются размеры (в байтах) файлов, полученных в результате сжатия входных данных. В данном конкретном случае повышение степени сжатия не всегда приводит к уменьшению размера выходного файла. Конечные результаты зависят от природы входных данных.

```
$ python3 bz2_file_compresslevel.py

3018243926 8771 compress-level-1.bz2
1942389165 4949 compress-level-2.bz2
2596054176 3708 compress-level-3.bz2
1491394456 2705 compress-level-4.bz2
1425874420 2705 compress-level-5.bz2
2232840816 2574 compress-level-6.bz2
447681641 2394 compress-level-7.bz2
3699654768 1137 compress-level-8.bz2
3103658384 1137 compress-level-9.bz2
Input contains 754688 bytes
```

Экземпляр `BZ2File` включает также метод `writelines()`, который можно использовать для записи последовательности строк.

Листинг 8.20. `bz2_file_writelines.py`

```
import bz2
import io
import itertools
import os

data = 'The same line, over and over.\n'

with bz2.BZ2File('lines.bz2', 'wb') as output:
    with io.TextIOWrapper(output, encoding='utf-8') as enc:
        enc.writelines(itertools.repeat(data, 10))

os.system('bzcat lines.bz2')
```

Как и в случае обычных файлов, входные строки должны заканчиваться символом перевода строки.

```
$ python3 bz2_file_writelines.py

The same line, over and over.
The same line, over and over.
```

```
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
The same line, over and over.
```

8.3.5. Чтение сжатых файлов

Чтобы прочитать обратно данные из ранее сжатых файлов, откройте файл в режиме чтения двоичных данных (`'rb'`). Значением, возвращенным методом `read()`, будет байтовая строка.

Листинг 8.21. `bz2_file_read.py`

```
import bz2
import io

with bz2.BZ2File('example.bz2', 'rb') as input:
    with io.TextIOWrapper(input, encoding='utf-8') as dec:
        print(dec.read())
```

В этом примере файл, записанный с помощью сценария `bz2_file_write.py` из предыдущего раздела, читается с использованием функции `BZ2File()` для декодирования прочитанных байтов в текст Unicode.

```
$ python3 bz2_file_read.py

Contents of the example file go here.
```

В процессе чтения файла можно переместиться вперед с помощью метода `seek()` и прочитать только часть данных.

Листинг 8.22. `bz2_file_seek.py`

```
import bz2
import contextlib

with bz2.BZ2File('example.bz2', 'rb') as input:
    print('Entire file:')
    all_data = input.read()
    print(all_data)

    expected = all_data[5:15]

    # Вернуться в начало
    input.seek(0)

    # Переместиться вперед на 5 байтов
    input.seek(5)
    print('Starting at position 5 for 10 bytes:')
    partial = input.read(10)
```

```
    print(partial)

    print()
    print(expected == partial)
```

В методе `seek()` позиция отсчитывается относительно несжатых данных, поэтому вызывающему коду необязательно знать о том, что файл сжат. Это позволяет передавать экземпляр `BZ2File` функции, ожидающей получения обычного несжатого файла.

```
$ python3 bz2_file_seek.py

Entire file:
b'Contents of the example file go here.\n'
Starting at position 5 for 10 bytes:
b'nts of the'

True
```

8.3.6. Чтение и запись данных Unicode

Предыдущие примеры были основаны на непосредственном использовании класса `BZ2File` и кодировании/декодировании строк Unicode с помощью класса `io.TextIOWrapper` в необходимых случаях. Этих дополнительных шагов можно избежать, используя функцию `bz2.open()`, которая настраивает класс `io.TextIOWrapper` для автоматического кодирования/декодирования текста.

Листинг 8.23. `bz2_unicode.py`

```
import bz2
import os

data = 'Character with an accent.'

with bz2.open('example.bz2', 'wt', encoding='utf-8') as output:
    output.write(data)

with bz2.open('example.bz2', 'rt', encoding='utf-8') as input:
    print('Full file: {}'.format(input.read()))

# Переместиться в начало символа с акцентом
with bz2.open('example.bz2', 'rt', encoding='utf-8') as input:
    input.seek(18)
    print('One character: {}'.format(input.read(1)))

# Переместиться в середину символа с акцентом
with bz2.open('example.bz2', 'rt', encoding='utf-8') as input:
    input.seek(19)
    try:
        print(input.read(1))
    except UnicodeDecodeError:
        print('ERROR: failed to decode')
```

Дескриптор файла, возвращаемый функцией `open()`, поддерживает метод `seek()`, но пользоваться им следует с осторожностью, поскольку позиция отсчитывается в байтах, а не символах, в результате чего указатель файла может оказаться посередине кода символа.

```
$ python3 bz2_unicode.py

Full file: Character with an åccent.
One character: å
ERROR: failed to decode
```

8.3.7. Сжатие сетевых данных

Код, приведенный в следующем примере, отвечает на запросы, состоящие из имен файлов, посредством записи сжатой версии файла в сокет, используемый для связи с клиентом. Этот листинг включает искусственное разбиение данных на порции, чтобы проиллюстрировать буферизацию, которая выполняется в тех случаях, когда передача данных методу `compress()` или `decompress()` не приводит к заполнению блока сжатых или распакованных выходных данных.

Листинг 8.24. `bz2_server.py`

```
import bz2
import logging
import socketserver
import binascii

BLOCK_SIZE = 32


class Bz2RequestHandler(socketserver.BaseRequestHandler):

    logger = logging.getLogger('Server')

    def handle(self):
        compressor = bz2.BZ2Compressor()

        # Определение файла, запрашиваемого клиентом
        filename = self.request.recv(1024).decode('utf-8')
        self.logger.debug('client asked for: "%s"', filename)

        # Отправка порций данных по мере их сжатия
        with open(filename, 'rb') as input:
            while True:
                block = input.read(BLOCK_SIZE)
                if not block:
                    break
                self.logger.debug('RAW %r', block)
                compressed = compressor.compress(block)
                if compressed:
                    self.logger.debug(
                        'SENDING %r',
                        binascii.hexlify(compressed))
```

```
                self.request.send(compressed)
            else:
                self.logger.debug('BUFFERING')

        # Отправка данных, буферизуемых упаковщиком
        remaining = compressor.flush()
        while remaining:
            to_send = remaining[:BLOCK_SIZE]
            remaining = remaining[BLOCK_SIZE:]
            self.logger.debug('FLUSHING %r',
                              binascii.hexlify(to_send))
            self.request.send(to_send)
        return
```

Основная программа запускает сервер в отдельном потоке, совместно используя классы `SocketServer` и `Bz2RequestHandler`.

```
if __name__ == '__main__':
    import socket
    import sys
    from io import StringIO
    import threading

    logging.basicConfig(level=logging.DEBUG,
                        format='%(name)s: %(message)s',
                        )

    # Настроить сервер, выполняющийся в отдельном потоке
    address = ('localhost', 0)  # позволить ядру назначить порт
    server = socketserver.TCPServer(address, Bz2RequestHandler)
    ip, port = server.server_address  # Какой порт назначен?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)
    t.start()

    logger = logging.getLogger('Client')

    # Подключиться к серверу в качестве клиента
    logger.info('Contacting server on %s:%s', ip, port)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Запросить файл
    requested_file = (sys.argv[0]
                      if len(sys.argv) > 1
                      else 'lorem.txt')
    logger.debug('sending filename: "%s"', requested_file)
    len_sent = s.send(requested_file.encode('utf-8'))

    # Получить ответ
    buffer = StringIO()
    decompressor = bz2.BZ2Decompressor()
```

```
    while True:
        response = s.recv(BLOCK_SIZE)
        if not response:
            break
        logger.debug('READ %r', binascii.hexlify(response))

        # Включить неиспользованные данные
        # при передаче данных распаковщику
        decompressed = decompressor.decompress(response)
        if decompressed:
            logger.debug('DECOMPRESSED %r', decompressed)
            buffer.write(decompressed.decode('utf-8'))
        else:
            logger.debug('BUFFERING')

    full_response = buffer.getvalue()
    lorem = open(requested_file, 'rt').read()
    logger.debug('response matches file contents: %s',
                 full_response == lorem)

    # Освобождение ресурсов
    server.shutdown()
    server.socket.close()
    s.close()
```

Затем программа открывает сокет для подключения к серверу в качестве клиента и запрашивает файл (по умолчанию — *lorem.txt*).

```
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec
egestas, enim et consectetuer ullamcorper, lectus ligula rutrum
leo, a elementum elit tortor eu quam. Duis tincidunt nisi ut ant
e. Nulla facilisi.
```

Предупреждение

В этом сервере не обеспечена надлежащая безопасность. Не выполняйте его в системе, подключенной к Интернету, или в любой незащищенной среде.

Выполнив сценарий `bz2_server.py`, получаем следующий вывод.

```
$ python3 bz2_server.py

Client: Contacting server on 127.0.0.1:57364
Client: sending filename: "lorem.txt"
Server: client asked for: "lorem.txt"
Server: RAW b'Lorem ipsum dolor sit amet, cons'
Server: BUFFERING
Server: RAW b'ectetuer adipiscing elit. Donec\n'
Server: BUFFERING
Server: RAW b'egestas, enim et consectetuer ul'
Server: BUFFERING
Server: RAW b'lamcorper, lectus ligula rutrum '
Server: BUFFERING
```

```
Server: RAW b'leo,\na elementum elit tortor eu '
Server: BUFFERING
Server: RAW b'quam. Duis tincidunt nisi ut ant'
Server: BUFFERING
Server: RAW b'e. Nulla\nfacilisi.\n'
Server: BUFFERING
Server: FLUSHING b'425a6839314159265359ba83a48c000014d5800010400
504052fa7fe003000ba'
Server: FLUSHING b'9112793d4ca789068698a0d1a341901a0d53f4d1119a8
d4c9e812d755a67c107'
Client: READ b'425a6839314159265359ba83a48c000014d58000104005040
52fa7fe003000ba'
Server: FLUSHING b'98387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6
faf209c52a90aaa4d16'
Client: BUFFERING
Server: FLUSHING b'a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b1
1660e392126fc18a72a'
Client: READ b'9112793d4ca789068698a0d1a341901a0d53f4d1119a8d4c9
e812d755a67c107'
Server: FLUSHING b'1088716cc8dedda5d489da410748531278043d70a8a13
1c2b8adcd6a221bdb8c'
Client: BUFFERING
Server: FLUSHING b'7ff76b88c1d5342ee48a70a12175074918'
Client: READ b'98387682c7ca7b5a3bb75da77755eb81c1cb1ca94c4b6faf2
09c52a90aaa4d16'
Client: BUFFERING
Client: READ b'a4a1b9c167a01c8d9ef32589d831e77df7a5753a398b11660
e392126fc18a72a'
Client: BUFFERING
Client: READ b'1088716cc8dedda5d489da410748531278043d70a8a131c2b
8adcd6a221bdb8c'
Client: BUFFERING
Client: READ b'7ff76b88c1d5342ee48a70a12175074918'
Client: DECOMPRESSED b'Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Donec\negestas, enim et consectetuer ullamcorpe
r, lectus ligula rutrum leo,\na elementum elit tortor eu quam. D
uis tincidunt nisi ut ante. Nulla\nfacilisi.\n'
Client: response matches file contents: True
```

Дополнительные ссылки

- Раздел стандартной библиотеки, посвященный модулю `bz2`[5].
- *bzip2 and libbzip2*[6]. Домашняя страница библиотеки *bzip2*.
- `zlib` (см. раздел 8.1). Модуль `zlib` для сжатия GNU zip.
- `gzip` (см. раздел 8.2). Интерфейс доступа к сжатым GNU zip-файлам.
- `io` (раздел 390). Строительные блоки для создания каналов ввода и вывода.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `bz2` (раздел A.6.7).

[5] `https://docs.python.org/3.5/library/bz2.html`

[6] `www.bzip.org`

8.4. tarfile: доступ к архивам Tar

Модуль tarfile позволяет читать и записывать *tar*-архивы Unix, в том числе сжатые файлы. В дополнение к стандартам POSIX предоставляется поддержка нескольких расширений GNU tar. Также обеспечивается работа со специальными типами файлов Unix, такими как жесткие и символические ссылки, и узлами устройств.

Примечание

Несмотря на то что модуль tarfile реализует формат Unix, его также можно использовать для создания и чтения *tar*-архивов в Windows.

8.4.1. Тестирование tar-файлов

Функция is_tarfile() возвращает булево значение, указывающее на то, ссылается ли имя файла, переданное в качестве аргумента, на допустимый *tar*-архив.

Листинг 8.25. tarfile_is_tarfile.py

```
import tarfile

for filename in ['README.txt', 'example.tar',
                 'bad_example.tar', 'notthere.tar']:
    try:
        print('{:>15}  {}'.format(filename, tarfile.is_tarfile(
            filename)))
    except IOError as err:
        print('{:>15}  {}'.format(filename, err))
```

Если такого файла не существует, функция is_tarfile() возбуждает исключение IOError.

```
$ python3 tarfile_is_tarfile.py

     README.txt False
    example.tar True
bad_example.tar False
   notthere.tar [Errno 2] No such file or directory:
'notthere.tar'
```

8.4.2. Чтение метаданных из архива

Класс TarFile позволяет работать непосредственно с *tar*-архивом. Данный класс поддерживает методы, предназначенные как для чтения существующих архивов, так и для изменения архивов путем добавления в них файлов. Для чтения имен файлов в существующем архиве используется метод getnames().

Листинг 8.26. tarfile_getnames.py

```
import tarfile

with tarfile.open('example.tar', 'r') as t:
    print(t.getnames())
```

Возвращаемым значением является список строк с именами содержимого архива.

```
$ python3 tarfile_getnames.py

['index.rst', 'README.txt']
```

Метаданные, содержащие информацию об элементах архива, доступны в виде экземпляров объектов `TarInfo`.

Листинг 8.27. `tarfile_getmembers.py`

```
import tarfile
import time

with tarfile.open('example.tar', 'r') as t:
    for member_info in t.getmembers():
        print(member_info.name)
        print('  Modified:', time.ctime(member_info.mtime))
        print('  Mode    :', oct(member_info.mode))
        print('  Type    :', member_info.type)
        print('  Size    :', member_info.size, 'bytes')
        print()
```

Метаданные загружаются с помощью методов `getmembers()` и `getmember()`.

```
$ python3 tarfile_getmembers.py

index.rst
  Modified: Fri Aug 19 16:27:54 2016
     Mode : 0o644
     Type : b'0'
     Size : 9878 bytes

README.txt
  Modified: Fri Aug 19 16:27:54 2016
     Mode : 0o644
     Type : b'0'
     Size : 75 bytes
```

Если имя элемента архива известно заранее, то объект `TarInfo` этого элемента можно извлечь с помощью метода `getmember()`.

Листинг 8.28. `tarfile_getmember.py`

```
import tarfile
import time

with tarfile.open('example.tar', 'r') as t:
    for filename in ['README.txt', 'notthere.txt']:
        try:
            info = t.getmember(filename)
        except KeyError:
            print('ERROR: Did not find {} in tar archive'.format(
```

```
                filename))
        else:
            print('{} is {:d} bytes'.format(
                info.name, info.size))
```

Если данный элемент отсутствует в архиве, метод `getmember()` возбуждает исключение `KeyError`.

```
$ python3 tarfile_getmember.py

README.txt is 75 bytes
ERROR: Did not find notthere.txt in tar archive
```

8.4.3. Извлечение файлов из архива

Для доступа к данным, содержащимся в элементе архива, из программы используется метод `extractfile()`, которому передается имя элемента.

Листинг 8.29. `tarfile_extractfile.py`

```
import tarfile

with tarfile.open('example.tar', 'r') as t:
    for filename in ['README.txt', 'notthere.txt']:
        try:
            f = t.extractfile(filename)
        except KeyError:
            print('ERROR: Did not find {} in tar archive'.format(
                filename))
        else:
            print(filename, ':')
            print(f.read().decode('utf-8'))
```

Возвращаемым значением является файловый объект, из которого может быть прочитано содержимое элемента архива.

```
$ python3 tarfile_extractfile.py

README.txt :
The examples for the tarfile module use this file and
example.tar as data.

ERROR: Did not find notthere.txt in tar archive
```

Чтобы распаковать архив и записать файлы в файловую систему, используйте метод `extract()` или `extractall()`.

Листинг 8.30. `tarfile_extract.py`

```
import tarfile
import os

os.mkdir('outdir')
with tarfile.open('example.tar', 'r') as t:
```

```
    t.extract('README.txt', 'outdir')
print(os.listdir('outdir'))
```

Элемент (или элементы) читается из архива и записывается в файловую систему, начиная с каталога, указанного в составе аргументов.

```
$ python3 tarfile_extract.py

['README.txt']
```

В документации стандартной библиотеки обращается внимание на то, что метод `extractall()` является более безопасным по сравнению с методом `extract()`, особенно при работе с потоковыми данными, когда перемещение к предыдущим порциям данных для их прочтения невозможно. Именно этот метод следует использовать в большинстве случаев.

Листинг 8.31. `tarfile_extractall.py`

```
import tarfile
import os

os.mkdir('outdir')
with tarfile.open('example.tar', 'r') as t:
    t.extractall('outdir')
print(os.listdir('outdir'))
```

В качестве первого аргумента метод `extractall()` получает имя каталога, в который должны быть записаны файлы.

```
$ python3 tarfile_extractall.py

['README.txt', 'index.rst']
```

Чтобы извлечь из архива только определенные файлы, следует передать методу `extractall()` имена этих файлов или контейнеры метаданных `TarInfo`.

Листинг 8.32. `tarfile_extractall_members.py`

```
import tarfile
import os

os.mkdir('outdir')
with tarfile.open('example.tar', 'r') as t:
    t.extractall('outdir',
                 members=[t.getmember('README.txt')],
                 )
print(os.listdir('outdir'))
```

При предоставлении списка элементов извлекаются лишь файлы с указанными именами.

```
$ python3 tarfile_extractall_members.py

['README.txt']
```

8.4.4. Создание новых архивов

Чтобы создать новый архив, откройте экземпляр `TarFile` в режиме `'w'`.

Листинг 8.33. `tarfile_add.py`

```
import tarfile

print('creating archive')
with tarfile.open('tarfile_add.tar', mode='w') as out:
    print('adding README.txt')
    out.add('README.txt')

print()
print('Contents:')
with tarfile.open('tarfile_add.tar', mode='r') as t:
    for member_info in t.getmembers():
        print(member_info.name)
```

Если файл с таким именем уже существует, его содержимое усекается и архив создается заново. Файлы добавляются с помощью метода `add()`.

```
$ python3 tarfile_add.py

creating archive
adding README.txt

Contents:
README.txt
```

8.4.5. Использование альтернативных имен файлов в архиве

Файл можно добавить в архив с использованием имени, отличного от исходного, создав объект `TarInfo` с альтернативным именем и передав его методу `addfile()`.

Листинг 8.34. `tarfile_addfile.py`

```
import tarfile

print('creating archive')
with tarfile.open('tarfile_addfile.tar', mode='w') as out:
    print('adding README.txt as RENAMED.txt')
    info = out.gettarinfo('README.txt', arcname='RENAMED.txt')
    out.addfile(info)

print()
print('Contents:')
with tarfile.open('tarfile_addfile.tar', mode='r') as t:
    for member_info in t.getmembers():
        print(member_info.name)
```

Архив включает лишь файл с измененным именем.

```
$ python3 tarfile_addfile.py

creating archive
adding README.txt as RENAMED.txt

Contents:
RENAMED.txt
```

8.4.6. Запись данных из источников, отличных от файлов

Иногда требуется записать в архив данные непосредственно из памяти. Вместо того чтобы сначала записывать данные в файл, а затем добавлять файл в архив, можно воспользоваться методом `addfile()` для добавления данных с использованием открытого дескриптора файлового объекта, который возвращает байты.

Листинг 8.35. `tarfile_addfile_string.py`

```
import io
import tarfile

text = 'This is the data to write to the archive.'
data = text.encode('utf-8')

with tarfile.open('addfile_string.tar', mode='w') as out:
    info = tarfile.TarInfo('made_up_file.txt')
    info.size = len(data)
    out.addfile(info, io.BytesIO(data))

print('Contents:')
with tarfile.open('addfile_string.tar', mode='r') as t:
    for member_info in t.getmembers():
        print(member_info.name)
        f = t.extractfile(member_info)
        print(f.read().decode('utf-8'))
```

При конструировании объекта `TarInfo` элементу архива можно присвоить любое имя. После задания объема данных они записываются в архив с помощью метода `addfile()` и буфера `BytesIO` в качестве источника данных.

```
$ python3 tarfile_addfile_string.py

Contents:
made_up_file.txt
This is the data to write to the archive.
```

8.4.7. Присоединение файла к архиву

В дополнение к созданию новых архивов возможно присоединение к существующему файлу. В этом случае следует использовать режим `'a'`.

Листинг 8.36. tarfile_append.py

```
import tarfile

print('creating archive')
with tarfile.open('tarfile_append.tar', mode='w') as out:
    out.add('README.txt')

print('contents:',)
with tarfile.open('tarfile_append.tar', mode='r') as t:
    print([m.name for m in t.getmembers()])

print('adding index.rst')
with tarfile.open('tarfile_append.tar', mode='a') as out:
    out.add('index.rst')

print('contents:',)
with tarfile.open('tarfile_append.tar', mode='r') as t:
    print([m.name for m in t.getmembers()])
```

Результирующий архив содержит два элемента.

```
$ python3 tarfile_append.py

creating archive
contents:
['README.txt']
adding index.rst
contents:
['README.txt', 'index.rst']
```

8.4.8. Работа со сжатыми архивами

Помимо обычных архивных *tar*-файлов модуль `tarfile` может работать с архивами, сжатыми с использованием протоколов сжатия `gzip` и `bzip2`. Чтобы открыть сжатый архив, следует изменить строку режима открытия, передаваемую методу `open()`, включив в нее суффикс `":gz"` или `":bz2"`, в зависимости от используемого метода сжатия.

Листинг 8.37. tarfile_compression.py

```
import tarfile
import os

fmt = '{:<30} {:<10}'
print(fmt.format('FILENAME', 'SIZE'))
print(fmt.format('README.txt', os.stat('README.txt').st_size))

FILES = [
    ('tarfile_compression.tar', 'w'),
    ('tarfile_compression.tar.gz', 'w:gz'),
    ('tarfile_compression.tar.bz2', 'w:bz2'),
]
```

```
for filename, write_mode in FILES:
    with tarfile.open(filename, mode=write_mode) as out:
        out.add('README.txt')

    print(fmt.format(filename, os.stat(filename).st_size),
          end=' ')
    print([
        m.name
        for m in tarfile.open(filename, 'r:*').getmembers()
    ])
```

Открывая существующий архив для чтения, следует указать режим `"r:*"`, позволяющий модулю `tarfile` автоматически определить используемый метод сжатия.

```
$ python3 tarfile_compression.py

FILENAME                        SIZE
README.txt                      75
tarfile_compression.tar         10240      ['README.txt']
tarfile_compression.tar.gz      213        ['README.txt']
tarfile_compression.tar.bz2     199        ['README.txt']
```

Дополнительные ссылки

- Раздел стандартной библиотеки, посвященный модулю `tarfile`[7].
- Руководство по использованию формата GNU tar[8]. Описание формата *tar*, включая расширения.
- `zipfile` (раздел 8.5). Доступ к архивам ZIP.
- `gzip` (см. раздел 8.2). Реализация протокола сжатия GNU zip.
- `bz2` (раздел 8.3). Реализация протокола сжатия Bzip2.

8.5. `zipfile`: доступ к ZIP-архивам

Модуль `zipfile` позволяет осуществлять чтение и запись архивных файлов в формате ZIP, получившем популярность благодаря программе PKZIP для ПК.

8.5.1. Тестирование ZIP-файлов

Функция `is_zipfile()` возвращает булево значение, указывающее на то, ссылается ли имя файла, переданное в качестве аргумента, на допустимый ZIP-архив.

Листинг 8.38. `zipfile_is_zipfile.py`

```
import zipfile

for filename in ['README.txt', 'example.zip',
                 'bad_example.zip', 'notthere.zip']:
    print('{:>15}  {}'.format(
        filename, zipfile.is_zipfile(filename)))
```

[7] https://docs.python.org/3.5/library/tarfile.html

[8] www.gnu.org/software/tar/manual/html_node/Standard.html

Если такого файла не существует, функция `is_zipfile()` возвращает значение `False`.

```
$ python3 zipfile_is_zipfile.py

     README.txt  False
    example.zip  True
bad_example.zip  False
   notthere.zip  False
```

8.5.2. Чтение метаданных из архива

Класс `ZipFile` позволяет работать непосредственно с ZIP-архивом. Данный класс поддерживает методы, предназначенные как для чтения существующих архивов, так и для изменения архивов путем добавления в них файлов.

Листинг 8.39. `zipfile_namelist.py`

```
import zipfile

with zipfile.ZipFile('example.zip', 'r') as zf:
    print(zf.namelist())
```

Метод `namelist()` возвращает имена файлов, содержащихся в существующем архиве.

```
$ python3 zipfile_namelist.py

['README.txt']
```

Однако список имен — это всего лишь часть информации, которую можно получить из архива. Для доступа к метаданным, касающимся содержимого ZIP-архива, следует использовать методы `infolist()` и `getinfo()`.

Листинг 8.40. `zipfile_infolist.py`

```
import datetime
import zipfile


def print_info(archive_name):
    with zipfile.ZipFile(archive_name) as zf:
        for info in zf.infolist():
            print(info.filename)
            print('  Comment     :', info.comment)
            mod_date = datetime.datetime(*info.date_time)
            print('  Modified    :', mod_date)
            if info.create_system == 0:
                system = 'Windows'
            elif info.create_system == 3:
                system = 'Unix'
            else:
                system = 'UNKNOWN'
            print('  System      :', system)
```

```
        print('  ZIP version :', info.create_version)
        print('  Compressed  :', info.compress_size, 'bytes')
        print('  Uncompressed:', info.file_size, 'bytes')
        print()


if __name__ == '__main__':
    print_info('example.zip')
```

Помимо выведенных здесь полей существуют и другие поля, но расшифровка их значений требует внимательного прочтения документа *PKZIP Application Note*, содержащего спецификацию файлов формата ZIP.

```
$ python3 zipfile_infolist.py

README.txt
  Comment     : b''
  Modified    : 2010-11-15 06:48:02
  System      : Unix
  ZIP version : 30
  Compressed  : 65 bytes
  Uncompressed: 76 bytes
```

Если имя элемента архива известно заранее, то объект `ZipInfo` этого элемента можно извлечь с помощью метода `getinfo()`.

Листинг 8.41. zipfile_getinfo.py

```
import zipfile

with zipfile.ZipFile('example.zip') as zf:
    for filename in ['README.txt', 'notthere.txt']:
        try:
            info = zf.getinfo(filename)
        except KeyError:
            print('ERROR: Did not find {} in zip file'.format(
                filename))
        else:
            print('{} is {} bytes'.format(
                info.filename, info.file_size))
```

Если элемент отсутствует в архиве, то метод `getinfo()` возбуждает исключение `KeyError`.

```
$ python3 zipfile_getinfo.py

README.txt is 76 bytes
ERROR: Did not find notthere.txt in zip file
```

8.5.3. Извлечение файлов из архива

Для доступа к данным, содержащимся в элементе архива, используется метод `read()`, которому передается имя элемента.

Листинг 8.42. zipfile_read.py

```
import zipfile

with zipfile.ZipFile('example.zip') as zf:
    for filename in ['README.txt', 'notthere.txt']:
        try:
            data = zf.read(filename)
        except KeyError:
            print('ERROR: Did not find {} in zip file'.format(
                filename))
        else:
            print(filename, ':')
            print(data)
        print()
```

Данные распаковываются автоматически, если в этом есть необходимость.

```
$ python3 zipfile_read.py

README.txt :
b'The examples for the zipfile module use \nthis file and exampl
e.zip as data.\n'

ERROR: Did not find notthere.txt in zip file
```

8.5.4. Создание новых архивов

Чтобы создать новый архив, создайте экземпляр `ZipFile` в режиме `'w'`. Если файл с таким именем уже существует, то его содержимое усекается и архив создается заново. Для добавления в архив новых файлов используется метод `write()`.

Листинг 8.43. zipfile_write.py

```
from zipfile_infolist import print_info
import zipfile

print('creating archive')
with zipfile.ZipFile('write.zip', mode='w') as zf:
    print('adding README.txt')
    zf.write('README.txt')

print()
print_info('write.zip')
```

По умолчанию содержимое архива не сжимается.

```
$ python3 zipfile_write.py

creating archive
adding README.txt

README.txt
```

```
  Comment     : b''
  Modified    : 2016-08-07 13:31:24
  System      : Unix
  ZIP version : 20
  Compressed  : 76 bytes
  Uncompressed: 76 bytes
```

Чтобы добавить сжатие, необходимо использовать модуль `zlib` (см. раздел 8.1). Если модуль `zlib` доступен, то режим сжатия для отдельных файлов или архива в целом можно установить с помощью константы `zipfile.ZIP_DEFLATED`. По умолчанию используется режим сжатия `zipfile.ZIP_STORED`, которому соответствует добавление входных данных в архив без сжатия.

Листинг 8.44. `zipfile_write_compression.py`

```
from zipfile_infolist import print_info
import zipfile
try:
    import zlib
    compression = zipfile.ZIP_DEFLATED
except:
    compression = zipfile.ZIP_STORED

modes = {
    zipfile.ZIP_DEFLATED: 'deflated',
    zipfile.ZIP_STORED: 'stored',
}

print('creating archive')
with zipfile.ZipFile('write_compression.zip', mode='w') as zf:
    mode_name = modes[compression]
    print('adding README.txt with compression mode', mode_name)
    zf.write('README.txt', compress_type=compression)

print()
print_info('write_compression.zip')
```

На этот раз элемент архива подвергается сжатию.

```
$ python3 zipfile_write_compression.py

creating archive
adding README.txt with compression mode deflated

README.txt
  Comment     : b''
  Modified    : 2016-08-07 13:31:24
  System      : Unix
  ZIP version : 20
  Compressed  : 65 bytes
  Uncompressed: 76 bytes
```

8.5.5. Использование альтернативных имен файлов в архиве

Файл можно добавить в архив с использованием имени, отличного от исходного, передав методу `write()` требуемое альтернативное имя файла в архиве.

Листинг 8.45. `zipfile_write_arcname.py`

```
from zipfile_infolist import print_info
import zipfile

with zipfile.ZipFile('write_arcname.zip', mode='w') as zf:
    zf.write('README.txt', arcname='NOT_README.txt')

print_info('write_arcname.zip')
```

Архив включает лишь файл с измененным именем.

```
$ python3 zipfile_write_arcname.py

NOT_README.txt
  Comment     : b''
  Modified    : 2016-08-07 13:31:24
  System      : Unix
  ZIP version : 20
  Compressed  : 76 bytes
  Uncompressed: 76 bytes
```

8.5.6. Запись данных из источников, отличных от файлов

Иногда требуется записать в ZIP-архив данные, источником которых не является файл. Вместо того чтобы сначала записывать данные в файл, а затем добавлять файл в ZIP-архив, можно воспользоваться методом `writestr()` для непосредственного добавления строки байтов в архив.

Листинг 8.46. `zipfile_writestr.py`

```
from zipfile_infolist import print_info
import zipfile

msg = 'This data did not exist in a file.'
with zipfile.ZipFile('writestr.zip',
                     mode='w',
                     compression=zipfile.ZIP_DEFLATED,
                     ) as zf:
    zf.writestr('from_string.txt', msg)

print_info('writestr.zip')

with zipfile.ZipFile('writestr.zip', 'r') as zf:
    print(zf.read('from_string.txt'))
```

В данном случае для сжатия данных был использован аргумент `compress_type` конструктора `ZipFile`, поскольку для метода `writestr()` не предусмотрен аргумент, устанавливающий сжатие.

```
$ python3 zipfile_writestr.py

from_string.txt
  Comment     : b''
  Modified    : 2016-12-29 12:14:42
  System      : Unix
  ZIP version : 20
  Compressed  : 36 bytes
  Uncompressed: 34 bytes

b'This data did not exist in a file.'
```

8.5.7. Запись с помощью экземпляра ZipInfo

Как правило, дата изменения вычисляется при добавлении файла или строки в архив. Можно самостоятельно определить дату изменения и другие метаданные, передав экземпляр `ZipInfo` методу `writestr()`.

Листинг 8.47. zipfile_writestr_zipinfo.py

```
import time
import zipfile
from zipfile_infolist import print_info

msg = b'This data did not exist in a file.'

with zipfile.ZipFile('writestr_zipinfo.zip',
                     mode='w',
                     ) as zf:
    info = zipfile.ZipInfo('from_string.txt',
                           date_time=time.localtime(time.time()),
                           )
    info.compress_type = zipfile.ZIP_DEFLATED
    info.comment = b'Remarks go here'
    info.create_system = 0
    zf.writestr(info, msg)

print_info('writestr_zipinfo.zip')
```

В этом примере в качестве даты изменения устанавливается текущее время, данные сжимаются, а для атрибута `create_system` устанавливается ложное значение. Кроме того, с новым файлом связывается комментарий.

```
$ python3 zipfile_writestr_zipinfo.py

from_string.txt
  Comment     : b'Remarks go here'
  Modified    : 2016-12-29 12:14:42
```

```
  System      : Windows
  ZIP version : 20
  Compressed  : 36 bytes
  Uncompressed: 34 bytes
```

8.5.8. Присоединение архива к файлу

В дополнение к созданию новых архивов возможно присоединение файла к существующему архиву или добавление архива в конец существующего файла (например, *.exe*-афайла для создания самораспаковывающегося архива). Чтобы открыть файл в режиме присоединения, следует использовать режим `'a'`.

Листинг 8.48. `zipfile_append.py`

```
from zipfile_infolist import print_info
import zipfile

print('creating archive')
with zipfile.ZipFile('append.zip', mode='w') as zf:
    zf.write('README.txt')

print()
print_info('append.zip')

print('appending to the archive')
with zipfile.ZipFile('append.zip', mode='a') as zf:
    zf.write('README.txt', arcname='README2.txt')

print()
print_info('append.zip')
```

Результирующий архив содержит два элемента.

```
$ python3 zipfile_append.py

creating archive

README.txt
  Comment     : b''
  Modified    : 2016-08-07 20:31:24
  System      : Windows
  ZIP version : 20
  Compressed  : 76 bytes
  Uncompressed: 76 bytes

appending to the archive

README.txt
  Comment     : b''
  Modified    : 2016-08-07 20:31:24
  System      : Windows
  ZIP version : 20
  Compressed  : 76 bytes
  Uncompressed: 76 bytes
```

```
README2.txt
  Comment      : b''
  Modified     : 2016-08-07 20:31:24
  System       : Windows
  ZIP version  : 20
  Compressed   : 76 bytes
  Uncompressed: 76 bytes
```

8.5.9. ZIP-архивы Python

Python может импортировать модули из ZIP-архивов с помощью модуля `zipimport` (раздел 19.3), если эти архивы указаны в списке путей `sys.path`. Чтобы сконструировать модуль, пригодный для такого использования, можно воспользоваться классом `PyZipFile`. Дополнительный метод `writepy()` сообщает экземпляру `PyZipFile` о необходимости поиска *.py*-файлов в каталоге и добавления соответствующих *.pyo*- или *.pyc*-файлов в архив. Отсутствие этих компилированных форм приводит к созданию и добавлению нужных *.pyc*-файлов.

Листинг 8.49. `zipfile_pyzipfile.py`

```
import sys
import zipfile

if __name__ == '__main__':
    with zipfile.PyZipFile('pyzipfile.zip', mode='w') as zf:
        zf.debug = 3
        print('Adding python files')
        zf.writepy('.')
    for name in zf.namelist():
        print(name)

    print()
    sys.path.insert(0, 'pyzipfile.zip')
    import zipfile_pyzipfile
    print('Imported from:', zipfile_pyzipfile.__file__)
```

Установка значения 3 для атрибута `debug` экземпляра `PyZipFile` включает режим подробных отладочных сообщений, в результате чего они выводятся каждый раз, когда программа компилирует обнаруженный ею *.py*-файл.

```
$ python3 zipfile_pyzipfile.py

Adding python files
Adding files from directory .
Compiling ./zipfile_append.py
Adding zipfile_append.pyc
Compiling ./zipfile_getinfo.py
Adding zipfile_getinfo.pyc
Compiling ./zipfile_infolist.py
Adding zipfile_infolist.pyc
Compiling ./zipfile_is_zipfile.py
Adding zipfile_is_zipfile.pyc
```

```
Compiling ./zipfile_namelist.py
Adding zipfile_namelist.pyc
Compiling ./zipfile_printdir.py
Adding zipfile_printdir.pyc
Compiling ./zipfile_pyzipfile.py
Adding zipfile_pyzipfile.pyc
Compiling ./zipfile_read.py
Adding zipfile_read.pyc
Compiling ./zipfile_write.py
Adding zipfile_write.pyc
Compiling ./zipfile_write_arcname.py
Adding zipfile_write_arcname.pyc
Compiling ./zipfile_write_compression.py
Adding zipfile_write_compression.pyc
Compiling ./zipfile_writestr.py
Adding zipfile_writestr.pyc
Compiling ./zipfile_writestr_zipinfo.py
Adding zipfile_writestr_zipinfo.pyc
zipfile_append.pyc
zipfile_getinfo.pyc
zipfile_infolist.pyc
zipfile_is_zipfile.pyc
zipfile_namelist.pyc
zipfile_printdir.pyc
zipfile_pyzipfile.pyc
zipfile_read.pyc
zipfile_write.pyc
zipfile_write_arcname.pyc
zipfile_write_compression.pyc
zipfile_writestr.pyc
zipfile_writestr_zipinfo.pyc

Imported from: pyzipfile.zip/zipfile_pyzipfile.pyc
```

8.5.10. Ограничения

Модуль `zipfile` не поддерживает ZIP-файлы с присоединенными комментариями или многодисковыми архивами. В то же время он поддерживает ZIP-файлы размером свыше 4 Гбайт, использующие расширения ZIP64.

Дополнительные ссылки

- Раздел стандартной библиотеки, посвященный модулю `zipfile`[9].
- `zlib` (см. раздел 8.1). Библиотека сжатия ZIP.
- `tarfile` (см. раздел 8.4). Чтение и запись архивов *tar*.
- `zipimport` (раздел 19.3). Импорт модулей Python из ZIP-архива.
- *PKZIP Application Note*[10]. Официальная спецификация формата ZIP-архива.

[9] https://docs.python.org/3.5/library/zipfile.html

[10] www.pkware.com/documents/casestudies/APPNOTE.TXT

Глава 9

Криптография

Шифрование сообщений обеспечивает возможность проверки их целостности и сокрытие содержимого от неавторизованных пользователей. Поддержка криптографических средств в Python включает модуль hashlib (раздел 9.1), генерирующий цифровые подписи сообщений с использованием таких стандартных алгоритмов вычисления контрольных сумм (хеш-кодов), как MD5 и SHA, и модуль hmac (раздел 9.2), позволяющий удостовериться в том, что сообщение не было изменено в процессе передачи.

9.1. hashlib: криптографическое хеширование

Модуль hashlib определяет интерфейс доступа к различным алгоритмам криптографического хеширования. Для работы с определенным алгоритмом следует использовать соответствующую функцию-конструктор или функцию new(), позволяющую создать новый объект хеширования. Начиная с этого момента все объекты используют один и тот же API, независимо от того, какой именно алгоритм задействован.

9.1.1. Алгоритмы хеширования

Поскольку модуль hashlib связан с пакетом OpenSSL, доступны все алгоритмы, предоставляемые этой библиотекой, включая следующие:

- MD5;
- SHA1;
- SHA224;
- SHA256;
- SHA384;
- SHA512.

Одни алгоритмы доступны на всех платформах, доступность других зависит от базовых библиотек. Списки алгоритмов каждой из этих категорий содержатся в атрибутах algorithms_guaranteed и algorithms_available соответственно.

Листинг 9.1. hashlib_algorithms.py

```
import hashlib


print('Guaranteed:\n{}\n'.format(
    ', '.join(sorted(hashlib.algorithms_guaranteed))))
print('Available:\n{}'.format(
    ', '.join(sorted(hashlib.algorithms_available))))
```

```
$ python3 hashlib_algorithms.py

Guaranteed:
md5, sha1, sha224, sha256, sha384, sha512

Available:
DSA, DSA-SHA, MD4, MD5, MDC2, RIPEMD160, SHA, SHA1, SHA224,
SHA256, SHA384, SHA512, dsaEncryption, dsaWithSHA,
ecdsa-with-SHA1, md4, md5, mdc2, ripemd160, sha, sha1, sha224,
sha256, sha384, sha512
```

9.1.2. Пробный набор данных

Во всех примерах, приведенных в этом разделе, используется один и тот же набор данных (листинг 9.2).

Листинг 9.2. `hashlib_data.py`

```
import hashlib

lorem = '''Lorem ipsum dolor sit amet, consectetur adipisicing
elit, sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis
aute irure dolor in reprehenderit in voluptate velit esse cillum
dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt
mollit anim id est laborum.'''
```

9.1.3. Алгоритм MD5

Чтобы вычислить хеш-код MD5 (или *дайджест* сообщения) для блока данных (в данном случае это строка Unicode, преобразованная в байтовую строку), прежде всего следует создать объект, вычисляющий хеш-код (объект хеширования), затем добавить данные и, наконец, вызвать метод `digest()` или `hexdigest()`.

Листинг 9.3. `hashlib_md5.py`

```
import hashlib

from hashlib_data import lorem

h = hashlib.md5()
h.update(lorem.encode('utf-8'))
print(h.hexdigest())
```

Использование в этом примере метода `hexdigest()` вместо `digest()` обеспечивает получение более удобочитаемого вывода. Если приемлемо двоичное значение дайджеста, то используйте метод `digest()`.

```
$ python3 hashlib_md5.py

3f2fd2c9e25d60fb0fa5d593b802b7a8
```

9.1.4. Алгоритм SHA1

Дайджест SHA1 вычисляется аналогичным образом.

Листинг 9.4. hashlib_sha1.py

```
import hashlib

from hashlib_data import lorem

h = hashlib.sha1()
h.update(lorem.encode('utf-8'))
print(h.hexdigest())
```

Полученное значение дайджеста отличается от предыдущего, поскольку вместо алгоритма MD5 был использован алгоритм SHA1.

```
$ python3 hashlib_sha1.py

ea360b288b3dd178fe2625f55b2959bf1dba6eef
```

9.1.5. Создание хеш-кода с указанием имени алгоритма

Иногда удобнее сослаться на алгоритм, предоставив его имя в виде строки вместо непосредственного использования функции-конструктора. Это, например, полезно тем, что дает возможность сохранить тип используемого алгоритма в конфигурационном файле. В подобных случаях следует использовать функцию new() для создания объекта хеширования.

Листинг 9.5. hashlib_new.py

```
import argparse
import hashlib
import sys

from hashlib_data import lorem


parser = argparse.ArgumentParser('hashlib demo')
parser.add_argument(
    'hash_name',
    choices=hashlib.algorithms_available,
    help='the name of the hash algorithm to use',
)
parser.add_argument(
    'data',
    nargs='?',
    default=lorem,
    help='the input data to hash, defaults to lorem ipsum',
)
args = parser.parse_args()

h = hashlib.new(args.hash_name)
```

```
h.update(args.data.encode('utf-8'))
print(h.hexdigest())
```

Выполнение этой программы с использованием различных аргументов позволяет получить следующий вывод.

```
$ python3 hashlib_new.py sha1

ea360b288b3dd178fe2625f55b2959bf1dba6eef

$ python3 hashlib_new.py sha256
3c887cc71c67949df29568119cc646f46b9cd2c2b39d456065646bc2fc09ffd8

$ python3 hashlib_new.py sha512
a7e53384eb9bb4251a19571450465d51809e0b7046101b87c4faef96b9bc904cf
7f90035f444952dfd9f6084eeee2457433f3ade614712f42f80960b2fca43ff

$ python3 hashlib_new.py md5
3f2fd2c9e25d60fb0fa5d593b802b7a8
```

9.1.6. Инкрементное обновление

Метод `update()` объектов хеширования можно вызывать многократно. Каждый раз дайджест обновляется на основании очередной порции предоставленного текста. Инкрементное обновление более эффективно по сравнению с чтением всего файла в память и дает те же результаты.

Листинг 9.6. `hashlib_update.py`

```
import hashlib

from hashlib_data import lorem

h = hashlib.md5()
h.update(lorem.encode('utf-8'))
all_at_once = h.hexdigest()


def chunkize(size, text):
    "Return parts of the text in size-based increments."
    start = 0
    while start < len(text):
        chunk = text[start:start + size]
        yield chunk
        start += size
    return


h = hashlib.md5()
for chunk in chunkize(64, lorem.encode('utf-8')):
    h.update(chunk)
line_by_line = h.hexdigest()

print('All at once :', all_at_once)
```

```
print('Line by line:', line_by_line)
print('Same        :', (all_at_once == line_by_line))
```

Этот пример демонстрирует инкрементное обновление дайджеста по мере получения поступающих данных.

```
$ python3 hashlib_update.py
All at once : 3f2fd2c9e25d60fb0fa5d593b802b7a8
Line by line: 3f2fd2c9e25d60fb0fa5d593b802b7a8
Same : True
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `hashlib`[1].
- `hmac` (раздел 9.2). Модуль `hmac`.
- *OpenSSL*[2]. Набор инструментов шифрования с открытым исходным кодом.
- Пакет `cryptography`[3]. Пакет Python, предоставляющий готовые криптографические схемы и примитивы.
- Сайт Voidspace: *IronPython and hashlib*[4]. Оболочка для модуля `hashlib`, предназначенная для работы с IronPython.

9.2. hmac: криптографические цифровые подписи и верификация сообщений

Алгоритм HMAC можно использовать для проверки целостности данных, передаваемых между приложениями или сохраняемых в потенциально уязвимой среде. Основная идея заключается в генерировании криптографического хеш-кода для фактических данных в сочетании с использованием разделяемого секретного ключа. После этого результирующий хеш-код может быть использован для проверки переданных или сохраненных сообщений с целью определения степени их надежности без передачи секретного ключа.

Предупреждение

Отказ от ответственности. Я не являюсь специалистом в вопросах безопасности. Для получения более подробной информации относительно протокола HMAC обратитесь к документу **RFC 2104**[5].

9.2.1. Подписывание сообщений

Функция `new()` создает новый объект HMAC, вычисляющий цифровую подпись сообщения. В следующем примере используется заданный по умолчанию алгоритм MD5.

[1] `https://docs.python.org/3.5/library/hashlib.html`

[2] `www.openssl.org`

[3] `https://pypi.python.org/pypi/cryptography`

[4] `www.voidspace.org.uk/python/weblog/arch_d7_2006_10_07.shtml#e497`

[5] `https://tools.ietf.org/html/rfc2104.html`

Листинг 9.7. `hmac_simple.py`

```
import hmac

digest_maker = hmac.new(b'secret-shared-key-goes-here')

with open('lorem.txt', 'rb') as f:
    while True:
        block = f.read(1024)
        if not block:
            break
        digest_maker.update(block)

digest = digest_maker.hexdigest()
print(digest)
```

Этот код читает данные из файла и вычисляет для них цифровую подпись HMAC.

```
$ python3 hmac_simple.py

4bcb287e284f8c21e87e14ba2dc40b16
```

9.2.2. Альтернативные типы дайджестов

Несмотря на то что используемым по умолчанию алгоритмом хеширования для модуля hmac является MD5, эту опцию нельзя считать наиболее безопасной. Хеши MD5 имеют слабые стороны, такие как вероятность возникновения коллизий (когда хеш-коды для двух разных сообщений оказываются одинаковыми). Вместо алгоритма MD5 лучше использовать алгоритм SHA1, который признан более устойчивым.

Листинг 9.8. `hmac_sha.py`

```
import hmac
import hashlib

digest_maker = hmac.new(
    b'secret-shared-key-goes-here',
    b'',
    hashlib.sha1,
)

with open('hmac_sha.py', 'rb') as f:
    while True:
        block = f.read(1024)
        if not block:
            break
        digest_maker.update(block)

digest = digest_maker.hexdigest()
print(digest)
```

Метод new() получает три аргумента. Первый из них – это секретный ключ, который должен разделяться обеими конечными точками соединения, чтобы на обоих концах использовалось одно и то же значение. Второй аргумент – начальное сообщение. В случае таких сообщений небольшого размера, подлежащих аутентификации, как временная метка или HTTP-запрос POST, можно не использовать метод update() и передать методу new() все тело сообщения. Последний аргумент задает используемый алгоритм хеширования. По умолчанию он принимает значение hashlib.md5, но в данном примере он задан равным hashlib.sha1.

```
$ python3 hmac_sha.py

3c3992fa7aefb81b73a52f49713cf3faa272382a
```

9.2.3. Двоичные дайджесты

В предыдущих примерах для получения дайджестов, пригодных для вывода на печать, использовался метод hexdigest(), обеспечивающий удобное представление значения, вычисляемого методом digest(), которое является двоичным и может содержать непечатаемые символы, включая NUL. Некоторые веб-службы (в том числе сервис обработки онлайн-платежей Google Checkout и сервис файлового хостинга Amazon S3) вместо метода hexdigest() используют двоичные версии дайджестов, преобразованные с помощью схемы кодирования *base64*.

Листинг 9.9. hmac_base64.py

```
import base64
import hmac
import hashlib

with open('lorem.txt', 'rb') as f:
    body = f.read()

hash = hmac.new(
    b'secret-shared-key-goes-here',
    body,
    hashlib.sha1,
)

digest = hash.digest()
print(base64.encodestring(digest))
```

Строка, преобразованная с помощью схемы кодирования *base64*, заканчивается символом перевода строки, который во многих случаях должен отбрасываться при вставке строки в HTTP-заголовки или другие чувствительные к форматированию контексты.

```
$ python3 hmac_base64.py

b'olW2DoXHGJEKGU0aE9fOwSVE/o4=\n'
```

9.2.4. Применение цифровых подписей сообщений

HMAC-аутентификация должна использоваться при работе с любыми публичными сетевыми службами, а также при сохранении данных, когда соображения безопасности выходят на первый план. Например, данные, передаваемые с использованием канала или сокета, должны снабжаться цифровой подписью, подлежащей проверке перед использованием этих данных. Приведенный ниже расширенный пример содержится в файле `hmac_pickle.py`.

В первую очередь следует определить функцию, с помощью которой будет вычисляться дайджест для строки, а также простой класс, который будет инстанциализироваться и передаваться через канал передачи данных.

Листинг 9.10. `hmac_pickle.py`

```
import hashlib
import hmac
import io
import pickle
import pprint


def make_digest(message):
    "Вернуть дайджест сообщения."
    hash = hmac.new(
        b'secret-shared-key-goes-here',
        message,
        hashlib.sha1,
    )
    return hash.hexdigest().encode('utf-8')


class SimpleObject:
    """Продемонстрировать проверку дайджестов перед десериализацией.
    """

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
```

Затем создается буфер `BytesIO`, представляющий сокет или канал. В данном примере используется простейший формат потока данных, облегчающий выполнение синтаксического анализа. Вслед за дайджестом и длиной данных записывается символ перевода строки. Далее следует сериализованное представление объекта, сгенерированное с помощью модуля `pickle` (см. раздел 7.1). В реальной системе никакой зависимости от длины данных не должно быть — в конце концов, если дайджест неверен, то, вероятно, неверен и параметр длины. Более подходящим вариантом было бы использование определенной завершающей последовательности, появление которой в составе реальных данных маловероятно.

Далее программа записывает в поток два объекта. Первый объект записывается с использованием корректного значения дайджеста.

```
# Имитация записываемого сокета или потока с помощью буфера
out_s = io.BytesIO()

# Запись корректного объекта в поток
#  digest\nlength\npickle
o = SimpleObject('digest matches')
pickled_data = pickle.dumps(o)
digest = make_digest(pickled_data)
header = b'%s %d\n' % (digest, len(pickled_data))
print('WRITING: {}'.format(header))
out_s.write(header)
out_s.write(pickled_data)
```

Второй объект записывается с использованием некорректного дайджеста, полученного для совершенно других данных.

```
# Запись некорректного объекта в поток
o = SimpleObject('digest does not match')
pickled_data = pickle.dumps(o)
digest = make_digest(b'not the pickled data at all')
header = b'%s %d\n' % (digest, len(pickled_data))
print('\nWRITING: {}'.format(header))
out_s.write(header)
out_s.write(pickled_data)

out_s.flush()
```

Теперь, когда данные находятся в буфере `BytesIO`, их можно прочитать. Сначала читается строка с дайджестом и длиной данных. Затем читаются оставшиеся данные с использованием параметра длины. С помощью метода `pickle.load()` можно было бы выполнить чтение непосредственно из потока, но эта стратегия предполагает надежность данных, тогда как в нашем случае мы предполагаем обратное и потому не должны десериализовать данные преждевременно. Чтение сериализованного объекта в виде строки из потока без его фактической десериализации является более безопасным.

```
# Имитация читаемого сокета или канала
in_s = io.BytesIO(out_s.getvalue())

# Чтение данных
while True:
    first_line = in_s.readline()
    if not first_line:
        break
    incoming_digest, incoming_length = first_line.split(b' ')
    incoming_length = int(incoming_length.decode('utf-8'))
    print('\nREAD:', incoming_digest, incoming_length)
```

Как только сериализованные данные оказались в памяти, можно заново вычислить значение дайджеста и сравнить его с прочитанными данными, используя метод `compare_digest()`. Если дайджесты совпадают, то это означает, что данные безопасны и их можно десериализовать.

```
    incoming_pickled_data = in_s.read(incoming_length)

    actual_digest = make_digest(incoming_pickled_data)
    print('ACTUAL:', actual_digest)

    if hmac.compare_digest(actual_digest, incoming_digest):
        obj = pickle.loads(incoming_pickled_data)
        print('OK:', obj)
    else:
        print('WARNING: Data corruption')
```

Как можно видеть, первый объект успешно прошел верификацию, в то время как второй признан дефектным, чего и следовало ожидать.

```
$ python3 hmac_pickle.py

WRITING: b'f49cd2bf7922911129e8df37f76f95485a0b52ca 69\n'

WRITING: b'b01b209e28d7e053408ebe23b90fe5c33bc6a0ec 76\n'

READ: b'f49cd2bf7922911129e8df37f76f95485a0b52ca' 69
ACTUAL: b'f49cd2bf7922911129e8df37f76f95485a0b52ca'
OK: digest matches

READ: b'b01b209e28d7e053408ebe23b90fe5c33bc6a0ec' 76
ACTUAL: b'2ab061f9a9f749b8dd6f175bf57292e02e95c119'
WARNING: Data corruption
```

Сравнение двух дайджестов с использованием операторов сравнения простых строк или байтов может быть использовано в *атаках по времени* для раскрытия части или всего секретного ключа путем передачи дайджестов различной длины. Метод `compare_digest()` реализует быструю функцию с постоянным временем сравнения, обеспечивающую защиту от атак по времени.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `hmac`[6].
- **RFC 2104**[7]. *HMAC: Keyed-Hashing for Message Authentication.*
- `hashlib` (раздел 9.1). Модуль `hashlib` предоставляет генераторы хеш-значений для алгоритмов хеширования MD5 и SHA1.
- `pickle` (см. раздел 7.1). Библиотека сериализации.
- Википедия: *MD5*[8]. Описание алгоритма хеширования MD5.
- *Signing and Authenticating REST Requests* (Amazon AWS)[9]. Инструкции относительно процедуры аутентификации в облачном хранилище Amazon S3 с использованием подписанных цифровых удостоверений HMAC-SHA1.

[6] https://docs.python.org/3.5/library/hmac.html

[7] https://tools.ietf.org/html/rfc2104.html

[8] https://ru.wikipedia.org/wiki/MD5

[9] http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAuthentication.html

Глава 10

Параллельные вычисления: процессы, потоки и сопрограммы

Python включает эффективные инструменты, обеспечивающие управление параллельными вычислениями на основе процессов и потоков выполнения. Даже относительно простые программы можно заставить работать быстрее за счет организации параллельного (одновременного) выполнения отдельных частей задачи с помощью перечисленных ниже модулей.

Модуль `subprocess` (раздел 10.1) предоставляет API, обеспечивающий создание дополнительных процессов и обмен данными с ними. В частности, такая возможность полезна для программ, обрабатывающих текст, поскольку указанный API поддерживает двухсторонний обмен данными с новыми процессами через стандартные каналы ввода-вывода.

Модуль `signal` (раздел 10.2) позволяет использовать механизм сигналов Unix для передачи информации о событиях другим процессам. Сигналы обрабатываются асинхронно — обычно с прерыванием текущих операций, выполняемых программой в момент поступления сигнала. Обмен сигналами можно использовать лишь в качестве грубого прототипа системы обмена сообщениями, но существуют и другие, более надежные механизмы взаимодействия процессов, способные обеспечивать доставку более сложных сообщений.

Модуль `threading` (раздел 10.3) включает высокоуровневый, объектно-ориентированный API для управления одновременными вычислениями из Python. Объекты потоков (`Thread`) выполняются в рамках одного и того же процесса, разделяя общую память. Использование потоков — простейший способ масштабирования задач, в которых процессор в основном простаивает и бо́льшая часть времени тратится на обработку операций ввода-вывода.

Модуль `multiprocessing` (раздел 10.4) повторяет модуль `threading`, только вместо класса `Thread` предоставляется класс `Process`. Каждый объект `Process` — это полноценный системный процесс, имеющий собственное адресное пространство, но модуль `multiprocessing` обеспечивает возможности совместного использования данных и обмена сообщениями между процессами, так что во многих случаях переделка программы для перехода от использования потоков к использованию процессов сводится к изменению всего лишь нескольких инструкций импорта.

Модуль `asyncio` (раздел 10.5) предоставляет библиотеку, обеспечивающую распараллеливание вычислений и управление асинхронным вводом-выводом с использованием либо протоколов на основе классов, либо сопрограмм. Он заменяет устаревшие модули `asyncore` и `asynchat`, которые по-прежнему остаются доступными, но не рекомендованы к применению.

Модуль `concurrent.futures` (раздел 10.6) предоставляет реализацию исполняющих объектов на основе потоков и процессов, которая обеспечивает управление пулами ресурсов для выполнения параллельных задач.

10.1. subprocess: порождение дополнительных процессов

Модуль subprocess предоставляет три API для поддержки работы с процессами. Функция run(), добавленная в версию Python 3.5, — это высокоуровневый интерфейс для выполнения процесса и возможного захвата его вывода. Функции call(), check_call() и check_output() реализуют более ранний интерфейс, который перенесен из версии Python 2, но все еще поддерживается и широко применяется в существующих программах. Класс Popen — это низкоуровневый API, используемый для создания других программных интерфейсов и пригодный для организации более сложных видов взаимодействия процессов. Конструктору класса Popen передаются три аргумента, обеспечивающих обмен данными между новым процессом и его родителем посредством каналов. Класс Popen предоставляет всю функциональность модулей и функций, которые он заменяет, дополняя ее многими другими возможностями. В его API встроено множество дополнительных операций (таких, например, как закрытие ненужных файловых дескрипторов и каналов), что избавляет от необходимости выполнять их вручную из кода приложения.

Модуль subprocess призван заменить такие функции, как os.system(), os.spawnv(), различные версии popen(), входящие в состав модулей os (раздел 17.3) и popen2, а также модуль commands. Чтобы упростить сравнение модуля subprocess с указанными модулями, многие из приведенных в этом разделе примеров воспроизводят аналогичные примеры, в которых использовались модули os и popen2.

Примечание

Программные интерфейсы для работы с процессами под управлением Unix и Windows в основном совпадают, отличаясь лишь внутренней реализацией из-за различий в моделях процесса в этих операционных системах. Все представленные ниже примеры тестировались на платформе Mac OS X. На платформах, отличных от Unix, они могут вести себя иначе.

10.1.1. Выполнение внешних команд

Чтобы выполнить внешнюю команду без взаимодействия с ней, как это делается с помощью функции os.system(), используйте функцию run().

Листинг 10.1. subprocess_os_system.py

```
import subprocess

completed = subprocess.run(['ls', '-1'])
print('returncode:', completed.returncode)
```

Аргументы командной строки передаются в виде списка строк, благодаря чему отпадает необходимость экранировать кавычки и другие специальные символы, которые могут интерпретироваться командной оболочкой. Функция run() возвращает экземпляр CompletedProcess, содержащий связанную с процессом информацию, такую как код завершения и вывод.

```
$ python3 subprocess_os_system.py

index.rst
interaction.py
repeater.py
signal_child.py
signal_parent.py
subprocess_check_output_error_trap_output.py
subprocess_os_system.py
subprocess_pipes.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
subprocess_popen_read.py
subprocess_popen_write.py
subprocess_run_check.py
subprocess_run_output.py
subprocess_run_output_error.py
subprocess_run_output_error_suppress.py
subprocess_run_output_error_trap.py
subprocess_shell_variables.py
subprocess_signal_parent_shell.py
subprocess_signal_setpgrp.py
returncode: 0
```

Установка значения `True` для аргумента `shell` функции `run()` порождает вспомогательный процесс оболочки, в котором затем и выполняется команда. По умолчанию используется непосредственное выполнение команды.

Листинг 10.2. subprocess_shell_variables.py

```
import subprocess

completed = subprocess.run('echo $HOME', shell=True)
print('returncode:', completed.returncode)
```

Использование промежуточной оболочки означает, что перед выполнением команды будут обработаны шаблоны модуля `glob` и другие специальные символы.

```
$ python3 subprocess_shell_variables.py

/Users/dhellmann
returncode: 0
```

Примечание

Вызов функции `run()` без передачи аргумента `check=True` эквивалентен использованию функции `call()`, которая возвращает лишь код завершения процесса.

10.1.1.1. Обработка ошибок

Атрибут `returncode` экземпляра `CompletedProcess` является кодом завершения программы. Ответственность за его интерпретацию с целью обнаружения

ошибок возлагается на вызывающий код. Если для аргумента check функции run() установлено значение True, то выполняется проверка кода завершения. Если код завершения указывает на возникновение ошибки, то возбуждается исключение CalledProcessError.

Листинг 10.3. subprocess_run_check.py

```
import subprocess

try:
    subprocess.run(['false'], check=True)
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
```

Команда false всегда завершается с ненулевым кодом состояния, который интерпретируется функцией run() как ошибка.

```
$ python3 subprocess_run_check.py

ERROR: Command '['false']' returned non-zero exit status 1
```

Примечание

Вызов функции run() с передачей ей аргумента check=True эквивалентен использованию функции check_call().

10.1.1.2. Захват вывода

Каналы стандартного ввода-вывода процесса, запущенного с помощью функции run(), привязаны к каналам родительского процесса. Как следствие, перехват вывода команды вызывающей программой невозможен. Передача значения PIPE в качестве именованного аргумента stdout или stderr позволяет перехватить вывод для последующей обработки.

Листинг 10.4. subprocess_run_output.py

```
import subprocess

completed = subprocess.run(
    ['ls', '-1'],
    stdout=subprocess.PIPE,
)
print('returncode:', completed.returncode)
print('Have {} bytes in stdout:\n{}'.format(
    len(completed.stdout),
    completed.stdout.decode('utf-8'))
)
```

Команда ls -1 успешно выполняется, поэтому текст, который она выводит в канал стандартного вывода, перехватывается и выводится на печать.

```
$ python3 subprocess_run_output.py

returncode: 0
```

```
Have 522 bytes in stdout:

index.rst
interaction.py
repeater.py
signal_child.py
signal_parent.py
subprocess_check_output_error_trap_output.py
subprocess_os_system.py
subprocess_pipes.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
subprocess_popen_read.py
subprocess_popen_write.py
subprocess_run_check.py
subprocess_run_output.py
subprocess_run_output_error.py
subprocess_run_output_error_suppress.py
subprocess_run_output_error_trap.py
subprocess_shell_variables.py
subprocess_signal_parent_shell.py
subprocess_signal_setpgrp.py
```

Примечание

Вызов функции `run()` с передачей ей аргументов `check=True` и `stdout=subprocess.PIPE` эквивалентен использованию функции `check_output()`.

В следующем примере предпринимается попытка выполнения серии команд в производной оболочке. Сообщения направляются в стандартный поток вывода и стандартный поток ошибок до тех пор, пока при выполнении одной из команд не возникнет ошибка.

Листинг 10.5. `subprocess_run_output_error.py`

```
import subprocess

try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        check=True,
        shell=True,
        stdout=subprocess.PIPE,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('Have {} bytes in stdout: {!r}'.format(
        len(completed.stdout),
        completed.stdout.decode('utf-8'))
    )
```

Сообщения, направляемые в стандартный поток ошибок, выводятся на консоль, тогда как сообщения, направляемые в стандартный поток вывода, скрываются.

```
$ python3 subprocess_run_output_error.py

to stderr
ERROR: Command 'echo to stdout; echo to stderr 1>&2; exit 1'
returned non-zero exit status 1
```

Чтобы предотвратить вывод на консоль сообщений об ошибках при выполнении команд, запущенных с помощью функции `run()`, следует установить для аргумента `stderr` значение `PIPE`.

Листинг 10.6. subprocess_run_output_error_trap.py

```
import subprocess

try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('Have {} bytes in stdout: {!r}'.format(
        len(completed.stdout),
        completed.stdout.decode('utf-8'))
    )
    print('Have {} bytes in stderr: {!r}'.format(
        len(completed.stderr),
        completed.stderr.decode('utf-8'))
    )
```

В данном примере аргумент `check=True` не передается, поэтому выходная информация команды перехватывается и выводится на консоль.

```
$ python3 subprocess_run_output_error_trap.py

returncode: 1
Have 10 bytes in stdout: 'to stdout\n'
Have 10 bytes in stderr: 'to stderr\n'
```

Чтобы перехватывать сообщения об ошибках при использовании функции `check_output()`, следует установить для аргумента `stderr` значение `STDOUT`. В этом случае сообщения об ошибках будут выводиться вместе с остальными результатами выполнения команды.

Листинг 10.7. subprocess_check_output_error_trap_output.py

```
import subprocess

try:
    output = subprocess.check_output(
        'echo to stdout; echo to stderr 1>&2',
        shell=True,
        stderr=subprocess.STDOUT,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('Have {} bytes in output: {!r}'.format(
        len(output),
        output.decode('utf-8'))
    )
```

Порядок вывода может меняться в зависимости от использования буферизации стандартного потока вывода и объема выводимых данных.

```
$ python3 subprocess_check_output_error_trap_output.py

Have 20 bytes in output: 'to stdout\nto stderr\n'
```

10.1.1.3. Подавление вывода

В тех случаях, когда вывод не должен отображаться или перехватываться, его можно подавить, установив для соответствующего аргумента значение DEVNULL. В следующем примере подавляются стандартные потоки вывода и ошибок.

Листинг 10.8. subprocess_run_output_error_suppress.py

```
import subprocess

try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        shell=True,
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('stdout is {!r}'.format(completed.stdout))
    print('stderr is {!r}'.format(completed.stderr))
```

Имя константы DEVNULL происходит от названия специального файла устройства Unix (*нулевого* устройства) — /dev/null. Это устройство возвращает признак конца файла при его открытии для чтения и получает, но игнорирует, любые объемы данных при записи.

```
$ python3 subprocess_run_output_error_suppress.py

returncode: 1
stdout is None
stderr is None
```

10.1.2. Непосредственная работа с каналами

Функции `run()`, `call()`, `check_call()` и `check_output()` являются обертками вокруг класса `Popen`. Непосредственное использование класса `Popen` предоставляет больше возможностей контроля над выполнением команд и обработкой потоков ввода-вывода. Например, варьируя значения параметров `stdin`, `stdout` и `stderr`, можно имитировать различные режимы вызова функции `os.popen()`.

10.1.2.1. Одностороннее взаимодействие с процессом

Чтобы запустить процесс и прочитать весь его вывод, следует установить значение `PIPE` для аргумента `stdout` и вызвать метод `communicate()`.

Листинг 10.9. subprocess_popen_read.py

```
import subprocess

print('read:')
proc = subprocess.Popen(
    ['echo', '"to stdout"'],
    stdout=subprocess.PIPE,
)
stdout_value = proc.communicate()[0].decode('utf-8')
print('stdout:', repr(stdout_value))
```

Это аналогично тому, как работает функция `popen()`, за исключением того факта, что управление чтением осуществляется экземпляром `Popen`.

```
$ python3 subprocess_popen_read.py

read:
stdout: '"to stdout"\n'
```

Чтобы настроить канал для записи в него данных вызывающей программой, следует установить для аргумента `stdin` значение `PIPE`.

Листинг 10.10. subprocess_popen_write.py

```
import subprocess

print('write:')
proc = subprocess.Popen(
    ['cat', '-'],
    stdin=subprocess.PIPE,
)
proc.communicate('stdin: to stdin\n'.encode('utf-8'))
```

Чтобы однократно направить данные в канал стандартного ввода процесса, их следует передать методу `communicate()`. Этот подход аналогичен использованию функции `popen()` в режиме `'w'`.

```
$ python3 -u subprocess_popen_write.py

write:
stdin: to stdin
```

10.1.2.2. Двухстороннее взаимодействие с процессом

Чтобы настроить экземпляр `Popen` как для чтения, так и для записи, следует использовать комбинацию предыдущих подходов.

Листинг 10.11. subprocess_popen2.py

```
import subprocess

print('popen2:')

proc = subprocess.Popen(
    ['cat', '-'],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
msg = 'through stdin to stdout'.encode('utf-8')
stdout_value = proc.communicate(msg)[0].decode('utf-8')
print('pass through:', repr(stdout_value))
```

Настроенный таким образом канал имитирует вызов функции `popen2()`.

```
$ python3 -u subprocess_popen2.py

popen2:
pass through: 'through stdin to stdout'
```

10.1.2.3. Перехват вывода сообщений об ошибках

Также возможно наблюдать одновременно за двумя потоками, `stdout` и `stderr`, как это делается с помощью функции `popen3()`.

Листинг 10.12. subprocess_popen3.py

```
import subprocess

print('popen3:')
proc = subprocess.Popen(
    'cat -; echo "to stderr" 1>&2',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
)
```

```
msg = 'through stdin to stdout'.encode('utf-8')
stdout_value, stderr_value = proc.communicate(msg)
print('pass through:', repr(stdout_value.decode('utf-8')))
print('stderr      :', repr(stderr_value.decode('utf-8')))
```

Чтение из потока `stderr` работает так же, как и чтение из потока `stdout`. Передача значения `PIPE` сообщает экземпляру `Popen` о необходимости подключения к каналу, а метод `communicate()` читает все данные из него, прежде чем выполнить возврат.

```
$ python3 -u subprocess_popen3.py

popen3:
pass through: 'through stdin to stdout'
stderr      : 'to stderr\n'
```

10.1.2.4. Объединение каналов стандартного вывода и ошибок

Чтобы направить вывод сообщений об ошибках из процесса в канал его стандартного вывода, следует использовать в качестве значения аргумента `stderr` константу `STDOUT` вместо `PIPE`.

Листинг 10.13. `subprocess_popen4.py`

```
import subprocess

print('popen4:')
proc = subprocess.Popen(
    'cat -; echo "to stderr" 1>&2',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
)
msg = 'through stdin to stdout\n'.encode('utf-8')
stdout_value, stderr_value = proc.communicate(msg)
print('combined output:', repr(stdout_value.decode('utf-8')))
print('stderr value   :', repr(stderr_value))
```

Подобное объединение каналов вывода имитирует работу функции `popen4()`.

```
$ python3 -u subprocess_popen4.py

popen4:
combined output: 'through stdin to stdout\nto stderr\n'
stderr value   : None
```

10.1.3. Соединение сегментов канала

Несколько команд можно соединить, чтобы они образовали *конвейер* в стиле оболочки Unix, создав отдельные экземпляры `Popen` и объединив в одну цепочку их каналы ввода и вывода. Атрибут `stdout` одного экземпляра `Popen` используется

вместо константы PIPE в качестве аргумента stdin следующего экземпляра в конвейере. Вывод читается из дескриптора stdout последней команды конвейера.

Листинг 10.14. subprocess_pipes.py

```
import subprocess

cat = subprocess.Popen(
    ['cat', 'index.rst'],
    stdout=subprocess.PIPE,
)

grep = subprocess.Popen(
    ['grep', '.. literalinclude::'],
    stdin=cat.stdout,
    stdout=subprocess.PIPE,
)

cut = subprocess.Popen(
    ['cut', '-f', '3', '-d:'],
    stdin=grep.stdout,
    stdout=subprocess.PIPE,
)

end_of_pipe = cut.stdout

print('Included files:')
for line in end_of_pipe:
    print(line.decode('utf-8').strip())
```

Этот пример воспроизводит выполнение следующей команды:

```
$ cat index.rst | grep ".. literalinclude" | cut -f 3 -d:
```

Конвейер читает файл в формате reSstructuredText (*index.rst*) для данного раздела и находит в нем все строки, включающие другие файлы. После этого он выводит имена всех включаемых файлов.

```
$ python3 -u subprocess_pipes.py

Included files:
subprocess_os_system.py
subprocess_shell_variables.py
subprocess_run_check.py
subprocess_run_output.py
subprocess_run_output_error.py
subprocess_run_output_error_trap.py
subprocess_check_output_error_trap_output.py
subprocess_run_output_error_suppress.py
subprocess_popen_read.py
subprocess_popen_write.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
```

```
subprocess_pipes.py
repeater.py
interaction.py
signal_child.py
signal_parent.py
subprocess_signal_parent_shell.py
subprocess_signal_setpgrp.py
```

10.1.4. Взаимодействие с другой командой

Во всех предыдущих примерах предполагалась ограниченная степень взаимодействия. Метод `communicate()` читает весь вывод и ожидает завершения дочернего процесса, прежде чем вернуть управление. Помимо этого существует возможность читать и записывать информацию с помощью отдельных дескрипторов каналов, используемых экземпляром `Popen` для инкрементного обмена данными по мере выполнения программы.

В следующем примере в качестве дочернего процесса используется сценарий `repeater.py`. Этот сценарий читает данные из канала `stdin` и записывает в канал `stdout` по одной строке за раз до тех пор, пока не перестанет получать входные данные. Он также записывает в канал `stderr` сообщения о начале и завершении выполнения, что позволяет судить о времени жизни дочернего процесса.

Листинг 10.15. `repeater.py`

```
import sys

sys.stderr.write('repeater.py: starting\n')
sys.stderr.flush()

while True:
    next_line = sys.stdin.readline()
    sys.stderr.flush()
    if not next_line:
        break
    sys.stdout.write(next_line)
    sys.stdout.flush()

sys.stderr.write('repeater.py: exiting\n')
sys.stderr.flush()
```

В приведенном ниже сценарии дескрипторы файлов `stdin` и `stdout`, принадлежащие экземпляру `Popen`, используются двумя способами. Сначала последовательность, включающая пять чисел, записывается в канал `stdin` процесса; после каждой операции записи выводимые данные считываются обратно. Далее записываются те же пять чисел, но весь вывод читается за один раз с помощью метода `communicate()`.

Листинг 10.16. `interaction.py`

```
import io
import subprocess
```

```
print('One line at a time:')
proc = subprocess.Popen(
    'python3 repeater.py',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
stdin = io.TextIOWrapper(
    proc.stdin,
    encoding='utf-8',
    line_buffering=True,  # отправка данных при получении
                          # символа перевода строки
)
stdout = io.TextIOWrapper(
    proc.stdout,
    encoding='utf-8',
)
for i in range(5):
    line = '{}\n'.format(i)
    stdin.write(line)
    output = stdout.readline()
    print(output.rstrip())
remainder = proc.communicate()[0].decode('utf-8')
print(remainder)

print()
print('All output at once:')
proc = subprocess.Popen(
    'python3 repeater.py',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
stdin = io.TextIOWrapper(
    proc.stdin,
    encoding='utf-8',
)
for i in range(5):
    line = '{}\n'.format(i)
    stdin.write(line)
stdin.flush()

output = proc.communicate()[0].decode('utf-8')
print(output)
```

Строки "repeater.py: exiting" появляются для каждого цикла в разных позициях.

```
$ python3 -u interaction.py

One line at a time:
repeater.py: starting
0
1
```

```
2
3
4
repeater.py: exiting


All output at once:
repeater.py: starting
repeater.py: exiting
0
1
2
3
4
```

10.1.5. Межпроцессный обмен сигналами

Примеры управления процессами для модуля os (раздел 17.3) включают демонстрацию передачи сигналов между процессами с помощью функций os.fork() и os.kill(). Поскольку каждый экземпляр Popen предоставляет атрибут pid, содержащий идентификатор ID дочернего процесса, нечто аналогичное можно сделать и с помощью модуля subprocess.

Следующий пример объединяет два сценария. Дочерний процесс устанавливает дескриптор для сигнала USR.

Листинг 10.17. signal_child.py

```
import os
import signal
import time
import sys

pid = os.getpid()
received = False


def signal_usr1(signum, frame):
    "Обратный вызов, запускаемый при получении сигнала."
    global received
    received = True
    print('CHILD {:>6}: Received USR1'.format(pid))
    sys.stdout.flush()


print('CHILD {:>6}: Setting up signal handler'.format(pid))
sys.stdout.flush()
signal.signal(signal.SIGUSR1, signal_usr1)
print('CHILD {:>6}: Pausing to wait for signal'.format(pid))
sys.stdout.flush()
time.sleep(3)

if not received:
    print('CHILD {:>6}: Never received signal'.format(pid))
```

Второй сценарий запускается в качестве родительского процесса. Он вызывает сценарий `signal_child.py`, а затем посылает сигнал `USR1`.

Листинг 10.18. `signal_parent.py`

```
import os
import signal
import subprocess
import time
import sys

proc = subprocess.Popen(['python3', 'signal_child.py'])
print('PARENT      : Pausing before sending signal...')
sys.stdout.flush()
time.sleep(1)
print('PARENT      : Signaling child')
sys.stdout.flush()
os.kill(proc.pid, signal.SIGUSR1)
```

Результаты представлены ниже.

```
$ python3 signal_parent.py

PARENT      : Pausing before sending signal...
CHILD 26976: Setting up signal handler
CHILD 26976: Pausing to wait for signal
PARENT      : Signaling child
CHILD 26976: Received USR1
```

10.1.5.1. Группы/сеансы процессов

Если процесс, созданный экземпляром `Popen`, порождает подпроцессы, то эти дочерние процессы не будут получать сигналы, посланные их родителю. Как следствие, если использовать аргумент `shell` в конструкторе `Popen`, то будет нелегко прекратить выполнение команды, запущенной в оболочке, путем отправки сигнала `SIGINT` или `SIGTERM`.

Листинг 10.19. `subprocess_signal_parent_shell.py`

```
import os
import signal
import subprocess
import tempfile
import time
import sys

script = '''#!/bin/sh
echo "Shell script in process $$"
set -x
python3 signal_child.py
'''
script_file = tempfile.NamedTemporaryFile('wt')
script_file.write(script)
script_file.flush()
```

```
proc = subprocess.Popen(['sh', script_file.name])
print('PARENT      : Pausing before signaling {}...'.format(
    proc.pid))
sys.stdout.flush()
time.sleep(1)
print('PARENT      : Signaling child {}'.format(proc.pid))
sys.stdout.flush()
os.kill(proc.pid, signal.SIGUSR1)
time.sleep(3)
```

В этом примере атрибут `pid`, используемый для отправки сигнала, не совпадает с `pid` дочернего процесса сценария оболочки, ожидающего получения сигнала, поскольку в данном случае взаимодействуют три процесса:

- программа `subprocess_signal_parent_shell.py`;
- процесс оболочки, который выполняет сценарий, созданный основной программой на Python;
- программа `signal_child.py`.

```
$ python3 subprocess_signal_parent_shell.py

PARENT      : Pausing before signaling 26984...
Shell script in process 26984
+ python3 signal_child.py
CHILD 26985: Setting up signal handler
CHILD 26985: Pausing to wait for signal
PARENT      : Signaling child 26984
CHILD 26985: Never received signal
```

Чтобы обеспечить возможность отправки сигналов потомкам, не зная идентификаторы их процессов, следует использовать *группу процессов*, ассоциирующую дочерние процессы, что позволяет доставлять сигналы сразу всей группе. Группа процессов создается при помощи функции `os.setpgrp()`, которая устанавливает для идентификатора группы значение идентификатора текущего процесса. Все дочерние процессы наследуют членство в группе от своего родителя. Поскольку эта группа должна быть установлена только в оболочке, создаваемой объектом `Popen` и его наследниками, то функция `os.setpgrp()` не должна вызываться в том же процессе, в котором создается объект `Popen`. Вместо этого данная функция передается конструктору `Popen` в качестве аргумента `preexec_fn`, так что она выполняется после вызова `fork()` в новом процессе до того, как использует функцию `exec()` для выполнения оболочки. Отправка сигнала всей группе процессов осуществляется при помощи функции `os.killpg()` со значением `pid` из экземпляра `Popen`.

Листинг 10.20. `subprocess_signal_setpgrp.py`

```
import os
import signal
import subprocess
import tempfile
import time
import sys
```

```
def show_setting_prgrp():
    print('Calling os.setpgrp() from {}'.format(os.getpid()))
    os.setpgrp()
    print('Process group is now {}'.format(
        os.getpid(), os.getpgrp()))
    sys.stdout.flush()


script = '''#!/bin/sh
echo "Shell script in process $$"
set -x
python3 signal_child.py
'''
script_file = tempfile.NamedTemporaryFile('wt')
script_file.write(script)
script_file.flush()

proc = subprocess.Popen(
    ['sh', script_file.name],
    preexec_fn=show_setting_prgrp,
)
print('PARENT      : Pausing before signaling {}...'.format(
    proc.pid))
sys.stdout.flush()
time.sleep(1)
print('PARENT      : Signaling process group {}'.format(
    proc.pid))
sys.stdout.flush()
os.killpg(proc.pid, signal.SIGUSR1)
time.sleep(3)
```

Соответствующая последовательность событий описана ниже.

1. Родительская программа создает экземпляр класса `Popen`.
2. Экземпляр `Popen` порождает новый процесс.
3. Новый процесс выполняет функцию `os.setpgrp()`.
4. Новый процесс выполняет функцию `exec()` для запуска командной оболочки.
5. Оболочка выполняет сценарий.
6. Сценарий оболочки вновь порождает процесс, и этот процесс выполняет код Python.
7. Код Python выполняет сценарий `signal_child.py`.
8. Родительская программа посылает сигнал группе процессов, используя идентификатор `pid` оболочки.
9. Процессы оболочки и Python получают сигнал.
10. Оболочка игнорирует сигнал.
11. Процесс Python, выполняющий сценарий `signal_child.py`, вызывает обработчик сигналов.

```
$ python3 subprocess_signal_setpgrp.py

Calling os.setpgrp() from 26992
Process group is now 26992
PARENT      : Pausing before signaling 26992...
Shell script in process 26992
+ python3 signal_child.py
CHILD 26993: Setting up signal handler
CHILD 26993: Pausing to wait for signal
PARENT      : Signaling process group 26992
CHILD 26993: Received USR1
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `subprocess`[1].
- `os` (раздел 17.3). Несмотря на то что функции модуля `subprocess` заменяют многие функции модуля `os`, предназначенные для работы с процессами, последние все еще широко применяются в существующем коде.
- *UNIX Signals and Process Groups*[2]. Неплохое описание работы с сигналами Unix и группами процессов.
- `signal` (раздел 10.2). Более подробное описание модуля `signal`.
- *Advanced Programming in the UNIX Environment, Third Edition*[3]. Обсуждение таких аспектов работы с несколькими процессами, как обмен сигналами и закрытие дубликатов файловых дескрипторов.
- `pipes`. Шаблоны конвейеров оболочки Unix в стандартной библиотеке.

10.2. `signal`: асинхронные системные события

Сигналы — это средство операционной системы, обеспечивающее доставку уведомлений о наступлении каких-либо событий и их асинхронную обработку. Сигналы могут генерироваться самой системой или посылаться одним процессом другому. Поскольку сигналы прерывают обычный ход выполнения программы, некоторые операции (особенно ввода-вывода) могут приводить к ошибкам, если сигнал был получен во время их выполнения.

Сигналы идентифицируются целыми числами и определяются в заголовочных файлах C операционной системы. Python предоставляет сигналы, соответствующие конкретной платформе, в виде символов в модуле `signal`. В примерах, приведенных в этом разделе, используются сигналы `SIGINT` и `SIGUSR1`, которые определены почти во всех Unix-подобных системах.

Примечание

Программирование с использованием обработчиков сигналов Unix — задача непростая. Данный раздел служит лишь введением в эту сложную тему и не охватывает все детали,

[1] https://docs.python.org/3.5/library/subprocess.html

[2] www.cs.ucsb.edu/~almeroth/classes/W99.276/assignment1/signals.html

[3] https://www.amazon.com/Advanced-Programming-UNIX-Environment-3rd/dp/0321637739/

знание которых необходимо для успешного использования сигналов на всех платформах. В различных версиях Unix соблюдается определенная степень стандартизации, однако возможны некоторые расхождения. Для получения более подробной информации по этому вопросу следует обратиться к документации операционной системы.

10.2.1. Получение сигналов

Как и в случае других подходов, основанных на обработке событий, для получения сигналов используют функции обратного вызова — *обработчики сигналов*, которые вызываются при получении сигнала. Аргументами обработчиков служат номера сигналов и кадры стека из той точки программы, в которой сигнал прервал ее выполнение.

Листинг 10.21. `signal_signal.py`

```
import signal
import os
import time


def receive_signal(signum, stack):
    print('Received:', signum)


# Регистрация дескрипторов сигналов
signal.signal(signal.SIGUSR1, receive_signal)
signal.signal(signal.SIGUSR2, receive_signal)

# Вывод ID процесса, который впоследствии можно будет
# использовать с командой kill для отправки сигналов этой программе
print('My PID is:', os.getpid())

while True:
    print('Waiting...')
    time.sleep(3)
```

В этом примере сценарий выполняет бесконечный цикл, делая паузы длительностью в несколько секунд на каждой итерации. При получении сигнала вызов `sleep()` прерывается и обработчик `receive_signal` выводит на консоль номер сигнала. Выполнение цикла продолжается, когда обработчик сигнала возвращает управление.

Для отправки сигналов выполняющейся программе можно использовать вызов `os.kill()` или программу `kill` командной строки Unix.

```
$ python3 signal_signal.py

My PID is: 71387
Waiting...
Waiting...
Waiting...
Received: 30
Waiting...
```

```
Waiting...
Received: 31
Waiting...
Waiting...
Traceback (most recent call last):
  File "signal_signal.py", line 28, in <module>
    time.sleep(3)
KeyboardInterrupt
```

Предыдущий вывод был получен путем запуска сценария `signal_signal.py` в одном окне и выполнения приведенных ниже команд в другом.

```
$ kill -USR1 71387
$ kill -USR2 71387
$ kill -INT 71387
```

10.2.2. Получение информации о зарегистрированных обработчиках сигналов

Чтобы увидеть, какие обработчики зарегистрированы для сигнала, используйте функцию `getsignal()`. В качестве аргумента ей передается номер сигнала. Возвращаемым значением является зарегистрированный обработчик или одно из специальных значений: `SIG_IGN` (если данный сигнал игнорируется), `SIG_DFL` (если используется поведение, заданное по умолчанию) или `None` (если существующий обработчик сигналов зарегистрирован из C, а не из Python).

Листинг 10.22. `signal_getsignal.py`

```
import signal


def alarm_received(n, stack):
    return


signal.signal(signal.SIGALRM, alarm_received)

signals_to_names = {
    getattr(signal, n): n
    for n in dir(signal)
    if n.startswith('SIG') and '_' not in n
}

for s, name in sorted(signals_to_names.items()):
    handler = signal.getsignal(s)
    if handler is signal.SIG_DFL:
        handler = 'SIG_DFL'
    elif handler is signal.SIG_IGN:
        handler = 'SIG_IGN'
    print('{:<10} ({:2d}):'.format(name, s), handler)
```

Опять-таки, поскольку в разных ОС сигналы могут быть определены по-разному, вывод может меняться от системы к системе. Следующий вывод получен в OS X.

```
$ python3 signal_getsignal.py

SIGHUP    ( 1): SIG_DFL
SIGINT    ( 2): <built-in function default_int_handler>
SIGQUIT   ( 3): SIG_DFL
SIGILL    ( 4): SIG_DFL
SIGTRAP   ( 5): SIG_DFL
SIGIOT    ( 6): SIG_DFL
SIGEMT    ( 7): SIG_DFL
SIGFPE    ( 8): SIG_DFL
SIGKILL   ( 9): None
SIGBUS    (10): SIG_DFL
SIGSEGV   (11): SIG_DFL
SIGSYS    (12): SIG_DFL
SIGPIPE   (13): SIG_IGN
SIGALRM   (14): <function alarm_received at 0x100757f28>
SIGTERM   (15): SIG_DFL
SIGURG    (16): SIG_DFL
SIGSTOP   (17): None
SIGTSTP   (18): SIG_DFL
SIGCONT   (19): SIG_DFL
SIGCHLD   (20): SIG_DFL
SIGTTIN   (21): SIG_DFL
SIGTTOU   (22): SIG_DFL
SIGIO     (23): SIG_DFL
SIGXCPU   (24): SIG_DFL
SIGXFSZ   (25): SIG_IGN
SIGVTALRM (26): SIG_DFL
SIGPROF   (27): SIG_DFL
SIGWINCH  (28): SIG_DFL
SIGINFO   (29): SIG_DFL
SIGUSR1   (30): SIG_DFL
SIGUSR2   (31): SIG_DFL
```

10.2.3. Отправка сигналов

В Python для отправки сигналов предназначена функция `os.kill()`. Пример ее использования приведен в разделе 17.3.10.

10.2.4. Сигналы таймера

Сигналы таймера — особый вид сигналов, которые генерируются, если программа направляет ОС запрос на передачу ей уведомления по истечении определенного промежутка времени. Сигналы этого типа удобно использовать для предотвращения бесконечных блокировок при выполнении операций ввода-вывода и других системных вызовов.

Листинг 10.23. signal_alarm.py

```
import signal
import time


def receive_alarm(signum, stack):
    print('Alarm :', time.ctime())


# Вызвать receive_alarm через 2 секунды
signal.signal(signal.SIGALRM, receive_alarm)
signal.alarm(2)

print('Before:', time.ctime())
time.sleep(4)
print('After :', time.ctime())
```

В этом примере выполнение вызова `sleep()` прерывается сигналом, но восстанавливается после его обработки. Как следует из сообщения, которое выводится после возврата из функции `sleep()`, выполнение программы было приостановлено на время, по крайней мере равное установленной длительности паузы.

```
$ python3 signal_alarm.py

Before: Sun Sep 11 11:31:18 2016
Alarm : Sun Sep 11 11:31:20 2016
After : Sun Sep 11 11:31:22 2016
```

10.2.5. Игнорирование сигналов

Чтобы игнорировать сигнал, следует зарегистрировать `SIG_IGN` в качестве обработчика. В следующем сценарии обработчик по умолчанию для сигнала `SIGINT` заменяется на `SIG_IGN` и устанавливается обработчик для сигнала `SIGUSR1`. После этого вызов `signal.pause()` переводит программу в состояние ожидания до получения сигнала.

Листинг 10.24. signal_ignore.py

```
import signal
import os
import time


def do_exit(sig, stack):
    raise SystemExit('Exiting')


signal.signal(signal.SIGINT, signal.SIG_IGN)
signal.signal(signal.SIGUSR1, do_exit)

print('My PID:', os.getpid())

signal.pause()
```

Обычно сигнал SIGINT (сигнал, посылаемый программе командной оболочкой при нажатии пользователем комбинации клавиш <Ctrl+C) возбуждает исключение KeyboardInterrupt. В этом примере сигнал SIGINT игнорируется, а при получении сигнала SIGUSR1 возбуждается исключение SystemExit. Каждое появление символов ^C в выводе означает, что пользователь предпринимал попытку прервать выполнение сценария нажатием комбинации клавиш <Ctrl+C> на консоли. Чтобы завершить выполнение сценария, следует выполнить на другом терминале команду kill -USR1 72598.

```
$ python3 signal_ignore.py

My PID: 72598
^C^C^C^CExiting
```

10.2.6. Сигналы и потоки

Совместное использование сигналов и потоков редко работает хорошо, поскольку получать сигналы будет только основной поток. Следующий сценарий устанавливает обработчик сигналов, ожидает получения сигналов в одном потоке и посылает сигнал из другого потока.

Листинг 10.25. signal_threads.py

```
import signal
import threading
import os
import time


def signal_handler(num, stack):
    print('Received signal {} in {}'.format(
        num, threading.currentThread().name))


signal.signal(signal.SIGUSR1, signal_handler)


def wait_for_signal():
    print('Waiting for signal in',
          threading.currentThread().name)
    signal.pause()
    print('Done waiting')


# Запустить поток, который не будет получать сигналы
receiver = threading.Thread(
    target=wait_for_signal,
    name='receiver',
)
receiver.start()
time.sleep(0.1)
```

```
def send_signal():
    print('Sending signal in', threading.currentThread().name)
    os.kill(os.getpid(), signal.SIGUSR1)


sender = threading.Thread(target=send_signal, name='sender')
sender.start()
sender.join()

# Ожидать появления сигнала (этого никогда не произойдет!)
print('Waiting for', receiver.name)
signal.alarm(2)
receiver.join()
```

Здесь все обработчики сигналов были зарегистрированы в основном потоке, как того требует реализация модуля `signal` в Python, независимо от обеспечиваемой базовой платформой поддержки сочетания потоков и сигналов. Несмотря на то что поток-получатель выполняет вызов `signal.pause()`, в действительности он не получает сигнал. Вызов `signal.alarm(2)` в конце сценария предотвращает бесконечную блокировку, обусловленную тем, что поток-получатель никогда не вернет управление.

```
$ python3 signal_threads.py

Waiting for signal in receiver
Sending signal in sender
Received signal 30 in MainThread
Waiting for receiver
Alarm clock
```

Несмотря на то что сигналы таймера можно установить в любом потоке, их всегда получает основной поток.

Листинг 10.26. `signal_threads_alarm.py`

```
import signal
import time
import threading


def signal_handler(num, stack):
    print(time.ctime(), 'Alarm in',
          threading.currentThread().name)


signal.signal(signal.SIGALRM, signal_handler)


def use_alarm():
    t_name = threading.currentThread().name
    print(time.ctime(), 'Setting alarm in', t_name)
    signal.alarm(1)
    print(time.ctime(), 'Sleeping in', t_name)
```

```
    time.sleep(3)
    print(time.ctime(), 'Done with sleep in', t_name)


# Запустить поток, который не будет получать сигналы
alarm_thread = threading.Thread(
    target=use_alarm,
    name='alarm_thread',
)
alarm_thread.start()
time.sleep(0.1)

# Ожидать появления сигнала (этого никогда не произойдет!)
print(time.ctime(), 'Waiting for', alarm_thread.name)
alarm_thread.join()

print(time.ctime(), 'Exiting normally')
```

В этом примере сигналы таймера не могут прекратить выполнение функции `sleep()` в функции `use_alarm()`.

```
$ python3 signal_threads_alarm.py

Sun Sep 11 11:31:22 2016 Setting alarm in alarm_thread
Sun Sep 11 11:31:22 2016 Sleeping in alarm_thread
Sun Sep 11 11:31:22 2016 Waiting for alarm_thread
Sun Sep 11 11:31:23 2016 Alarm in MainThread
Sun Sep 11 11:31:25 2016 Done with sleep in alarm_thread
Sun Sep 11 11:31:25 2016 Exiting normally
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `signal`[4].
- **PEP 475**[5]. *Retry system calls failing with EINTR.*
- `subprocess` (раздел 10.1). Приведены дополнительные примеры передачи сигналов процессам.
- Раздел 17.3.10. Пример использования функции `kill()` для передачи сигналов между процессами.

10.3. `threading`: управление параллельными вычислениями в рамках одного процесса

Модуль `threading` предоставляет программные интерфейсы для управления несколькими потоками управления, используя которые программа может выполнять одновременно (параллельно) несколько операций в адресном пространстве одного и того же процесса.

[4] https://docs.python.org/3.5/library/signal.html

[5] www.python.org/dev/peps/pep-0475

10.3.1. Объекты Thread

Простейший способ использования класса Thread — это его инстанциализация с указанием целевой функции и вызов метода start(), запускающего поток.

Листинг 10.27. threading_simple.py

```
import threading


def worker():
    """Функция рабочего потока"""
    print('Worker')


threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

Вывод состоит из пяти строк, каждая из которых содержит текст "Worker".

```
$ python3 threading_simple.py

Worker
Worker
Worker
Worker
Worker
```

Полезно иметь возможность создать поток с передачей ему аргументов, сообщающих, какую работу необходимо выполнить. В качестве аргумента потоку может быть передан объект любого типа. В следующем примере таким аргументом является число, которое поток выводит на печать.

Листинг 10.28. threading_simpleargs.py

```
import threading


def worker(num):
    """Функция рабочего потока"""
    print('Worker: %s' % num)


threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
```

Теперь в сообщения, выводимые каждым потоком, включается целочисленный аргумент.

```
$ python3 threading_simpleargs.py

Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

10.3.2. Определение текущего потока

Использование аргументов для идентификации или именования потоков выглядит неуклюже и не является необходимым. Каждый экземпляр Thread получает имя с суффиксом по умолчанию, который может изменяться по мере создания новых потоков. Присваивание потокам имен полезно в серверных процессах, в которых несколько служебных потоков управляют различными операциями.

Листинг 10.29. threading_names.py

```
import threading
import time


def worker():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.2)
    print(threading.current_thread().getName(), 'Exiting')


def my_service():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(0.3)
    print(threading.current_thread().getName(), 'Exiting')


t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker)  # использовать имя
                                      # по умолчанию

w.start()
w2.start()
t.start()
```

Отладочный вывод включает строки, каждая из которых содержит имя текущего потока. Строкам, содержащим имя "Thread-1", соответствует неименованный поток w2.

```
$ python3 threading_names.py

worker Starting
Thread-1 Starting
my_service Starting
```

```
worker Exiting
Thread-1 Exiting
my_service Exiting
```

Как правило, для отладки программ не используют функцию `print`. Модуль `logging` (раздел 14.8) поддерживает встраивание имени потока в каждое протоколируемое сообщение с использованием спецификатора формата `%(threadName)s`. Включение имен потоков в протоколируемые сообщения позволяет отслеживать источники этих сообщений.

Листинг 10.30. `threading_names_log.py`

```
import logging
import threading
import time


def worker():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')


def my_service():
    logging.debug('Starting')
    time.sleep(0.3)
    logging.debug('Exiting')


logging.basicConfig(
    level=logging.DEBUG,
    format='[%(levelname)s] (%(threadName)-10s) %(message)s',
)

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker)  # использовать имя по умолчанию

w.start()
w2.start()
t.start()
```

Кроме того, модуль `logging` (раздел 14.8) потокобезопасен, и поэтому в выводе отражаются различия между сообщениями, поступающими из различных потоков.

```
$ python3 threading_names_log.py

[DEBUG] (worker    ) Starting
[DEBUG] (Thread-1  ) Starting
[DEBUG] (my_service) Starting
[DEBUG] (worker    ) Exiting
[DEBUG] (Thread-1  ) Exiting
[DEBUG] (my_service) Exiting
```

10.3.3. Потоки, являющиеся и не являющиеся демонами

До сих пор во всех примерах предполагалось, что выход из программы неявно откладывался до тех пор, пока все потоки не завершат свою работу. Однако иногда программы могут создавать потоки-демоны, выполнение которых не блокирует выход из основной программы. Потоки такого типа удобно использовать в службах в тех случаях, когда прерывание работы потока затруднено или же его преждевременное завершение не может привести к потере или повреждению данных (например, если поток генерирует контрольные сигналы в целях мониторинга службы). Для указания того, что поток создается как демон, следует передать аргумент `daemon=True` его конструктору или вызвать метод `set_daemon()`, передав ему значение `True`. Создаваемые потоки по умолчанию не являются демонами.

Листинг 10.31. `threading_daemon.py`

```
import threading
import time
import logging


def daemon():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')


def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

d = threading.Thread(name='daemon', target=daemon, daemon=True)

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

Результирующий вывод не включает строку сообщения `"Exiting"` из потока-демона, поскольку все остальные потоки (включая основной поток) завершаются до того, как он успеет пробудиться после вызова функции `sleep()`.

```
$ python3 threading_daemon.py

(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
```

Чтобы дождаться завершения работы потока-демона, следует использовать метод `join()`.

Листинг 10.32. `threading_daemon_join.py`

```
import threading
import time
import logging


def daemon():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')


def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

d = threading.Thread(name='daemon', target=daemon, daemon=True)

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join()
t.join()
```

Ожидание завершения потока-демона с помощью метода `join()` означает, что этот поток имеет возможность передать свое сообщение `"Exiting"`.

```
$ python3 threading_daemon_join.py

(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
(daemon    ) Exiting
```

По умолчанию метод `join()` блокируется на бесконечное время, однако ему можно передать аргумент в виде значения с плавающей точкой — тайм-аута, определяющего предельное время ожидания перехода потока в неактивное состояние. Если поток не успеет завершиться в течение указанного времени, то метод `join()` в любом случае выполнит возврат.

Листинг 10.33. threading_daemon_join_timeout.py

```
import threading
import time
import logging


def daemon():
    logging.debug('Starting')
    time.sleep(0.2)
    logging.debug('Exiting')


def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

d = threading.Thread(name='daemon', target=daemon, daemon=True)

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join(0.1)
print('d.isAlive()', d.isAlive())
t.join()
```

Поскольку значение тайм-аута меньше длительности "сна" потока-демона, поток остается активным даже после того, как будет выполнен возврат из метода `join()`.

```
$ python3 threading_daemon_join_timeout.py

(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
d.isAlive() True
```

10.3.4. Перечисление всех потоков

Чтобы обеспечить завершение всех потоков-демонов прежде, чем завершится основной поток, вовсе не обязательно хранить явные дескрипторы каждого из них. Метод `enumerate()` возвращает список всех активных экземпляров `Thread`. А поскольку основной поток входит в этот список, то его присоединение к теку-

щему потоку с помощью метода `join()` создало бы ситуацию взаимоблокировки, так что он должен быть исключен.

Листинг 10.34. `threading_enumerate.py`

```
import random
import threading
import time
import logging


def worker():
    """Функция рабочего потока"""
    pause = random.randint(1, 5) / 10
    logging.debug('sleeping %0.2f', pause)
    time.sleep(pause)
    logging.debug('ending')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

for i in range(3):
    t = threading.Thread(target=worker, daemon=True)
    t.start()

main_thread = threading.main_thread()
for t in threading.enumerate():
    if t is main_thread:
        continue
    logging.debug('joining %s', t.getName())
    t.join()
```

Поскольку длительность бездействия каждого рабочего потока определяется случайным числом, результирующий вывод будет меняться от одного запуска сценария к другому.

```
$ python3 threading_enumerate.py

(Thread-1  ) sleeping 0.20
(Thread-2  ) sleeping 0.30
(Thread-3  ) sleeping 0.40
(MainThread) joining Thread-1
(Thread-1  ) ending
(MainThread) joining Thread-3
(Thread-2  ) ending
(Thread-3  ) ending
(MainThread) joining Thread-2
```

10.3.5. Использование подклассов Thread

Прежде всего объект Thread выполняет некоторые базовые операции инициализации, а затем вызывает метод run(), который, в свою очередь, вызывает целевую функцию, переданную конструктору. Создание подкласса Thread сводится к переопределению метода run() для выполнения любых необходимых действий.

Листинг 10.35. threading_subclass.py

```
import threading
import logging


class MyThread(threading.Thread):

    def run(self):
        logging.debug('running')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

for i in range(5):
    t = MyThread()
    t.start()
```

Возвращаемое методом run() значение игнорируется.

```
$ python3 threading_subclass.py

(Thread-1 ) running
(Thread-2 ) running
(Thread-3 ) running
(Thread-4 ) running
(Thread-5 ) running
```

Поскольку значения args и kwargs, переданные конструктору Thread, сохраняются в закрытых переменных, имена которых содержат префикс '__', доступ к ним из подкласса затруднен. Для передачи аргументов потоку пользовательского типа следует переопределить конструктор и сохранить эти значения в атрибуте экземпляра, чтобы они были видимы в подклассе.

Листинг 10.36. threading_subclass_args.py

```
import threading
import logging


class MyThreadWithArgs(threading.Thread):

    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, *, daemon=None):
```

```
        super().__init__(group=group, target=target, name=name,
                         daemon=daemon)
        self.args = args
        self.kwargs = kwargs

    def run(self):
        logging.debug('running with %s and %s',
                      self.args, self.kwargs)


logging.basicConfig(
    level=logging.DEBUG,
3    format='(%(threadName)-10s) %(message)s',
)

for i in range(5):
    t = MyThreadWithArgs(args=(i,), kwargs={'a': 'A', 'b': 'B'})
    t.start()
```

Класс `MyThreadWithArgs` использует тот же API, что и класс `Thread`, но другой класс может изменить метод конструктора так, чтобы он поддерживал большее количество аргументов, теснее связывая их с назначением потока, как это делается в случае любого другого класса.

```
$ python3 threading_subclass_args.py

(Thread-1 ) running with (0,) and {'b': 'B', 'a': 'A'}
(Thread-2 ) running with (1,) and {'b': 'B', 'a': 'A'}
(Thread-3 ) running with (2,) and {'b': 'B', 'a': 'A'}
(Thread-4 ) running with (3,) and {'b': 'B', 'a': 'A'}
(Thread-5 ) running with (4,) and {'b': 'B', 'a': 'A'}
```

10.3.6. Потоки `Timer`

Один из примеров, объясняющих причину того, почему приходится создавать подкласс `Thread`, предоставляет класс `Timer`, также содержащийся в модуле `threading`. Объект `Timer` начинает работать с некоторой задержкой, и его можно отменить в любой момент периода задержки.

Листинг 10.37. `threading_timer.py`

```
import threading
import time
import logging


def delayed():
    logging.debug('worker running')


logging.basicConfig(
    level=logging.DEBUG,
```

```
    format='(%(threadName)-10s) %(message)s',
)

t1 = threading.Timer(0.3, delayed)
t1.setName('t1')
t2 = threading.Timer(0.3, delayed)
t2.setName('t2')

logging.debug('starting timers')
t1.start()
t2.start()

logging.debug('waiting before canceling %s', t2.getName())
time.sleep(0.2)
logging.debug('canceling %s', t2.getName())
t2.cancel()
logging.debug('done')
```

В этом примере второй таймер не успевает вызвать функцию, тогда как первый выполняется после того, как выполнится остальная часть программы. Поскольку данный поток не является демоном, он присоединяется неявно по завершении основного потока.

```
$ python3 threading_timer.py

(MainThread) starting timers
(MainThread) waiting before canceling t2
(MainThread) canceling t2
(MainThread) done
(t1        ) worker running
```

10.3.7. Обмен сигналами между потоками

Несмотря на то что цель использования нескольких потоков — параллельное выполнение независимых операций, иногда важно синхронизировать операции, выполняемые в двух или более потоках. Объекты событий обеспечивают простой способ организации безопасного взаимодействия потоков. Объект `Event` имеет внутренний флаг, который другие потоки могут устанавливать и сбрасывать с помощью методов `set()` и `clear()`. С помощью метода `wait()` можно приостановить работу потока до тех пор, пока другой поток не установит указанный флаг, вызвав метод `set()`, или пока не истечет заданный интервал времени.

Листинг 10.38. `threading_event.py`

```
import logging
import threading
import time


def wait_for_event(e):
    """Дождаться установки события, прежде чем что-то делать."""
    logging.debug('wait_for_event starting')
```

```
    event_is_set = e.wait()
    logging.debug('event set: %s', event_is_set)


def wait_for_event_timeout(e, t):
    """Подождать t секунд и завершиться по тайм-ауту."""
    while not e.is_set():
        logging.debug('wait_for_event_timeout starting')
        event_is_set = e.wait(t)
        logging.debug('event set: %s', event_is_set)
        if event_is_set:
            logging.debug('processing event')
        else:
            logging.debug('doing other work')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

e = threading.Event()
t1 = threading.Thread(
    name='block',
    target=wait_for_event,
    args=(e,),
)
t1.start()

t2 = threading.Thread(
    name='nonblock',
    target=wait_for_event_timeout,
    args=(e, 2),
)
t2.start()

logging.debug('Waiting before calling Event.set()')
time.sleep(0.3)
e.set()
logging.debug('Event is set')
```

Метод `wait()` получает один аргумент, определяющий предельное время ожидания события (в секундах). Возвращаемое им булево значение указывает на то, установлен ли внутренний флаг, благодаря чему вызывающему коду становится известно, по какой причине метод `wait()` вернул управление. Чтобы проверить состояние этого флага для заданного события без риска блокировки, можно использовать метод `is_set()`.

В данном примере функция `wait_for_event_timeout()` проверяет статус события, не создавая бесконечной блокировки. Функция `wait_for_event()` блокируется на вызове метода `wait()`, который не возвращает управление до тех пор, пока не изменится статус события.

```
$ python3 threading_event.py

(block     ) wait_for_event starting
(nonblock  ) wait_for_event_timeout starting
(MainThread) Waiting before calling Event.set()
(MainThread) Event is set
(nonblock  ) event set: True
(nonblock  ) processing event
(block     ) event set: True
```

10.3.8. Управление доступом к ресурсам

Кроме синхронизации операций, выполняемых потоками, очень важно иметь возможность контролировать доступ к разделяемым ресурсам во избежание потери данных или их повреждения. Потоковая безопасность встроенных структур данных Python (например, списков или словарей) является следствием того факта, что для манипулирования ими используются атомарные операции, выполняемые байт-кодом (глобальная блокировка интерпретатора, защищающая внутренние структуры данных, не освобождается в процессе выполнения операций, обновляющих данные). Другие структуры данных, реализуемые в Python, или более простые типы наподобие целых чисел или чисел с плавающей точкой не имеют такой защиты. Для защиты объекта от попыток одновременного доступа к нему следует использовать объект `Lock`.

Листинг 10.39. `threading_lock.py`

```
import logging
import random
import threading
import time


class Counter:

    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.value = start

    def increment(self):
        logging.debug('Waiting for lock')
        self.lock.acquire()
        try:
            logging.debug('Acquired lock')
            self.value = self.value + 1
        finally:
            self.lock.release()


def worker(c):
    for i in range(2):
        pause = random.random()
```

```
        logging.debug('Sleeping %0.02f', pause)
        time.sleep(pause)
        c.increment()
    logging.debug('Done')


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

counter = Counter()
for i in range(2):
    t = threading.Thread(target=worker, args=(counter,))
    t.start()

logging.debug('Waiting for worker threads')
main_thread = threading.main_thread()
for t in threading.enumerate():
    if t is not main_thread:
        t.join()
logging.debug('Counter: %d', counter.value)
```

В этом примере функция `worker()` увеличивает значение счетчика в экземпляре `Counter`, который управляет объектом `Lock`, предотвращающим изменение его внутреннего состояния одновременно двумя потоками. Если бы объект `Lock` не использовался, то некоторые значения атрибута `value` могли быть пропущены.

```
$ python3 threading_lock.py

(Thread-1  ) Sleeping 0.18
(Thread-2  ) Sleeping 0.93
(MainThread) Waiting for worker threads
(Thread-1  ) Waiting for lock
(Thread-1  ) Acquired lock
(Thread-1  ) Sleeping 0.11
(Thread-1  ) Waiting for lock
(Thread-1  ) Acquired lock
(Thread-1  ) Done
(Thread-2  ) Waiting for lock
(Thread-2  ) Acquired lock
(Thread-2  ) Sleeping 0.81
(Thread-2  ) Waiting for lock
(Thread-2  ) Acquired lock
(Thread-2  ) Done
(MainThread) Counter: 4
```

Чтобы определить, получил ли другой поток блокировку, не останавливая текущий поток, следует передать методу `acquire()` значение `False` в качестве аргумента. В следующем примере функция `worker()` пытается запросить блокировку три раза и подсчитывает количество таких попыток. В это время функция `lock_holder()`, используя цикл, периодически захватывает и освобождает блокиров-

ку, делая короткие паузы в каждом из состояний для имитации процесса загрузки данных.

Листинг 10.40. threading_lock_noblock.py

```
import logging
import threading
import time


def lock_holder(lock):
    logging.debug('Starting')
    while True:
        lock.acquire()
        try:
            logging.debug('Holding')
            time.sleep(0.5)
        finally:
            logging.debug('Not holding')
            lock.release()
        time.sleep(0.5)


def worker(lock):
    logging.debug('Starting')
    num_tries = 0
    num_acquires = 0
    while num_acquires < 3:
        time.sleep(0.5)
        logging.debug('Trying to acquire')
        have_it = lock.acquire(0)
        try:
            num_tries += 1
            if have_it:
                logging.debug('Iteration %d: Acquired',
                              num_tries)
                num_acquires += 1
            else:
                logging.debug('Iteration %d: Not acquired',
                              num_tries)
        finally:
            if have_it:
                lock.release()
    logging.debug('Done after %d iterations', num_tries)


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

lock = threading.Lock()
```

```
holder = threading.Thread(
    target=lock_holder,
    args=(lock,),
    name='LockHolder',
    daemon=True,
)
holder.start()

worker = threading.Thread(
    target=worker,
    args=(lock,),
    name='Worker',
)
worker.start()
```

Чтобы получить блокировку три раза, методу `worker()` потребовалось более трех итераций.

```
$ python3 threading_lock_noblock.py

(LockHolder) Starting
(LockHolder) Holding
(Worker    ) Starting
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 1: Acquired
(LockHolder) Holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 2: Not acquired
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 3: Acquired
(LockHolder) Holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 4: Not acquired
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 5: Acquired
(Worker    ) Done after 5 iterations
```

10.3.8.1. Повторная блокировка

Обычные объекты блокировки типа `Lock` не могут быть получены более одного раза даже одним и тем же потоком. Получение блокировки более чем одной функцией в одной и той же цепочке вызовов может породить нежелательные побочные эффекты.

Листинг 10.41. `threading_lock_reacquire.py`

```
import threading

lock = threading.Lock()
```

```
print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
```

В данном случае методу `acquire()` во втором вызове передается нулевое значение тайм-аута, предотвращающее его блокирование, поскольку блокировка уже была получена первым вызовом.

```
$ python3 threading_lock_reacquire.py

First try : True
Second try: False
```

В ситуациях, когда другой код в том же потоке должен повторно получить блокировку, следует использовать объект `RLock`.

Листинг 10.42. `threading_rlock.py`

```
import threading

lock = threading.RLock()

print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
```

Единственное, чем отличается этот код от предыдущего, — это использование объекта `RLock` вместо объекта `Lock`.

```
$ python3 threading_rlock.py

First try : True
Second try: True
```

10.3.8.2. Блокировки как менеджеры контекста

Объекты `Lock` реализуют API менеджера контекста и совместимы с инструкцией `with`. Использование инструкции `with` устраняет необходимость в явном получении и освобождении блокировки.

Листинг 10.43. `threading_lock_with.py`

```
import threading
import logging


def worker_with(lock):
    with lock:
        logging.debug('Lock acquired via with')


def worker_no_with(lock):
    lock.acquire()
    try:
        logging.debug('Lock acquired directly')
    finally:
```

```
        lock.release()


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

lock = threading.Lock()
w = threading.Thread(target=worker_with, args=(lock,))
nw = threading.Thread(target=worker_no_with, args=(lock,))

w.start()
nw.start()
```

В основе управления блокировками с помощью функций `worker_with()` и `worker_no_with()` лежит один и тот же принцип.

```
$ python3 threading_lock_with.py

(Thread-1  ) Lock acquired via with
(Thread-2  ) Lock acquired directly
```

10.3.9. Синхронизация потоков

Помимо объектов `Event` существует другой способ синхронизации потоков: с помощью объектов `Condition`. Поскольку класс `Condition` использует класс `Lock`, объект этого типа можно связать с разделяемым ресурсом, позволяя нескольким потокам ожидать, пока данный ресурс не будет обновлен. В следующем примере потоки `consumer()` ожидают, пока не будет установлен объект `Condition`, прежде чем продолжить выполнение. За установку условия и извещение других потоков о том, что они могут продолжить выполнение, отвечает поток `producer()`.

Листинг 10.44. `threading_condition.py`

```
import logging
import threading
import time


def consumer(cond):
    """Дождаться наступления условия и затем использовать ресурс."""
    logging.debug('Starting consumer thread')
    with cond:
        cond.wait()
        logging.debug('Resource is available to consumer')


def producer(cond):
    """Настроить ресурс для использования потребителем."""
    logging.debug('Starting producer thread')
```

```
    with cond:
        logging.debug('Making resource available')
        cond.notifyAll()


logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s (%(threadName)-2s) %(message)s',
)

condition = threading.Condition()
c1 = threading.Thread(name='c1', target=consumer,
                      args=(condition,))
c2 = threading.Thread(name='c2', target=consumer,
                      args=(condition,))
p = threading.Thread(name='p', target=producer,
                     args=(condition,))

c1.start()
time.sleep(0.2)
c2.start()
time.sleep(0.2)
p.start()
```

Для получения блокировки, связанной с изменением состояния объекта `Condition`, потоки используют инструкцию `with`. Допускается также явное использование методов `acquire()` и `release()`.

```
$ python3 threading_condition.py

2016-07-10 10:45:28,170 (c1) Starting consumer thread
2016-07-10 10:45:28,376 (c2) Starting consumer thread
2016-07-10 10:45:28,581 (p ) Starting producer thread
2016-07-10 10:45:28,581 (p ) Making resource available
2016-07-10 10:45:28,582 (c1) Resource is available to consumer
2016-07-10 10:45:28,582 (c2) Resource is available to consumer
```

Еще одним механизмом синхронизации являются барьеры. Объект `Barrier` создает контрольную точку для заданного количества потоков, каждый из которых, достигнув контрольной точки, блокируется до тех пор, пока ее не достигнут все потоки, участвующие в этом механизме блокировки. При таком подходе потоки могут запускаться по отдельности, а затем переходить в состояние ожидания до тех пор, пока все они не будут готовы к продолжению выполнения.

Листинг 10.45. `threading_barrier.py`

```
import threading
import time


def worker(barrier):
    print(threading.current_thread().name,
          'waiting for barrier with {} others'.format(
```

```
            barrier.n_waiting))
    worker_id = barrier.wait()
    print(threading.current_thread().name, 'after barrier',
          worker_id)


NUM_THREADS = 3

barrier = threading.Barrier(NUM_THREADS)

threads = [
    threading.Thread(
        name='worker-%s' % i,
        target=worker,
        args=(barrier,),
    )
    for i in range(NUM_THREADS)
]

for t in threads:
    print(t.name, 'starting')
    t.start()
    time.sleep(0.1)

for t in threads:
    t.join()
```

В этом примере объект `Barrier` конфигурируется для блокирования потоков в контрольной точке до тех пор, пока все три потока не перейдут в состояние ожидания. Как только это условие выполняется, все три потока освобождаются одновременно. Значение, возвращаемое методом `wait()`, указывает на количество освобожденных процессов-участников и может использоваться для наложения ограничений на выполнение потоками таких действий, как освобождение совместно используемого ресурса.

```
$ python3 threading_barrier.py

worker-0 starting
worker-0 waiting for barrier with 0 others
worker-1 starting
worker-1 waiting for barrier with 1 others
worker-2 starting
worker-2 waiting for barrier with 2 others
worker-2 after barrier 2
worker-0 after barrier 0
worker-1 after barrier 1
```

Вызов метода `abort()` объекта `Barrier` генерирует во всех ожидающих потоках исключение `BrokenBarrierError`. Благодаря этому потоки получают возможность выполнить завершающие операции по освобождению ресурсов, если их работа была прервана в то время, когда они были заблокированы вызовом `wait()`.

Листинг 10.46. threading_barrier_abort.py

```
import threading
import time


def worker(barrier):
    print(threading.current_thread().name,
          'waiting for barrier with {} others'.format(
              barrier.n_waiting))
    try:
        worker_id = barrier.wait()
    except threading.BrokenBarrierError:
        print(threading.current_thread().name, 'aborting')
    else:
        print(threading.current_thread().name, 'after barrier',
              worker_id)


NUM_THREADS = 3

barrier = threading.Barrier(NUM_THREADS + 1)

threads = [
    threading.Thread(
        name='worker-%s' % i,
        target=worker,
        args=(barrier,),
    )
    for i in range(NUM_THREADS)
]

for t in threads:
    print(t.name, 'starting')
    t.start()
    time.sleep(0.1)

barrier.abort()

for t in threads:
    t.join()
```

В этом примере объект `Barrier` конфигурируется таким образом, чтобы блокировать в контрольной точке на один поток больше их фактического количества, что приводит к блокированию всех трех потоков. Вызов `abort()` возбуждает исключение в каждом из заблокированных потоков.

```
$ python3 threading_barrier_abort.py

worker-0 starting
worker-0 waiting for barrier with 0 others
worker-1 starting
worker-1 waiting for barrier with 1 others
```

```
worker-2 starting
worker-2 waiting for barrier with 2 others
worker-0 aborting
worker-2 aborting
worker-1 aborting
```

10.3.10. Ограничение одновременного доступа к ресурсам

Иногда требуется разрешить одновременный доступ к ресурсу нескольким потокам, но при этом ограничить их общее количество. В качестве примера можно привести пул, поддерживающий фиксированное количество соединений, или сетевое приложение, поддерживающее фиксированное количество одновременных загрузок. Одним из способов управления подобными соединениями являются семафоры, представляемые классом `Semaphore`.

Листинг 10.47. `threading_semaphore.py`

```
import logging
import random
import threading
import time


class ActivePool:

    def __init__(self):
        super(ActivePool, self).__init__()
        self.active = []
        self.lock = threading.Lock()

    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
            logging.debug('Running: %s', self.active)

    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
            logging.debug('Running: %s', self.active)


def worker(s, pool):
    logging.debug('Waiting to join the pool')
    with s:
        name = threading.current_thread().getName()
        pool.makeActive(name)
        time.sleep(0.1)
        pool.makeInactive(name)


logging.basicConfig(
    level=logging.DEBUG,
```

```
    format='%(asctime)s (%(threadName)-2s) %(message)s',
)

pool = ActivePool()
s = threading.Semaphore(2)
for i in range(4):
    t = threading.Thread(
        target=worker,
        name=str(i),
        args=(s, pool),
    )
    t.start()
```

В этом примере класс `ActivePool` обеспечивает удобный способ отслеживания потоков, способных выполняться в данный момент. Вероятно, при работе с реальным пулом ресурсов он распределял бы соединения или другие значения между активными потоками и возвращал их в пул по завершении потока. В данном случае пул используется всего лишь для хранения имен активных потоков и демонстрации того факта, что одновременно выполняется не более двух потоков.

```
$ python3 threading_semaphore.py

2016-07-10 10:45:29,398 (0 ) Waiting to join the pool
2016-07-10 10:45:29,398 (0 ) Running: ['0']
2016-07-10 10:45:29,399 (1 ) Waiting to join the pool
2016-07-10 10:45:29,399 (1 ) Running: ['0', '1']
2016-07-10 10:45:29,399 (2 ) Waiting to join the pool
2016-07-10 10:45:29,399 (3 ) Waiting to join the pool
2016-07-10 10:45:29,501 (1 ) Running: ['0']
2016-07-10 10:45:29,501 (0 ) Running: []
2016-07-10 10:45:29,502 (3 ) Running: ['3']
2016-07-10 10:45:29,502 (2 ) Running: ['3', '2']
2016-07-10 10:45:29,607 (3 ) Running: ['2']
2016-07-10 10:45:29,608 (2 ) Running: []
```

10.3.11. Данные, специфичные для потока

В то время как одни ресурсы должны блокироваться, чтобы их могли использовать несколько потоков, защита других должна быть организована таким образом, чтобы они скрывались от потоков, которые не владеют ими. Конструктор `local()` создает объект, способный скрывать значения от отдельных потоков.

Листинг 10.48. `threading_local.py`

```
import random
import threading
import logging


def show_value(data):
    try:
        val = data.value
    except AttributeError:
```

```
        logging.debug('No value yet')
    else:
        logging.debug('value=%s', val)


def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

local_data = threading.local()
show_value(local_data)
local_data.value = 1000
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()
```

Атрибут `local_data.value` отсутствует в любом потоке до тех пор, пока этот поток не установит его.

```
$ python3 threading_local.py

(MainThread) No value yet
(MainThread) value=1000
(Thread-1  ) No value yet
(Thread-1  ) value=33
(Thread-2  ) No value yet
(Thread-2  ) value=74
```

Для инициализации параметров таким образом, чтобы все потоки запускались с одним и тем же значением, можно использовать подкласс и установить атрибуты в методе `__init__()`.

Листинг 10.49. `threading_local_defaults.py`

```
import random
import threading
import logging


def show_value(data):
    try:
        val = data.value
    except AttributeError:
        logging.debug('No value yet')
    else:
```

```
        logging.debug('value=%s', val)


def worker(data):
    show_value(data)
    data.value = random.randint(1, 100)
    show_value(data)


class MyLocal(threading.local):

    def __init__(self, value):
        super().__init__()
        logging.debug('Initializing %r', self)
        self.value = value


logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

local_data = MyLocal(1000)
show_value(local_data)

for i in range(2):
    t = threading.Thread(target=worker, args=(local_data,))
    t.start()
```

Значения по умолчанию устанавливаются посредством вызова метода `__init__()` для одного и того же объекта (обратите внимание на значение `id()`) по одному разу в каждом потоке.

```
$ python3 threading_local_defaults.py

(MainThread) Initializing <__main__.MyLocal object at
0x101c6c288>
(MainThread) value=1000
(Thread-1  ) Initializing <__main__.MyLocal object at
0x101c6c288>
(Thread-1  ) value=1000
(Thread-1  ) value=18
(Thread-2  ) Initializing <__main__.MyLocal object at
0x101c6c288>
(Thread-2  ) value=1000
(Thread-2  ) value=77
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `threading`[6].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `threading` (раздел A.6.45).

[6] https://docs.python.org/3.5/library/threading.html

- `thread`. Низкоуровневый API потоков.
- `Queue`. Потокобезопасная очередь, которую удобно использовать для передачи сообщений между потоками.
- `multiprocessing` (раздел 10.4). API для работы с процессами, являющийся зеркальным отражением API `threading`.

10.4. `multiprocessing`: использование процессов вместо потоков

Модуль `multiprocessing` включает API, обеспечивающий распределение работы между несколькими процессами на основе API многопоточной обработки (см. раздел 10.3). В некоторых случаях использование процессов вместо потоков позволяет организовать параллельное выполнение задачи с использованием нескольких ядер CPU, реализация чего с помощью потоков невозможна из-за глобальной блокировки интерпретатора Python.

В силу сходства модулей `multiprocessing` и `threading` первые несколько примеров, приведенных в этом разделе, — это видоизмененные версии ранее рассмотренных примеров многопоточной обработки. Последующие примеры иллюстрируют возможности, предоставляемые модулем `multiprocessing`, но недоступные в модуле `threading`.

10.4.1. Основы многопроцессной обработки

Проще всего можно запустить дополнительный процесс, создав объект `Process` с целевой функцией и вызвав его метод `start()`, чтобы разрешить выполнение процесса.

Листинг 10.50. `multiprocessing_simple.py`

```
import multiprocessing


def worker():
    """Рабочая функция"""
    print('Worker')


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker)
        jobs.append(p)
        p.start()
```

Этот сценарий выводит пять строк, каждая из которых содержит слово "Worker".

```
$ python3 multiprocessing_simple.py

Worker
```

```
Worker
Worker
Worker
Worker
```

Порождаемому процессу можно передавать аргументы, конкретизирующие работу, которую он должен выполнить. В отличие от модуля `threading`, аргументы, передаваемые объекту `Process` модуля `multiprocessing`, должны сериализоваться с помощью модуля `pickle` (см. раздел 7.1). В следующем примере каждому процессу передается его идентификатор, который выводится на консоль.

Листинг 10.51. `multiprocessing_simpleargs.py`

```
import multiprocessing


def worker(num):
    """Функция рабочего процесса"""
    print('Worker:', num)


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        jobs.append(p)
        p.start()
```

Теперь сообщения, выводимые каждым процессом, включают целочисленный аргумент, хотя вывод не всегда может выглядеть столь же аккуратно, как показано ниже, поскольку процессы конкурирует между собой за доступ к выходному потоку, а очередность их выполнения не детерминирована.

```
$ python3 multiprocessing_simpleargs.py

Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

10.4.2. Импортируемые целевые функции

Одним из отличий примеров, иллюстрирующих многопроцессную обработку, от примеров, которые приводились при обсуждении многопоточности, является дополнительная защита той части кода, которая должна выполняться только при запуске сценария как основной программы. В силу особенностей способа запуска новых процессов дочерний процесс должен иметь возможность импортировать сценарий, содержащий целевую функцию. Обертывание основной части приложения кодом проверки на `__main__` гарантирует, что она не будет выполняться рекурсивно в каждом дочернем процессе при импортировании модуля. Целевую функцию можно также импортировать из другого сценария. Например, в сцена-

рии multiprocessing_import_main.py используется рабочая функция, определенная в другом модуле.

Листинг 10.52. multiprocessing_import_main.py

```
import multiprocessing
import multiprocessing_import_worker

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(
            target=multiprocessing_import_worker.worker,
        )
        jobs.append(p)
        p.start()
```

Рабочая функция определена в сценарии multiprocessing_import_worker.py.

Листинг 10.53. multiprocessing_import_worker.py

```
def worker():
    """worker function"""
    print('Worker')
    return
```

Вызов основной программы приводит к тем же результатам, что и в первом примере.

```
$ python3 multiprocessing_import_main.py

Worker
Worker
Worker
Worker
Worker
```

10.4.3. Определение текущего процесса

Передача аргумента для идентификации процесса или присвоения ему имени — это хлопоты, от которых можно избавиться. Каждый экземпляр Process получает имя по умолчанию, которое можно изменить при создании процесса. Именование процессов удобно для их отслеживания, особенно в приложениях, в которых одновременно выполняется несколько различных типов процессов.

Листинг 10.54. multiprocessing_names.py

```
import multiprocessing
import time


def worker():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
```

```
    time.sleep(2)
    print(name, 'Exiting')


def my_service():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
    time.sleep(3)
    print(name, 'Exiting')


if __name__ == '__main__':
    service = multiprocessing.Process(
        name='my_service',
        target=my_service,
    )
    worker_1 = multiprocessing.Process(
        name='worker 1',
        target=worker,
    )
    worker_2 = multiprocessing.Process(  # default name
        target=worker,
    )

    worker_1.start()
    worker_2.start()
    service.start()
```

Отладочный вывод включает имя процесса в каждой строке. Строки с именем `Process-3` соответствуют неименованному процессу `worker_2`.

```
$ python3 multiprocessing_names.py

worker 1 Starting
worker 1 Exiting
Process-3 Starting
Process-3 Exiting
my_service Starting
my_service Exiting
```

10.4.4. Процессы-демоны

По умолчанию выход из основной программы осуществляется лишь после того, как завершатся все дочерние процессы. Но иногда запускаются фоновые процессы, которые выполняются, не блокируя выход из основной программы, что, например, может быть полезным в случае служб, которым непросто прервать выполнение рабочего процесса, или же когда прерывание процесса посреди выполняемой им задачи не может привести к потере или повреждению ценных данных (например, в случае задачи, генерирующей контрольные сигналы для средств мониторинга службы).

Можно указать, что процесс является демоном, установив для его атрибута daemon значение True. По умолчанию создаются процессы, не являющиеся демонами.

Листинг 10.55. multiprocessing_daemon.py

```
import multiprocessing
import time
import sys


def daemon():
    p = multiprocessing.current_process()
    print('Starting:', p.name, p.pid)
    sys.stdout.flush()
    time.sleep(2)
    print('Exiting :', p.name, p.pid)
    sys.stdout.flush()


def non_daemon():
    p = multiprocessing.current_process()
    print('Starting:', p.name, p.pid)
    sys.stdout.flush()
    print('Exiting :', p.name, p.pid)
    sys.stdout.flush()


if __name__ == '__main__':
    d = multiprocessing.Process(
        name='daemon',
        target=daemon,
    )
    d.daemon = True

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()
```

Вывод не включает сообщение "Exiting" из процесса-демона, поскольку все процессы, не являющиеся демонами (включая основную программу), успевают завершиться, прежде чем процесс-демон пробудится после приостановки на 2 секунды.

```
$ python3 multiprocessing_daemon.py

Starting: daemon 70880
Starting: non-daemon 70881
Exiting : non-daemon 70881
```

Выполнение процесса-демона автоматически прекращается перед выходом из программы, что позволяет избежать ситуаций, в которых после завершения основной программы в памяти остаются процессы-сироты. Такой характер поведения можно проконтролировать, выведя идентификатор процесса во время выполнения программы, а затем проверив список выполняющихся процессов с помощью команды ps.

10.4.5. Ожидание завершения процессов

Чтобы дождаться завершения процесса, следует использовать метод join().

Листинг 10.56. multiprocessing_daemon_join.py

```
import multiprocessing
import time
import sys


def daemon():
    name = multiprocessing.current_process().name
    print('Starting:', name)
    time.sleep(2)
    print('Exiting :', name)


def non_daemon():
    name = multiprocessing.current_process().name
    print('Starting:', name)
    print('Exiting :', name)


if __name__ == '__main__':
    d = multiprocessing.Process(
        name='daemon',
        target=daemon,
    )
    d.daemon = True

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()

    d.join()
    n.join()
```

На этот раз, поскольку основной процесс ожидает завершения процесса-демона, используя метод join(), соответствующая строка "Exiting" есть в выводе.

```
$ python3 multiprocessing_daemon_join.py

Starting: non-daemon
Exiting : non-daemon
Starting: daemon
Exiting : daemon
```

По умолчанию метод `join()` создает бесконечную блокировку. Передача ему аргумента `timeout` (число с плавающей точкой) позволяет определить предельную длительность периода ожидания (тайм-аут) в секундах. Если процесс не завершается в течение отведенного ему времени, то метод `join()` возвращает управление.

Листинг 10.57. `multiprocessing_daemon_join_timeout.py`

```
import multiprocessing
import time
import sys


def daemon():
    name = multiprocessing.current_process().name
    print('Starting:', name)
    time.sleep(2)
    print('Exiting :', name)


def non_daemon():
    name = multiprocessing.current_process().name
    print('Starting:', name)
    print('Exiting :', name)


if __name__ == '__main__':
    d = multiprocessing.Process(
        name='daemon',
        target=daemon,
    )
    d.daemon = True

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )
    n.daemon = False

    d.start()
    n.start()

    d.join(1)
    print('d.is_alive()', d.is_alive())
    n.join()
```

Поскольку длительность тайм-аута меньше длительности периода, на который была приостановлена работа демона, процесс продолжает оставаться активным после возврата из метода `join()`.

```
$ python3 multiprocessing_daemon_join_timeout.py

Starting: non-daemon
Exiting : non-daemon
d.is_alive() True
```

10.4.6. Прекращение работы процесса

Несмотря на то что для отправки сигнала, извещающего процесс о том, что он должен завершить работу, лучше использовать метод "отравленнной таблетки" (раздел 10.4.10), в тех случаях, когда вероятно зависание процесса или создание ситуации взаимоблокировки, полезно иметь возможность принудительного завершения процесса. Это обеспечивается вызовом метода `terminate()` для объекта дочернего процесса.

Листинг 10.58. `multiprocessing_terminate.py`

```
import multiprocessing
import time


def slow_worker():
    print('Starting worker')
    time.sleep(0.1)
    print('Finished worker')


if __name__ == '__main__':
    p = multiprocessing.Process(target=slow_worker)
    print('BEFORE:', p, p.is_alive())

    p.start()
    print('DURING:', p, p.is_alive())

    p.terminate()
    print('TERMINATED:', p, p.is_alive())

    p.join()
    print('JOINED:', p, p.is_alive())
```

Примечание

Важно вызвать метод `join()` для процесса после принудительного прекращения его работы, чтобы код, управляющий процессом, имел достаточно времени для обновления состояния объекта с учетом преждевременного завершения выполнения.

```
$ python3 multiprocessing_terminate.py

BEFORE: <Process(Process-1, initial)> False
DURING: <Process(Process-1, started)> True
TERMINATED: <Process(Process-1, started)> True
JOINED: <Process(Process-1, stopped[SIGTERM])> False
```

10.4.7. Код завершения процесса

Код завершения процесса доступен через атрибут `exitcode`. Диапазон его возможных значений приведен в табл. 10.1.

Таблица 10.1. Коды завершения процессов при многопроцессной обработке

Код завершения	Описание
`== 0`	Ошибки не возникали
`> 0`	В процессе возникла ошибка, и он завершился с данным кодом
`< 0`	Выполнение процесса было принудительно прекращено с помощью сигнала `-1 * код_завершения`

Листинг 10.59. `multiprocessing_exitcode.py`

```
import multiprocessing
import sys
import time


def exit_error():
    sys.exit(1)


def exit_ok():
    return


def return_value():
    return 1


def raises():
    raise RuntimeError('There was an error!')


def terminated():
    time.sleep(3)


if __name__ == '__main__':
    jobs = []
    funcs = [
        exit_error,
        exit_ok,
```

```
        return_value,
        raises,
        terminated,
    ]
    for f in funcs:
        print('Starting process for', f.__name__)
        j = multiprocessing.Process(target=f, name=f.__name__)
        jobs.append(j)
        j.start()

    jobs[-1].terminate()

    for j in jobs:
        j.join()
        print('{:>15}.exitcode = {}'.format(j.name, j.exitcode))
```

Процессы, в которых возникло исключение, автоматически получают код завершения 1.

```
$ python3 multiprocessing_exitcode.py

Starting process for exit_error
Starting process for exit_ok
Starting process for return_value
Starting process for raises
Starting process for terminated
Process raises:
Traceback (most recent call last):
  File ".../lib/python3.5/multiprocessing/process.py", line 249,
in _bootstrap
    self.run()
  File ".../lib/python3.5/multiprocessing/process.py", line 93,
in run
    self._target(*self._args, **self._kwargs)
  File "multiprocessing_exitcode.py", line 28, in raises
    raise RuntimeError('There was an error!')
RuntimeError: There was an error!
     exit_error.exitcode = 1
        exit_ok.exitcode = 0
   return_value.exitcode = 0
         raises.exitcode = 1
     terminated.exitcode = -15
```

10.4.8. Протоколирование

При отладке кода, связанного с параллельными вычислениями, полезно иметь доступ к внутреннему состоянию объектов, предоставляемых модулем `multiprocessing`. Для активизации протоколирования предусмотрена удобная функция `log_to_stderr()` уровня модуля. Она настраивает объект протоколирования ("логгер"), используя модуль `logging` (раздел 14.80), и добавляет обработчик, обеспечивая вывод сообщений в стандартный канал ошибок.

Листинг 10.60. multiprocessing_log_to_stderr.py

```
import multiprocessing
import logging
import sys


def worker():
    print('Doing some work')
    sys.stdout.flush()


if __name__ == '__main__':
    multiprocessing.log_to_stderr(logging.DEBUG)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

По умолчанию устанавливается уровень протоколирования NOTSET, при котором сообщения не записываются в журнал (не выводятся). Инициализировав объект протоколирования другим значением, можно установить желаемый уровень детализации сообщений.

```
$ python3 multiprocessing_log_to_stderr.py

[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[DEBUG/Process-1] running all "atexit" finalizers with priority
>= 0
[DEBUG/Process-1] running the remaining "atexit" finalizers
[INFO/Process-1] process exiting with exitcode 0
[INFO/MainProcess] process shutting down
[DEBUG/MainProcess] running all "atexit" finalizers with
priority >= 0
[DEBUG/MainProcess] running the remaining "atexit" finalizers
```

Для непосредственного манипулирования объектом протоколирования (изменения уровня детализации сообщений или добавления обработчиков) предназначена функция get_logger().

Листинг 10.61. multiprocessing_get_logger.py

```
import multiprocessing
import logging
import sys


def worker():
    print('Doing some work')
    sys.stdout.flush()


if __name__ == '__main__':
```

```
    multiprocessing.log_to_stderr()
    logger = multiprocessing.get_logger()
    logger.setLevel(logging.INFO)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

10.4.9. Создание подклассов `Process`

Использование экземпляров `Process`, представляющих процессы, с передачей им целевой функции — простейший способ запуска задачи в отдельном процессе. Однако этот способ — не единственный. Другой вариант — использовать пользовательский подкласс.

Листинг 10.62. `multiprocessing_subclass.py`

```
import multiprocessing


class Worker(multiprocessing.Process):

    def run(self):
        print('In {}'.format(self.name))
        return


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = Worker()
        jobs.append(p)
        p.start()
    for j in jobs:
        j.join()
```

Для выполнения возложенной на него работы производный класс должен переопределить метод `run()`.

```
$ python3 multiprocessing_subclass.py

In Worker-1
In Worker-2
In Worker-3
In Worker-4
In Worker-5
```

10.4.10. Передача сообщений процессам

Как и в случае потоков, многопроцессная обработка чаще всего применяется для распределения задачи между несколькими рабочими процессами, выполняющимися параллельно. Как правило, эффективная реализация такого подхода, предполагающего разделение работы и аккумуляцию результатов, требует органи-

зации межпроцессного взаимодействия. Наиболее простым способом организации взаимодействия процессов в условиях многопроцессной обработки является двухсторонний обмен сообщениями с помощью объекта очереди `Queue`. Очередь можно использовать для передачи любого объекта, сериализуемого средствами модуля `pickle` (раздел 7.1).

Листинг 10.63. `multiprocessing_queue.py`

```
import multiprocessing


class MyFancyClass:

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print('Doing something fancy in {} for {}!'.format(
            proc_name, self.name))


def worker(q):
    obj = q.get()
    obj.do_something()


if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Ждать завершения рабочего процесса
    queue.close()
    queue.join_thread()
    p.join()
```

В этом коротком примере сообщение передается только одному процессу, после чего основной процесс ожидает, пока рабочий процесс завершит свою работу.

```
$ python3 multiprocessing_queue.py

Doing something fancy in Process-1 for Fancy Dan!
```

Ниже приведен более сложный пример управления несколькими рабочими процессами, которые получают данные из очереди `JoinableQueue` и возвращают результаты родительскому процессу. Для принудительного завершения рабочих процессов используется метод "отравленной таблетки". После настройки реальных задач основная программа добавляет в очередь по одному "стоп-значению" на рабочий процесс. Когда рабочему процессу встречается это специальное зна-

чение, он выходит из своего цикла обработки. Основной процесс использует метод `join()` очереди задач для того, чтобы организовать ожидание завершения всех задач, прежде чем обработать результаты.

Листинг 10.64. `multiprocessing_producer_consumer.py`

```
import multiprocessing
import time


class Consumer(multiprocessing.Process):

    def __init__(self, task_queue, result_queue):
        multiprocessing.Process.__init__(self)
        self.task_queue = task_queue
        self.result_queue = result_queue

    def run(self):
        proc_name = self.name
        while True:
            next_task = self.task_queue.get()
            if next_task is None:
                # Прекращение выполнения с помощью
                # "отравленной таблетки"
                print('{}: Exiting'.format(proc_name))
                self.task_queue.task_done()
                break
            print('{}: {}'.format(proc_name, next_task))
            answer = next_task()
            self.task_queue.task_done()
            self.result_queue.put(answer)


class Task:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self):
        time.sleep(0.1)  # имитация выполнения работы
        return '{self.a} * {self.b} = {product}'.format(
            self=self, product=self.a * self.b)

    def __str__(self):
        return '{self.a} * {self.b}'.format(self=self)


if __name__ == '__main__':
    # Создание очередей обмена сообщениями
    tasks = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()
```

```
    # Запуск потребителей
    num_consumers = multiprocessing.cpu_count() * 2
    print('Creating {} consumers'.format(num_consumers))
    consumers = [
        Consumer(tasks, results)
        for i in range(num_consumers)
    ]
    for w in consumers:
        w.start()

    # Помещение задач в очередь
    num_jobs = 10
    for i in range(num_jobs):
        tasks.put(Task(i, i))

    # Добавление "отравленной таблетки" для каждого потребителя
    for i in range(num_consumers):
        tasks.put(None)

    # Ожидание завершения всех задач
    tasks.join()

    # Вывод результатов
    while num_jobs:
        result = results.get()
        print('Result:', result)
        num_jobs -= 1
```

Несмотря на то что задачи помещаются в очередь последовательно одна за другой, они выполняются параллельно. Таким образом, нельзя дать никакой гарантии относительно того, в каком порядке они будут завершаться.

```
$ python3 -u multiprocessing_producer_consumer.py

Creating 8 consumers
Consumer-1: 0 * 0
Consumer-2: 1 * 1
Consumer-3: 2 * 2
Consumer-4: 3 * 3
Consumer-5: 4 * 4
Consumer-6: 5 * 5
Consumer-7: 6 * 6
Consumer-8: 7 * 7
Consumer-3: 8 * 8
Consumer-7: 9 * 9
Consumer-4: Exiting
Consumer-1: Exiting
Consumer-2: Exiting
Consumer-5: Exiting
Consumer-6: Exiting
Consumer-8: Exiting
Consumer-7: Exiting
Consumer-3: Exiting
```

```
Result: 6 * 6 = 36
Result: 2 * 2 = 4
Result: 3 * 3 = 9
Result: 0 * 0 = 0
Result: 1 * 1 = 1
Result: 7 * 7 = 49
Result: 4 * 4 = 16
Result: 5 * 5 = 25
Result: 8 * 8 = 64
Result: 9 * 9 = 81
```

10.4.11. Обмен сигналами между процессами

Класс Event, представляющий события, обеспечивает простой способ обмена информацией о состоянии между процессами. Объект события может находиться в одном из двух состояний: "установлено" и "не установлено". Пользователи объекта события могут дожидаться его перехода из состояния "не установлено" в состояние "установлено", используя необязательное значение тайм-аута.

Листинг 10.65. multiprocessing_event.py

```
import multiprocessing
import time


def wait_for_event(e):
    """Дождаться события, прежде чем делать что-либо."""
    print('wait_for_event: starting')
    e.wait()
    print('wait_for_event: e.is_set()->', e.is_set())


def wait_for_event_timeout(e, t):
    """Подождать t секунд и затем завершиться по тайм-ауту."""
    print('wait_for_event_timeout: starting')
    e.wait(t)
    print('wait_for_event_timeout: e.is_set()->', e.is_set())


if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(
        name='block',
        target=wait_for_event,
        args=(e,),
    )
    w1.start()

    w2 = multiprocessing.Process(
        name='nonblock',
        target=wait_for_event_timeout,
        args=(e, 2),
    )
```

```
    w2.start()

    print('main: waiting before calling Event.set()')
    time.sleep(3)
    e.set()
    print('main: event is set')
```

По истечении тайм-аута метод `wait()` возвращает управление, не генерируя ошибку. Ответственность за проверку состояния объекта события с помощью метода `is_set()` возлагается на вызывающий код.

```
$ python3 -u multiprocessing_event.py

main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is set
wait_for_event: e.is_set()-> True
```

10.4.12. Управление доступом к ресурсам

Чтобы избежать конфликтов доступа к ресурсу, разделяемому несколькими процессами, можно использовать экземпляр класса `Lock`.

Листинг 10.66. `multiprocessing_lock.py`

```
import multiprocessing
import sys


def worker_with(lock, stream):
    with lock:
        stream.write('Lock acquired via with\n')


def worker_no_with(lock, stream):
    lock.acquire()
    try:
        stream.write('Lock acquired directly\n')
    finally:
        lock.release()


lock = multiprocessing.Lock()
w = multiprocessing.Process(
    target=worker_with,
    args=(lock, sys.stdout),
)
nw = multiprocessing.Process(
    target=worker_no_with,
    args=(lock, sys.stdout),
)
```

```
w.start()
nw.start()

w.join()
nw.join()
```

Если бы в этом примере оба процесса не синхронизировали свой доступ к потоку вывода с помощью блокировки, то их сообщения, выводимые на консоль, могли бы перемежаться.

```
$ python3 multiprocessing_lock.py

Lock acquired via with
Lock acquired directly
```

10.4.13. Синхронизация операций

Класс Condition, представляющий условие, позволяет синхронизировать отдельные ветви вычислений таким образом, чтобы одни из них выполнялись параллельно, тогда как другие — последовательно, даже если они принадлежат отдельным процессам.

Листинг 10.67. multiprocessing_condition.py

```
import multiprocessing
import time


def stage_1(cond):
    """Выполнить первый этап работы,
    а затем уведомить stage_2 для продолжения.
    """
    name = multiprocessing.current_process().name
    print('Starting', name)
    with cond:
        print('{} done and ready for stage 2'.format(name))
        cond.notify_all()


def stage_2(cond):
    """Дождаться условия, сообщающего, что stage_1 завершен."""
    name = multiprocessing.current_process().name
    print('Starting', name)
    with cond:
        cond.wait()
        print('{} running'.format(name))


if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='s1',
                                 target=stage_1,
                                 args=(condition,))
```

```
    s2_clients = [
        multiprocessing.Process(
            name='stage_2[{}]'.format(i),
            target=stage_2,
            args=(condition,),
        )
        for i in range(1, 3)
    ]

    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()
```

В этом примере второй этап вычислений выполняется параллельно обоими процессами, но только после того, как выполнен первый этап.

```
$ python3 multiprocessing_condition.py

Starting s1
s1 done and ready for stage 2
Starting stage_2[2]
stage_2[2] running
Starting stage_2[1]
stage_2[1] running
```

10.4.14. Контроль одновременного доступа к ресурсам

Иногда полезно разрешить нескольким рабочим процессам одновременный доступ к одному и тому же ресурсу, но при этом ограничить общее количество процессов, которым предоставляется такая возможность. В качестве примера можно привести пул, поддерживающий фиксированное количество соединений, или сетевое приложение, поддерживающее фиксированное количество одновременных загрузок. Одним из способов управления подобными соединениями являются семафоры, представляемые классом `Semaphore`.

Листинг 10.68. `multiprocessing_semaphore.py`

```
import random
import multiprocessing
import time


class ActivePool:

    def __init__(self):
        super(ActivePool, self).__init__()
        self.mgr = multiprocessing.Manager()
```

```
        self.active = self.mgr.list()
        self.lock = multiprocessing.Lock()

    def makeActive(self, name):
        with self.lock:
            self.active.append(name)

    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)

    def __str__(self):
        with self.lock:
            return str(self.active)


def worker(s, pool):
    name = multiprocessing.current_process().name
    with s:
        pool.makeActive(name)
        print('Activating {} now running {}'.format(
            name, pool))
        time.sleep(random.random())
        pool.makeInactive(name)


if __name__ == '__main__':
    pool = ActivePool()
    s = multiprocessing.Semaphore(3)
    jobs = [
        multiprocessing.Process(
            target=worker,
            name=str(i),
            args=(s, pool),
        )
        for i in range(10)
    ]

    for j in jobs:
        j.start()

    while True:
        alive = 0
        for j in jobs:
            if j.is_alive():
                alive += 1
                j.join(timeout=0.1)
                print('Now running {}'.format(pool))
        if alive == 0:
            # все задачи выполнены
            break
```

В этом примере в качестве удобного средства, позволяющего отслеживать процессы, выполняющиеся в заданный момент времени, используется класс `ActivePool`. Вероятно, при работе с реальным пулом такие ресурсы, как соединения, распределялись бы между активными процессами и возвращались в пул по завершении выполнения задачи. В данном случае пул используется всего лишь для хранения имен активных процессов и демонстрации того факта, что только три процесса выполняются одновременно.

```
$ python3 -u multiprocessing_semaphore.py

Activating 0 now running ['0', '1', '2']
Activating 1 now running ['0', '1', '2']
Activating 2 now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Activating 3 now running ['0', '1', '3']
Activating 4 now running ['1', '3', '4']
Activating 6 now running ['1', '4', '6']
Now running ['1', '4', '6']
Now running ['1', '4', '6']
Activating 5 now running ['1', '4', '5']
Now running ['1', '4', '5']
Now running ['1', '4', '5']
Now running ['1', '4', '5']
Activating 8 now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Activating 7 now running ['5', '8', '7']
Now running ['5', '8', '7']
Activating 9 now running ['8', '7', '9']
Now running ['8', '7', '9']
Now running ['8', '9']
Now running ['8', '9']
Now running ['9']
Now running ['9']
Now running ['9']
Now running ['9']
Now running []
```

10.4.15. Управление разделяемым состоянием

В предыдущем примере список активных процессов поддерживался централизованно в экземпляре `ActivePool` объектом списка специального типа, предоставляемым классом `Manager`. Этот класс отвечает за координацию информации о разделяемом состоянии между всеми своими пользователями.

Листинг 10.69. `multiprocessing_manager_dict.py`

```
import multiprocessing
import pprint


def worker(d, key, value):
    d[key] = value


if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    d = mgr.dict()
    jobs = [
        multiprocessing.Process(
            target=worker,
            args=(d, i, i * 2),
        )
        for i in range(10)
    ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print('Results:', d)
```

Поскольку этот список создается посредством менеджера, он разделяется всеми процессами, а его обновления видимы в каждом из них. Поддерживаются также словари.

```
$ python3 multiprocessing_manager_dict.py

Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14,
8: 16, 9: 18}
```

10.4.16. Разделяемые пространства имен

Кроме словарей и списков класс `Manager` позволяет создавать разделяемые пространства имен.

Листинг 10.70. `multiprocessing_namespaces.py`

```
import multiprocessing


def producer(ns, event):
    ns.value = 'This is the value'
    event.set()


def consumer(ns, event):
    try:
        print('Before event: {}'.format(ns.value))
```

```
    except Exception as err:
        print('Before event, error:', str(err))
    event.wait()
    print('After event:', ns.value)


if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    event = multiprocessing.Event()
    p = multiprocessing.Process(
        target=producer,
        args=(namespace, event),
    )
    c = multiprocessing.Process(
        target=consumer,
        args=(namespace, event),
    )

    c.start()
    p.start()

    c.join()
    p.join()
```

Любое именованное значение, добавляемое в пространство имен, становится видимым для всех клиентов, получающих экземпляр Namespace.

```
$ python3 multiprocessing_namespaces.py

Before event, error: 'Namespace' object has no attribute 'value'
After event: This is the value
```

Обновление содержимого изменяемых значений в пространстве имен не распространяется автоматически, как показано в следующем примере.

Листинг 10.71. multiprocessing_namespaces_mutable.py

```
import multiprocessing


def producer(ns, event):
    # НЕ ОБНОВЛЯТЬ ГЛОБАЛЬНОЕ ЗНАЧЕНИЕ!
    ns.my_list.append('This is the value')
    event.set()


def consumer(ns, event):
    print('Before event:', ns.my_list)
    event.wait()
    print('After event :', ns.my_list)


if __name__ == '__main__':
```

```
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    namespace.my_list = []

    event = multiprocessing.Event()
    p = multiprocessing.Process(
        target=producer,
        args=(namespace, event),
    )
    c = multiprocessing.Process(
        target=consumer,
        args=(namespace, event),
    )

    c.start()
    p.start()

    c.join()
    p.join()
```

Чтобы список обновился, его следует вновь присоединить к объекту пространства имен.

```
$ python3 multiprocessing_namespaces_mutable.py

Before event: []
After event : []
```

10.4.17. Пулы процессов

Класс `Pool` можно использовать для управления фиксированным количеством рабочих процессов в тех простых случаях, когда работу можно разделить на независимые части, распределяемые между процессами. Значения, возвращаемые отдельными задачами, объединяются и возвращаются в виде списка. В число аргументов конструктора `Pool` входят количество процессов и функция, подлежащая выполнению при запуске процесса отдельной задачи (однократно вызывается каждым дочерним процессом).

Листинг 10.72. `multiprocessing_pool.py`

```
import multiprocessing


def do_calculation(data):
    return data * 2


def start_process():
    print('Starting', multiprocessing.current_process().name)


if __name__ == '__main__':
```

```
    inputs = list(range(10))
    print('Input   :', inputs)

    builtin_outputs = map(do_calculation, inputs)
    print('Built-in:', builtin_outputs)

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(
        processes=pool_size,
        initializer=start_process,
    )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close()  # больше нет задач
    pool.join()  # обернуть текущие задачи

    print('Pool    :', pool_outputs)
```

Результаты, получаемые с помощью метода `map()`, функционально эквивалентны результатам, получаемым с помощью встроенной функции `map()`, за исключением того, что отдельные задачи выполняются параллельно. Поскольку пул обеспечивает параллельную обработку всех своих входных данных, можно синхронизировать основной процесс с процессами задач с помощью методов `close()` и `join()`, тем самым гарантируя выполнение завершающих операций по освобождению неиспользуемых ресурсов.

```
$ python3 multiprocessing_pool.py

Input : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Starting ForkPoolWorker-3
Starting ForkPoolWorker-4
Starting ForkPoolWorker-5
Starting ForkPoolWorker-6
Starting ForkPoolWorker-1
Starting ForkPoolWorker-7
Starting ForkPoolWorker-2
Starting ForkPoolWorker-8
Pool : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

По умолчанию экземпляр `Pool` создает фиксированное количество рабочих процессов и передает им задачи до тех пор, пока все они не будут исчерпаны. С помощью параметра `maxtasksperchild` можно указать максимальное количество задач, передаваемых одному процессу, после чего он будет перезапущен, что позволяет предотвратить завладение чрезмерно большим количеством ресурсов длительно выполняющимися процессами.

Листинг 10.73. `multiprocessing_pool_maxtasksperchild.py`

```
import multiprocessing


def do_calculation(data):
    return data * 2
```

```
def start_process():
    print('Starting', multiprocessing.current_process().name)


if __name__ == '__main__':
    inputs = list(range(10))
    print('Input   :', inputs)

    builtin_outputs = map(do_calculation, inputs)
    print('Built-in:', builtin_outputs)

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(
        processes=pool_size,
        initializer=start_process,
        maxtasksperchild=2,
    )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close()  # больше нет задач
    pool.join()  # обернуть текущие задачи

    print('Pool    :', pool_outputs)
```

Пул перезапускает рабочие процессы после выполнения ими отведенного количества задач, даже если все задачи уже исчерпаны. Как следует из приведенного ниже вывода, в данном примере создаются восемь рабочих процессов, хотя имеется только 10 задач и каждый рабочий процесс способен выполнить две задачи за один раз.

```
$ python3 multiprocessing_pool_maxtasksperchild.py

Input : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Starting ForkPoolWorker-1
Starting ForkPoolWorker-2
Starting ForkPoolWorker-4
Starting ForkPoolWorker-5
Starting ForkPoolWorker-6
Starting ForkPoolWorker-3
Starting ForkPoolWorker-7
Starting ForkPoolWorker-8
Pool : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

10.4.18. Реализация MapReduce

Для создания простой реализации модели распределенных вычислений MapReduce в случае одиночного сервера можно использовать класс `Pool`. Несмотря на то что такой подход не в состоянии обеспечить все преимущества распределенной обработки, он позволяет продемонстрировать, как легко решаются некоторые задачи, если их удается разбить на распределяемые части.

В системе, основанной на библиотеке MapReduce, входные данные подвергаются предварительной обработке, в ходе которой они разделяются на порции,

обрабатываемые отдельными экземплярами рабочих объектов. Каждая такая порция входных данных *отображается* (map) на промежуточное состояние с использованием простого преобразования. Затем промежуточные данные группируются на основе ключей таким образом, чтобы все родственные значения хранились вместе. Наконец, выполняется *свертка* (reduce) сгруппированных данных в результирующий набор.

Листинг 10.74. `multiprocessing_mapreduce.py`

```
import collections
import itertools
import multiprocessing


class SimpleMapReduce:

    def __init__(self, map_func, reduce_func, num_workers=None):
        """
        map_func

          Функция для отображения входных данных на промежуточные.
          Получает один аргумент в качестве входного значения и
          возвращает кортеж с ключом и значением для свертки.

        reduce_func

          Функция для свертки сгруппированной версии промежуточных
          данных в финальный вывод. В качестве аргумента получает
          ключ, созданный map_func, и последовательность значений,
          связанных с этим ключом.

        num_workers

          Число создаваемых рабочих процессов в пуле. По умолчанию
          равно числу процессоров текущего хоста.
        """
        self.map_func = map_func
        self.reduce_func = reduce_func
        self.pool = multiprocessing.Pool(num_workers)

    def partition(self, mapped_values):
        """Организует отображаемые значения по ключам.
        Возвращает неотсортированную последовательность кортежей
        с ключом и последовательностью значений.
        """
        partitioned_data = collections.defaultdict(list)
        for key, value in mapped_values:
            partitioned_data[key].append(value)
        return partitioned_data.items()

    def __call__(self, inputs, chunksize=1):
        """Обработать входные значения через предоставленные
        функции отображения и свертки.
```

```
        inputs
          Итерируемый объект, содержащий входные данные
          для обработки.

        chunksize=1
          Фрагмент входных данных для передачи каждому рабочему
          процессу. Это можно использовать для настройки
          производительности во время фазы отображения.
        """
        map_responses = self.pool.map(
            self.map_func,
            inputs,
            chunksize=chunksize,
        )
        partitioned_data = self.partition(
            itertools.chain(*map_responses)
        )
        reduced_values = self.pool.map(
            self.reduce_func,
            partitioned_data,
        )
        return reduced_values
```

В следующем примере класс `SimpleMapReduce` используется для подсчета слов в файлах на языке разметки reStructuredText (расширение *.rst*), содержащих текст данного раздела, с учетом того, что некоторые элементы разметки игнорируются.

Листинг 10.75. `multiprocessing_wordcount.py`

```
import multiprocessing
import string

from multiprocessing_mapreduce import SimpleMapReduce


def file_to_words(filename):
    """Прочитать файл и вернуть последовательность значений
    (число вхождений слов).
    """
    STOP_WORDS = set([
        'a', 'an', 'and', 'are', 'as', 'be', 'by', 'for', 'if',
        'in', 'is', 'it', 'of', 'or', 'py', 'rst', 'that', 'the',
        'to', 'with',
    ])
    TR = str.maketrans({
        p: ' '
        for p in string.punctuation
    })

    print('{} reading {}'.format(
        multiprocessing.current_process().name, filename))
    output = []

    with open(filename, 'rt') as f:
```

```
        for line in f:
            # Пропуск строк комментариев
            if line.lstrip().startswith('..'):
                continue
            line = line.translate(TR)  # отсечение знаков пунктуации
            for word in line.split():
                word = word.lower()
                if word.isalpha() and word not in STOP_WORDS:
                    output.append((word, 1))
    return output


def count_words(item):
    """Преобразовать сгруппированные данные для слова
    в кортеж, содержащий слово и число вхождений.
    """
    word, occurences = item
    return (word, sum(occurences))


if __name__ == '__main__':
    import operator
    import glob

    input_files = glob.glob('*.rst')

    mapper = SimpleMapReduce(file_to_words, count_words)
    word_counts = mapper(input_files)
    word_counts.sort(key=operator.itemgetter(1))
    word_counts.reverse()

    print('\nTOP 20 WORDS BY FREQUENCY\n')
    top20 = word_counts[:20]
    longest = max(len(word) for word, count in top20)
    for word, count in top20:
        print('{word:<{len}}: {count:5}'.format(
            len=longest + 1,
            word=word,
            count=count)
        )
```

Функция `file_to_words()` преобразует каждый входной файл в последовательность кортежей, содержащих слово и число `1` (представляющее одиночное вхождение). Данные группируются с помощью метода `partition()` с использованием слова в качестве ключа, поэтому результирующая структура состоит из ключа и последовательности значений `1`, представляющих вхождение слова. Сгруппированные данные преобразуются в набор кортежей, содержащих слово и суммарное количество вхождений этого слова, подсчитанное с помощью функции `count_words()` на этапе свертки результатов.

```
$ python3 -u multiprocessing_wordcount.py

ForkPoolWorker-1 reading basics.rst
```

```
ForkPoolWorker-2 reading communication.rst
ForkPoolWorker-3 reading index.rst
ForkPoolWorker-4 reading mapreduce.rst

TOP 20 WORDS BY FREQUENCY

process          :    83
running          :    45
multiprocessing  :    44
worker           :    40
starting         :    37
now              :    35
after            :    34
processes        :    31
start            :    29
header           :    27
pymotw           :    27
caption          :    27
end              :    27
daemon           :    22
can              :    22
exiting          :    21
forkpoolworker   :    21
consumer         :    20
main             :    18
event            :    16
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `multiprocessing`[7].
- `threading` (см. раздел 10.3). Высокоуровневый API для работы с потоками.
- Википедия: *MapReduce*[8]. Обзор модели распределенных вычислений MapReduce.
- *MapReduce. Simplified Data Processing on Large Clusters*[9]. Статья и презентация по теме MapReduce, подготовленные Google Labs.
- `operator` (см. раздел 3.3). Функциональный интерфейс для встроенных операторов (например, оператора `itemgetter`).

10.5. `asyncio`: асинхронные операции ввода-вывода, цикл событий и инструменты параллелизма

Модуль `asyncio` предоставляет инструменты для создания приложений, основанных на выполнении параллельных вычислений с использованием сопрограмм. В то время как модуль `threading` (см. раздел 10.3) реализует параллелизм

[7] https://docs.python.org/3.5/library/multiprocessing.html
[8] https://ru.wikipedia.org/wiki/MapReduce
[9] http://research.google.com/archive/mapreduce.html

вычислений на основе потоков выполнения, а модуль `multiprocessing` (см. раздел 10.4) — на основе использования системных процессов, модуль `asyncio` использует подход на основе единственного потока и единственного процесса, в котором отдельные части приложения кооперируются для явного переключения задач в наиболее подходящие для этого моменты времени. Чаще всего такой контекст переключения возникает тогда, когда при иной организации работы программы она блокировалась бы, ожидая завершения операций чтения или записи данных. Однако модуль `asyncio` также позволяет откладывать выполнение кода на определенный будущий момент времени, чтобы одна сопрограмма могла ожидать, пока выполнится другая, завершится обработка системных сигналов или распознаются другие события, которые могут стать причиной того, чтобы приложение изменило вычисления, выполняемые в данный момент.

10.5.1. Принципы асинхронного параллелизма

Большинство программ, в которых используются другие модели параллельных вычислений, следует линейному стилю и основываются на соответствующем изменении контекста за счет привлечения базовых средств управления потоками и процессами, предлагаемых средой времени выполнения или операционной системой. В случае приложений, основанных на использовании модуля `asyncio`, требуется, чтобы код приложения явно обрабатывал изменения контекста, используя для этого методы, которые корректно учитывают несколько взаимосвязанных понятий.

В предлагаемом модулем `asyncio` фреймворке центральное место занимает *цикл событий* — объект первого класса, ответственный за эффективную обработку событий ввода-вывода и изменение контекста приложения. Предоставляются несколько реализаций цикла событий, что обеспечивает наиболее эффективное использование возможностей конкретной операционной системы. В то время как обычно автоматически выбирается вариант цикла, предусмотренный по умолчанию, существует возможность выбора определенной реализации цикла событий самим приложением. Например, такой вариант может быть полезным при работе под управлением Windows, где некоторые классы добавляют поддержку внешних процессов способом, который может компенсировать неэффективность средств сетевого ввода-вывода.

Приложение взаимодействует с циклом событий явным образом, регистрируя подлежащий выполнению код и позволяя циклу событий выполнять необходимые вызовы в коде приложения, если доступны ресурсы. Например, сетевой сервер может открыть сокеты, а затем зарегистрировать их для получения уведомлений о наступлении событий, представляющих интерес. Цикл событий может известить сервер об установлении нового соединения или доступности новых данных, ожидающих чтения. Ожидается, что код приложения должен вновь уступать управление, если в данном контексте для него уже нет работы. Например, если в сокете больше нет данных, которые следует прочитать, то сервер должен вновь вернуть управление циклу событий.

Механизм возврата управления циклу событий основан на использовании *сопрограмм* Python — специальных функций, уступающих управление вызвавшему их коду без потери своего состояния. Сопрограммы очень похожи на функции-генераторы. В действительности в версиях Python, предшествующих версии 3.5,

генераторы позволяют реализовать сопрограммы без их поддержки со стороны платформы. Кроме того, модуль asyncio предоставляет слой абстрагирования каналов и протоколов передачи данных на основе классов, предназначенный для написания кода, в котором используются функции обратного вызова вместо сопрограмм. В обеих моделях, классической и основанной на сопрограммах, явное изменение контекста посредством повторного вхождения в цикл событий заменяет его неявное изменение в рамках реализации многопоточной модели Python.

Фьючерс — это структура данных, представляющая результат работы, которая еще не завершена. Цикл событий может отслеживать переход объекта Future в состояние "выполнено", тем самым предоставляя возможность одной части приложения ожидать, пока другая его часть завершит работу. Кроме фьючерсов модуль asyncio включает другие примитивы параллелизма, такие как блокировки и семафоры.

Task — это подкласс Future, которому известно, как обернуть сопрограмму и управлять ее выполнением. Цикл событий планирует выполнение задач на те моменты времени, когда необходимые им ресурсы становятся доступными, а производимые ими результаты могут быть использованы другими сопрограммами.

10.5.2. Организация кооперативной многозадачности с помощью сопрограмм

Сопрограмма — это конструкция языка, предназначенная для параллельного выполнения операций. При вызове функции сопрограммы создается объект сопрограммы, и вызывающий код может выполнить код этой функции, используя метод send() объекта сопрограммы. Сопрограмма может приостановить выполнение, используя ключевое слово await совместно с именем другой сопрограммы. На протяжении такой паузы состояние сопрограммы сохраняется, что позволяет ей при пробуждении продолжить выполнение с той точки, в которой оно было прервано.

10.5.2.1. Запуск сопрограммы

Цикл событий модуля asyncio может запустить сопрограмму различными способами. Самый простой подход заключается в использовании метода run_until_complete() с непосредственной передачей ему объекта сопрограммы.

Листинг 10.76. asyncio_coroutine.py

```
import asyncio


async def coroutine():
    print('in coroutine')


event_loop = asyncio.get_event_loop()
try:
    print('starting coroutine')
    coro = coroutine()
    print('entering event loop')
    event_loop.run_until_complete(coro)
```

```
finally:
    print('closing event loop')
    event_loop.close()
```

Прежде всего необходимо получить ссылку на цикл событий. Для этого можно либо воспользоваться циклом, предлагаемым по умолчанию, либо создать специфический экземпляр цикла. В данном примере используется цикл, заданный по умолчанию. Метод `run_until_complete()` запускает цикл с объектом сопрограммы и завершает его, когда сопрограмма возвращает управление.

```
$ python3 asyncio_coroutine.py

starting coroutine
entering event loop
in coroutine
closing event loop
```

10.5.2.2. Возврат значений из сопрограмм

Значение, возвращаемое сопрограммой, передается коду, который ее запустил и ожидает ее завершения.

Листинг 10.77. `asyncio_coroutine_return.py`

```
import asyncio


async def coroutine():
    print('in coroutine')
    return 'result'


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(
        coroutine()
    )
    print('it returned: {!r}'.format(return_value))
finally:
    event_loop.close()
```

В данном случае метод `run_until_complete()` возвращает результат выполнения сопрограммы, завершения которой он ожидает.

```
$ python3 asyncio_coroutine_return.py

in coroutine
it returned: 'result'
```

10.5.2.3. Цепочки сопрограмм

Одна сопрограмма может запустить другую и ожидать ее результатов, что упрощает разбиение задачи на повторно используемые части. Следующий при-

мер имеет две фазы, которые должны быть выполнены по очереди, но могут выполняться параллельно с другими операциями.

Листинг 10.78. asyncio_coroutine_chain.py

```
import asyncio


async def outer():
    print('in outer')
    print('waiting for result1')
    result1 = await phase1()
    print('waiting for result2')
    result2 = await phase2(result1)
    return (result1, result2)


async def phase1():
    print('in phase1')
    return 'result1'


async def phase2(arg):
    print('in phase2')
    return 'result2 derived from {}'.format(arg)


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(outer())
    print('return value: {!r}'.format(return_value))
finally:
    event_loop.close()
```

Вместо добавления в цикл новых сопрограмм можно использовать ключевое слово await. Поскольку поток выполнения уже находится в теле сопрограммы, управляемой циклом, нет нужды сообщать циклу о необходимости управления новыми сопрограммами.

```
$ python3 asyncio_coroutine_chain.py

in outer
waiting for result1
in phase1
waiting for result2
in phase2
return value: ('result1', 'result2 derived from result1')
```

10.5.2.4. Генераторы вместо сопрограмм

Функции сопрограмм являются ключевым элементом модуля asyncio. Они предоставляют языковую конструкцию, позволяющую останавливать выполнение части программы, сохранять состояние этого вызова и осуществлять повтор-

ное вхождение в это же состояние в более поздний момент времени. Для фреймворка параллелизма важна возможность выполнения любого из этих действий.

Возможность определять сопрограммы с помощью инструкции `async def` и уступать управление с помощью ключевого слова `await`, как это делалось в приведенных выше примерах, появилась в версии Python 3.5. В предыдущих версиях Python 3 тот же эффект достигается за счет использования функций-генераторов, обернутых декоратором `asyncio.coroutine()`, и инструкции `yield from`.

Листинг 10.79. `asyncio_generator.py`

```
import asyncio


@asyncio.coroutine
def outer():
    print('in outer')
    print('waiting for result1')
    result1 = yield from phase1()
    print('waiting for result2')
    result2 = yield from phase2(result1)
    return (result1, result2)


@asyncio.coroutine
def phase1():
    print('in phase1')
    return 'result1'


@asyncio.coroutine
def phase2(arg):
    print('in phase2')
    return 'result2 derived from {}'.format(arg)


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(outer())
    print('return value: {!r}'.format(return_value))
finally:
    event_loop.close()
```

В этом примере работа сценария `asyncio_coroutine_chain.py` воспроизведена с использованием функций-генераторов вместо сопрограмм.

```
$ python3 asyncio_generator.py

in outer
waiting for result1
in phase1
waiting for result2
in phase2
return value: ('result1', 'result2 derived from result1')
```

10.5.3. Планирование вызовов обычных функций

Кроме управления сопрограммами и обратными вызовами функций ввода-вывода цикл событий модуля `asyncio` может планировать вызовы функций, основываясь на значении таймера, поддерживаемого в цикле.

10.5.3.1. Планирование обратного вызова "на ближайшее время"

Если точное время выполнения обратных вызовов не имеет значения, метод `call_soon()` позволяет запланировать вызов на следующую итерацию цикла. При запуске функции обратного вызова ей передаются любые дополнительные позиционные аргументы, указанные вслед за функцией обратного вызова при выполнении метода `call_soon()`. Передать функции обратного вызова ключевые аргументы можно с помощью функции `partial()` из модуля `functools` (см. раздел 3.1).

Листинг 10.80. `asyncio_call_soon.py`

```
import asyncio
import functools


def callback(arg, *, kwarg='default'):
    print('callback invoked with {} and {}'.format(arg, kwarg))


async def main(loop):
    print('registering callbacks')
    loop.call_soon(callback, 1)
    wrapped = functools.partial(callback, kwarg='not default')
    loop.call_soon(wrapped, 2)

    await asyncio.sleep(0.1)


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    event_loop.run_until_complete(main(event_loop))
finally:
    print('closing event loop')
    event_loop.close()
```

Функции обратного вызова запускаются в том порядке, в каком они были запланированы.

```
$ python3 asyncio_call_soon.py

entering event loop
registering callbacks
callback invoked with 1 and default
callback invoked with 2 and not default
closing event loop
```

10.5.3.2. Планирование обратного вызова с задержкой во времени

Чтобы отложить выполнение функции обратного вызова на более поздний момент времени, следует использовать метод `call_later()`. Его первым аргументом является длительность задержки в секундах, вторым — функция обратного вызова.

Листинг 10.81. `asyncio_call_later.py`

```
import asyncio


def callback(n):
    print('callback {} invoked'.format(n))


async def main(loop):
    print('registering callbacks')
    loop.call_later(0.2, callback, 1)
    loop.call_later(0.1, callback, 2)
    loop.call_soon(callback, 3)

    await asyncio.sleep(0.4)


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    event_loop.run_until_complete(main(event_loop))
finally:
    print('closing event loop')
    event_loop.close()
```

В этом примере выполнение той же функции обратного вызова запланировано на несколько различных моментов времени с разными аргументами. Вызов, запланированный с помощью метода `call_soon()`, которому был передан аргумент 3, выполняется раньше других, тем самым подтверждая тот факт, что планирование вызова "на ближайшее время" обычно означает минимальную задержку.

```
$ python3 asyncio_call_later.py

entering event loop
registering callbacks
callback 3 invoked
callback 2 invoked
callback 1 invoked
closing event loop
```

10.5.3.3. Планирование обратного вызова на определенное время

Также существует возможность запланировать вызов на определенное время. Используемый для этого цикл основывается на монотонных часах, а не на часах истекшего времени, что исключает отнесение момента времен "сейчас" к про-

шлому моменту времени. Чтобы выбрать время для запланированного обратного вызова, необходимо вести отсчет от внутреннего состояния этих часов, используя метод `time()` цикла.

Листинг 10.82. `asyncio_call_at.py`

```
import asyncio
import time


def callback(n, loop):
    print('callback {} invoked at {}'.format(n, loop.time()))


async def main(loop):
    now = loop.time()
    print('clock time: {}'.format(time.time()))
    print('loop  time: {}'.format(now))

    print('registering callbacks')
    loop.call_at(now + 0.2, callback, 1, loop)
    loop.call_at(now + 0.1, callback, 2, loop)
    loop.call_soon(callback, 3, loop)

    await asyncio.sleep(1)


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    event_loop.run_until_complete(main(event_loop))
finally:
    print('closing event loop')
    event_loop.close()
```

Обратите внимание на то, что время, соответствующее часам цикла, не совпадает со значением, возвращаемым функцией `time.time()`.

```
$ python3 asyncio_call_at.py

entering event loop
clock time: 1479050248.66192
loop time: 1008846.13856885
registering callbacks
callback 3 invoked at 1008846.13867956
callback 2 invoked at 1008846.239931555
callback 1 invoked at 1008846.343480996
closing event loop
```

10.5.4. Асинхронное получение результатов

Экземпляр `Future` представляет результат еще не завершенной работы. Цикл событий может отслеживать переход экземпляра `Future` в состояние "выполне-

но" (done), тем самым предоставляя возможность одной части приложения ожидать, пока другая его часть завершит работу.

10.5.4.1. Ожидание завершения задания экземпляром `Future`

Экземпляр `Future` (фьючерс) действует подобно сопрограмме, поэтому любые приемы, используемые для ожидания завершения сопрограммы, также применимы к фьючерсам. В следующем примере объект фьючерса передается методу `run_until_complete()` цикла.

Листинг 10.83. `asyncio_future_event_loop.py`

```
import asyncio


def mark_done(future, result):
    print('setting future result to {!r}'.format(result))
    future.set_result(result)


event_loop = asyncio.get_event_loop()
try:
    all_done = asyncio.Future()

    print('scheduling mark_done')
    event_loop.call_soon(mark_done, all_done, 'the result')

    print('entering event loop')
    result = event_loop.run_until_complete(all_done)
    print('returned result: {!r}'.format(result))
finally:
    print('closing event loop')
    event_loop.close()

print('future result: {!r}'.format(all_done.result()))
```

Экземпляр `Future` переходит в состояние "выполнено", когда вызывается метод `set_result()`, и сохраняет переданный методу результат для последующего извлечения.

```
$ python3 asyncio_future_event_loop.py

scheduling mark_done
entering event loop
setting future result to 'the result'
returned result: 'the result'
closing event loop
future result: 'the result'
```

Экземпляр `Future` также можно использовать вместе с ключевым словом `await`, как показано в следующем примере.

Листинг 10.84. `asyncio_future_await.py`

```
import asyncio


def mark_done(future, result):
    print('setting future result to {!r}'.format(result))
    future.set_result(result)


async def main(loop):
    all_done = asyncio.Future()

    print('scheduling mark_done')
    loop.call_soon(mark_done, all_done, 'the result')

    result = await all_done
    print('returned result: {!r}'.format(result))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

Результат, хранящийся в экземпляре `Future`, возвращается с помощью ключевого слова `await`, поэтому во многих случаях один и тот же код может работать как с обычной сопрограммой, так и с экземпляром `Future`.

```
$ python3 asyncio_future_await.py

scheduling mark_done
setting future result to 'the result'
returned result: 'the result'
```

10.5.4.2. Использование объектов `Future` с функциями обратного вызова

Экземпляры `Future` могут не только работать как сопрограммы, но и запускать функции обратного вызова при завершении работы. Функции обратного вызова выполняются в том порядке, в каком они регистрировались.

Листинг 10.85. `asyncio_future_callback.py`

```
import asyncio
import functools


def callback(future, n):
    print('{}: future done: {}'.format(n, future.result()))
```

```
async def register_callbacks(all_done):
    print('registering callbacks on future')
    all_done.add_done_callback(functools.partial(callback, n=1))
    all_done.add_done_callback(functools.partial(callback, n=2))


async def main(all_done):
    await register_callbacks(all_done)
    print('setting result of future')
    all_done.set_result('the result')


event_loop = asyncio.get_event_loop()
try:
    all_done = asyncio.Future()
    event_loop.run_until_complete(main(all_done))
finally:
    event_loop.close()
```

Функция обратного вызова должна получать один аргумент: экземпляр `Future`. Если требуется передать дополнительные аргументы, используйте функцию `functools.partial()` для создания обертки.

```
$ python3 asyncio_future_callback.py

registering callbacks on future
setting result of future
1: future done: the result
2: future done: the result
```

10.5.5. Параллельное выполнение задач

Задачи – один из основных способов взаимодействия с циклом событий. Задачи обертывают сопрограммы и отслеживают момент их завершения. Поскольку задачи являются подклассами `Future`, другие сопрограммы могут ожидать их завершения, и каждая задача имеет результат, который может быть извлечен после того, как задача завершится.

10.5.5.1. Запуск задачи

Чтобы запустить задачу, следует создать экземпляр `Task`, используя метод `create_task()`. Результирующая задача будет выполняться как часть параллельных операций, управляемых циклом событий, до тех пор пока выполняется цикл и сопрограмма не возвращает управление.

Листинг 10.86. `asyncio_create_task.py`

```
import asyncio


async def task_func():
    print('in task_func')
    return 'the result'
```

```
async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())
    print('waiting for {!r}'.format(task))
    return_value = await task
    print('task completed {!r}'.format(task))
    print('return value: {!r}'.format(return_value))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

В этом примере функция `main()` ожидает возврата результата задачей, после чего возвращает управление.

```
$ python3 asyncio_create_task.py

creating task
waiting for <Task pending coro=<task_func() running at
asyncio_create_task.py:12>>
in task_func
task completed <Task finished coro=<task_func() done, defined at
asyncio_create_task.py:12> result='the result'>
return value: 'the result'
```

10.5.5.2. Отмена выполнения задачи

Сохранив объект `Task`, возвращенный методом `create_task()`, можно отменить выполнение задачи до ее завершения.

Листинг 10.87. `asyncio_cancel_task.py`

```
import asyncio


async def task_func():
    print('in task_func')
    return 'the result'


async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())

    print('canceling task')
    task.cancel()

    print('canceled task {!r}'.format(task))
    try:
        await task
```

```
    except asyncio.CancelledError:
        print('caught error from canceled task')
    else:
        print('task result: {!r}'.format(task.result()))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

В этом примере выполнение предварительно созданной задачи отменяется до запуска цикла событий. Вследствие этого метод `run_until_complete()` возбуждает исключение `CancelledError`.

```
$ python3 asyncio_cancel_task.py

creating task
canceling task
canceled task <Task cancelling coro=<task_func() running at
asyncio_cancel_task.py:12>>
caught error from canceled task
```

Если задача отменяется в то время, когда она ожидает завершения другой параллельной операции, она извещается об этом возбуждением исключения `CancelledError` в точке ожидания.

Листинг 10.88. `asyncio_cancel_task2.py`

```
import asyncio


async def task_func():
    print('in task_func, sleeping')
    try:
        await asyncio.sleep(1)
    except asyncio.CancelledError:
        print('task_func was canceled')
        raise
    return 'the result'


def task_canceller(t):
    print('in task_canceller')
    t.cancel()
    print('canceled the task')


async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())
    loop.call_soon(task_canceller, task)
```

```
    try:
        await task
    except asyncio.CancelledError:
        print('main() also sees task as canceled')


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
```

Перехват исключения предоставляет возможность выполнить завершающие операции по освобождению ресурсов, если в этом есть необходимость.

```
$ python3 asyncio_cancel_task2.py

creating task
in task_func, sleeping
in task_canceller
canceled the task
task_func was canceled
main() also sees task as canceled
```

10.5.5.3. Создание задач из сопрограмм

Функция `ensure_future()` возвращает экземпляр `Task`, связанный с выполнением сопрограммы. Этот экземпляр можно передать другому коду, который будет ожидать его завершения, не зная, каким образом была сконструирована или вызвана сопрограмма.

Листинг 10.89. `asyncio_ensure_future.py`

```
import asyncio


async def wrapped():
    print('wrapped')
    return 'result'


async def inner(task):
    print('inner: starting')
    print('inner: waiting for {!r}'.format(task))
    result = await task
    print('inner: task returned {!r}'.format(result))


async def starter():
    print('starter: creating task')
    task = asyncio.ensure_future(wrapped())
    print('starter: waiting for inner')
    await inner(task)
    print('starter: inner returned')
```

```
event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    result = event_loop.run_until_complete(starter())
finally:
    event_loop.close()
```

Обратите внимание на то, что сопрограмма, переданная функции `ensure_future()`, не запустится до тех пор, пока где-нибудь не будет использовано ключевое слово `await`, разрешающее ее выполнение.

```
$ python3 asyncio_ensure_future.py

entering event loop
starter: creating task
starter: waiting for inner
inner: starting
inner: waiting for <Task pending coro=<wrapped() running at
asyncio_ensure_future.py:12>>
wrapped
inner: task returned 'result'
starter: inner returned
```

10.5.6. Сочетание сопрограмм с управляющими конструкциями

С линейным потоком управления между серией сопрограмм можно легко справиться с помощью встроенного ключевого слова `await`. Используя инструменты, предоставляемые модулем `asyncio`, можно создавать более сложные управляющие конструкции, которые позволяют одной сопрограмме дожидаться завершения нескольких других сопрограмм, выполняющихся параллельно.

10.5.6.1. Ожидание завершения нескольких сопрограмм

Во многих случаях полезно разбить одну операцию на несколько частей, выполняющихся по отдельности. Например, такой подход эффективен при загрузке данных из нескольких удаленных источников или опросе удаленных программных интерфейсов. В ситуациях, когда порядок выполнения не имеет значения, а количество операций может быть произвольным, можно использовать функцию `wait()` для приостановки одной программы до тех пор, пока не завершатся другие операции, выполняющиеся в фоновом режиме.

Листинг 10.90. `asyncio_wait.py`

```
import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    await asyncio.sleep(0.1 * i)
    print('done with phase {}'.format(i))
    return 'phase {} result'.format(i)
```

```
async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting for phases to complete')
    completed, pending = await asyncio.wait(phases)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

Функция `wait()` использует множество для хранения экземпляров `Task`, которые она создает, а это означает, что порядок запуска и завершения экземпляров непредсказуем. Возвращаемое значение функции `wait()` представляет собой кортеж, содержащий два набора: им соответствуют завершенные и выполняющиеся задачи.

```
$ python3 asyncio_wait.py
starting main
waiting for phases to complete
in phase 0
in phase 1
in phase 2
done with phase 0
done with phase 1
done with phase 2
results: ['phase 1 result', 'phase 0 result', 'phase 2 result']
```

Если функция `wait()` вызывается с аргументом `timeout`, то по истечении тайм-аута остаются только незавершенные операции.

Листинг 10.91. `asyncio_wait_timeout.py`

```
import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    try:
        await asyncio.sleep(0.1 * i)
    except asyncio.CancelledError:
        print('phase {} canceled'.format(i))
        raise
    else:
        print('done with phase {}'.format(i))
        return 'phase {} result'.format(i)
```

```
async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting 0.1 for phases to complete')
    completed, pending = await asyncio.wait(phases, timeout=0.1)
    print('{} completed and {} pending'.format(
        len(completed), len(pending),
    ))
    # Отменить оставшиеся задачи, чтобы они не сгенерировали
    # ошибки, если к моменту выхода еще не завершились
    if pending:
        print('canceling tasks')
        for t in pending:
            t.cancel()
    print('exiting main')


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

Остальные фоновые операции должны обрабатываться явным образом по нескольким причинам. Несмотря на то что выполнение незавершенных задач приостанавливается, когда функция `wait()` выполняет возврат, оно будет возобновлено, как только управление вернется циклу событий. Без дополнительного вызова `wait()` выходные результаты задач никуда не попадут — задачи будут выполняться и потреблять ресурсы, но от этого не будет никакого результата. Кроме того, модуль `asyncio` вырабатывает предупреждения в тех случаях, когда при выходе из программы имеются незавершенные задачи. Эти предупреждения могут выводиться на экран, чтобы пользователь увидел их. Поэтому лучше всего либо отменить остающиеся незавершенными фоновые операции, либо использовать функцию `wait()`, чтобы позволить им завершиться.

```
$ python3 asyncio_wait_timeout.py

starting main
waiting 0.1 for phases to complete
in phase 1
in phase 0
in phase 2
done with phase 0
1 completed and 2 pending
cancelling tasks
exiting main
phase 1 cancelled
phase 2 cancelled
```

10.5.6.2. Сбор результатов от сопрограмм

Если фоновые фазы определены корректно и имеют значение лишь их результаты, то для ожидания завершения нескольких операций целесообразно использовать функцию `gather()`.

Листинг 10.92. `asyncio_gather.py`

```
import asyncio


async def phase1():
    print('in phase1')
    await asyncio.sleep(2)
    print('done with phase1')
    return 'phase1 result'


async def phase2():
    print('in phase2')
    await asyncio.sleep(1)
    print('done with phase2')
    return 'phase2 result'


async def main():
    print('starting main')
    print('waiting for phases to complete')
    results = await asyncio.gather(
        phase1(),
        phase2(),
    )
    print('results: {!r}'.format(results))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main())
finally:
    event_loop.close()
```

Поскольку доступ к задачам, созданным с помощью функции `gather()`, не предоставляется, их нельзя отменить. Возвращаемым значением функции `gather()` является список результатов, представленных в том порядке, в каком ей передавались аргументы, независимо от того, какова была фактическая очередность завершения операций.

```
$ python3 asyncio_gather.py

starting main
waiting for phases to complete
in phase2
in phase1
```

```
done with phase2
done with phase1
results: ['phase1 result', 'phase2 result']
```

10.5.6.3. Обработка фоновых операций по мере их завершения

Функция `as_completed()` — это генератор, который управляет выполнением списка переданных ему программ и возвращает результаты по одному за раз по мере завершения выполняющихся сопрограмм. Как и функция `wait()`, функция `as_completed()` не гарантирует очередность завершения программ, но ждать, пока завершатся все фоновые операции, прежде чем предпринимать какие-либо другие действия, необязательно.

Листинг 10.93. `asyncio_as_completed.py`

```
import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    await asyncio.sleep(0.5 - (0.1 * i))
    print('done with phase {}'.format(i))
    return 'phase {} result'.format(i)


async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting for phases to complete')
    results = []
    for next_to_complete in asyncio.as_completed(phases):
        answer = await next_to_complete
        print('received answer {!r}'.format(answer))
        results.append(answer)
    print('results: {!r}'.format(results))
    return results


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

Этот пример начинается с запуска нескольких фоновых фаз, которые завершаются в порядке, обратном порядку их запуска. По исчерпании генератора цикл ожидает результаты работы сопрограммы, используя ключевое слово `await`.

```
$ python3 asyncio_as_completed.py

starting main
```

```
waiting for phases to complete
in phase 0
in phase 2
in phase 1
done with phase 2
received answer 'phase 2 result'
done with phase 1
received answer 'phase 1 result'
done with phase 0
received answer 'phase 0 result'
results: ['phase 2 result', 'phase 1 result', 'phase 0 result']
```

10.5.7. Примитивы синхронизации

Несмотря на то что приложения на основе модуля `asyncio` обычно выполняются в виде однопоточных процессов, они создаются как приложения, рассчитанные на параллельные вычисления. Каждая сопрограмма или задача может выполняться в недетерминированном порядке, в зависимости от задержек и прерываний со стороны устройств ввода-вывода и других внешних событий. С целью поддержания безопасности параллельных вычислений модуль `asyncio` включает реализации некоторых низкоуровневых примитивов, которые встречаются также в модулях `threading` (см. раздел 10.3) и `multiprocessing` (см. раздел 10.4).

10.5.7.1. Блокировки

Класс `Lock` можно использовать для защиты доступа к разделяемым ресурсам. Лишь владелец блокировки может использовать ресурс. Несколько одновременных попыток получить блокировку будут блокироваться, поэтому в каждый момент времени блокировка может иметь только одного владельца.

Листинг 10.94. `asyncio_lock.py`

```
import asyncio
import functools


def unlock(lock):
    print('callback releasing lock')
    lock.release()


async def coro1(lock):
    print('coro1 waiting for the lock')
    with await lock:
        print('coro1 acquired lock')
    print('coro1 released lock')


async def coro2(lock):
    print('coro2 waiting for the lock')
    await lock
    try:
        print('coro2 acquired lock')
```

```
    finally:
        print('coro2 released lock')
        lock.release()


async def main(loop):
    # Создать и получить разделяемую блокировку
    lock = asyncio.Lock()
    print('acquiring the lock before starting coroutines')
    await lock.acquire()
    print('lock acquired: {}'.format(lock.locked()))

    # Запланировать функцию обратного вызова для
    # отмены блокировки
    loop.call_later(0.1, functools.partial(unlock, lock))

    # Запустить сопрограммы, которые хотят использовать
    # блокировку
    print('waiting for coroutines')
    await asyncio.wait([coro1(lock), coro2(lock)]),


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

Блокировку можно активизировать непосредственно, используя ключевое слово `await` для ее получения и вызывая метод `release()`, когда работа выполнена, как в случае сопрограммы `coro2()` в данном примере. Кроме того, блокировки могут использоваться в качестве асинхронных менеджеров контекста вместе с ключевыми словами `await`, как в случае сопрограммы `coro1()`.

```
$ python3 asyncio_lock.py

acquiring the lock before starting coroutines
lock acquired: True
waiting for coroutines
coro1 waiting for the lock
coro2 waiting for the lock
callback releasing lock
coro1 acquired lock
coro1 released lock
coro2 acquired lock
coro2 released lock
```

10.5.7.2. События

Класс `asyncio.Event` основан на классе `threading.Event` и позволяет нескольким потребителям ожидать наступления какого-либо события без отслеживания конкретного значения, связанного с соответствующим уведомлением.

Листинг 10.95. asyncio_event.py

```
import asyncio
import functools


def set_event(event):
    print('setting event in callback')
    event.set()


async def coro1(event):
    print('coro1 waiting for event')
    await event.wait()
    print('coro1 triggered')


async def coro2(event):
    print('coro2 waiting for event')
    await event.wait()
    print('coro2 triggered')


async def main(loop):
    # Создать разделяемое событие
    event = asyncio.Event()
    print('event start state: {}'.format(event.is_set()))

    loop.call_later(
        0.1, functools.partial(set_event, event)
    )

    await asyncio.wait([coro1(event), coro2(event)])
    print('event end state: {}'.format(event.is_set()))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

Как и в случае класса `Lock`, сопрограммы `coro1()` и `coro2()` ожидают установки события. Отличие заключается в том, что выполнение обеих сопрограмм может начаться сразу же, как только событие изменит состояние, и они не должны становиться единоличными владельцами объекта события.

```
$ python3 asyncio_event.py

event start state: False
coro2 waiting for event
coro1 waiting for event
setting event in callback
```

```
coro2 triggered
coro1 triggered
event end state: True
```

10.5.7.3. Условия

Класс `Condition` работает аналогично классу `Event`, за исключением того, что вместо уведомления всех ожидающих сопрограмм можно ограничить количество тех из них, которые будут пробуждаться, передав соответствующий аргумент методу `notify()`.

Листинг 10.96. `asyncio_condition.py`

```
import asyncio


async def consumer(condition, n):
    with await condition:
        print('consumer {} is waiting'.format(n))
        await condition.wait()
        print('consumer {} triggered'.format(n))
    print('ending consumer {}'.format(n))


async def manipulate_condition(condition):
    print('starting manipulate_condition')

    # Пауза, позволяющая потребителям запуститься
    await asyncio.sleep(0.1)

    for i in range(1, 3):
        with await condition:
            print('notifying {} consumers'.format(i))
            condition.notify(n=i)
        await asyncio.sleep(0.1)

    with await condition:
        print('notifying remaining consumers')
        condition.notify_all()

    print('ending manipulate_condition')


async def main(loop):
    # Создать условие
    condition = asyncio.Condition()

    # Создать список задач, следящих за условием
    consumers = [
        consumer(condition, i)
        for i in range(5)
    ]
```

```
    # Запланировать задачу для манипулирования
    # переменной condition
    loop.create_task(manipulate_condition(condition))

    # Ждать завершения потребителей
    await asyncio.wait(consumers)


event_loop = asyncio.get_event_loop()
try:
    result = event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

В этом примере запускаются пять потребителей экземпляра `Condition`. Каждый из них использует метод `wait()` для ожидания уведомления о том, что он может продолжить выполнение. Сопрограмма `manipulate_condition()` уведомляет об этом одного потребителя, потом двух потребителей, а затем всех остальных потребителей.

```
$ python3 asyncio_condition.py

starting manipulate_condition
consumer 3 is waiting
consumer 1 is waiting
consumer 2 is waiting
consumer 0 is waiting
consumer 4 is waiting
notifying 1 consumers
consumer 3 triggered
ending consumer 3
notifying 2 consumers
consumer 1 triggered
ending consumer 1
consumer 2 triggered
ending consumer 2
notifying remaining consumers
ending manipulate_condition
consumer 0 triggered
ending consumer 0
consumer 4 triggered
ending consumer 4
```

10.5.7.4. Очереди

Класс `asyncio.Queue` предоставляет для сопрограмм структуру данных, организованную по принципу "первым пришел — первым ушел", которая работает во многом подобно структурам, предоставляемым классами `aqueue.Queue` для потоков и `multiprocessing.Queue` для процессов.

Листинг 10.97. asyncio_queue.py

```
import asyncio


async def consumer(n, q):
    print('consumer {}: starting'.format(n))
    while True:
        print('consumer {}: waiting for item'.format(n))
        item = await q.get()
        print('consumer {}: has item {}'.format(n, item))
        if item is None:
            # None - сигнал прекратить выполнение
            q.task_done()
            break
        else:
            await asyncio.sleep(0.01 * item)
            q.task_done()
    print('consumer {}: ending'.format(n))


async def producer(q, num_workers):
    print('producer: starting')
    # Добавить некоторое количество задач в очередь
    # для имитации выполнения работы
    for i in range(num_workers * 3):
        await q.put(i)
        print('producer: added task {} to the queue'.format(i))
    # Добавить в очередь значения None в качестве сигналов,
    # предписывающих потребителям прекратить выполнение
    print('producer: adding stop signals to the queue')
    for i in range(num_workers):
        await q.put(None)
    print('producer: waiting for queue to empty')
    await q.join()
    print('producer: ending')


async def main(loop, num_consumers):
    # Создать очередь фиксированного размера, чтобы производитель
    # блокировался до тех пор, пока потребители не извлекут
    # некоторое количество элементов
    q = asyncio.Queue(maxsize=num_consumers)

    # Запланировать задачи потребителей
    consumers = [
        loop.create_task(consumer(i, q))
        for i in range(num_consumers)
    ]

    # Запланировать задачи производителей
    prod = loop.create_task(producer(q, num_consumers))
```

```
    # Ждать завершения работы всех сопрограмм
    await asyncio.wait(consumers + [prod])


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop, 2))
finally:
    event_loop.close()
```

Операции добавления элементов с помощью метода `put()` и удаления с помощью метода `get()` выполняются как асинхронные, поскольку либо размер очереди может быть фиксированным (блокируется добавление в нее новых элементов), либо очередь может быть пустой (блокируется извлечение элемента).

```
$ python3 asyncio_queue.py

consumer 0: starting
consumer 0: waiting for item
consumer 1: starting
consumer 1: waiting for item
producer: starting
producer: added task 0 to the queue
producer: added task 1 to the queue
consumer 0: has item 0
consumer 1: has item 1
producer: added task 2 to the queue
producer: added task 3 to the queue
consumer 0: waiting for item
consumer 0: has item 2
producer: added task 4 to the queue
consumer 1: waiting for item
consumer 1: has item 3
producer: added task 5 to the queue
producer: adding stop signals to the queue
consumer 0: waiting for item
consumer 0: has item 4
consumer 1: waiting for item
consumer 1: has item 5
producer: waiting for queue to empty
consumer 0: waiting for item
consumer 0: has item None
consumer 0: ending
consumer 1: waiting for item
consumer 1: has item None
consumer 1: ending
producer: ending
```

10.5.8. Асинхронный ввод-вывод с использованием абстракций класса `Protocol`

До сих пор во всех примерах параллельные вычисления и операции ввода-вывода по возможности не смешивались, чтобы внимание было сосредоточено каждый раз только на одном из этих аспектов. Однако переключение контекстов, когда блокируются операции ввода-вывода, является одним из типичных вариантов применения модуля `asyncio`. Продолжая обсуждение введенных ранее понятий параллелизма, в этом разделе рассматриваются примеры двух программ, реализующих эхо-сервер и эхо-клиент, которые аналогичны примерам, приведенным при рассмотрении модулей `socket` (раздел 11.2) и `socketserver` (раздел 11.5). Клиент может соединиться с сервером, отправить ему некоторые данные, а затем получить те же данные в виде ответа. Каждый раз, когда инициируется операция ввода-вывода, выполняющийся код уступает управление циклу событий, позволяя другим задачам выполняться до тех пор, пока операции ввода-вывода не будут завершены.

10.5.8.1. Эхо-сервер

Код эхо-сервера начинается с импорта модулей `asyncio` и `logging` (раздел 14.80) и создания объекта цикла событий.

Листинг 10.98. `asyncio_echo_server_protocol.py`

```
import asyncio
import logging
import sys

SERVER_ADDRESS = ('localhost', 10000)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
    stream=sys.stderr,
)
log = logging.getLogger('main')

event_loop = asyncio.get_event_loop()
```

Далее определяется подкласс `asyncio.Protocol`, обеспечивающий взаимодействие с клиентом. Методы объекта протокола вызываются в ответ на события, связанные с сокетом сервера.

```
class EchoServer(asyncio.Protocol):
```

Подключение каждого нового клиента инициирует вызов метода `connection_made()`. Аргумент `transport` — это экземпляр `asyncio.Transport`, предоставляющий абстракцию для выполнения асинхронных операций ввода-вывода с использованием сокета. Различные типы протоколов связи предоставляют различные реализации объекта `transport` с одним и тем же API. Например, для работы с сокетами и для работы с каналами подпроцессов используются отдельные

транспортные классы. Адрес клиента доступен из объекта `transport` через метод `get_extra_info()`, специфический для каждой реализации.

```
    def connection_made(self, transport):
        self.transport = transport
        self.address = transport.get_extra_info('peername')
        self.log = logging.getLogger(
            'EchoServer_{}_{}'.format(*self.address)
        )
        self.log.debug('connection accepted')
```

После установления соединения, когда клиент отправил данные серверу, вызывается метод `data_received()` протокола, который передает поступившие данные для их обработки. Данные передаются в виде байтовой строки, за декодирование и надлежащую обработку которой отвечает приложение. В следующем фрагменте кода результаты протоколируются, а ответ немедленно отправляется обратно клиенту посредством вызова метода `transport.write()`.

```
    def data_received(self, data):
        self.log.debug('received {!r}'.format(data))
        self.transport.write(data)
        self.log.debug('sent {!r}'.format(data))
```

Некоторые объекты `transport` поддерживают специальные метки конца файла (EOF). Когда встречается метка EOF, вызывается метод `eof_received()`. В этой реализации метка EOF отправляется обратно клиенту в качестве подтверждения того, что она была получена. Поскольку не все объекты `transport` поддерживают метку EOF, данный протокол запрашивает у объекта `transport`, безопасно ли ее отправлять.

```
def eof_received(self):
    self.log.debug('received EOF')
    if self.transport.can_write_eof():
        self.transport.write_eof()
```

Когда соединение закрывается, будь то обычным способом или в результате возникновения ошибки, вызывается метод `connection_lost()`. В случае возникновения ошибки аргумент `error` содержит объект исключения, в противном случае он содержит значение `None`.

```
    def eof_received(self):
        self.log.debug('received EOF')
        if self.transport.can_write_eof():
            self.transport.write_eof()
```

Запуск сервера осуществляется в два этапа. Во-первых, приложение сообщает циклу событий о том, что необходимо создать новый объект сервера, используя заданные класс протокола, имя хоста и сокет. Метод `create_server()` — это сопрограмма, поэтому для фактического запуска сервера результаты должны быть обработаны циклом событий. По завершении работы сопрограммы создается экземпляр `asyncio.Server`, связанный с циклом событий.

```
# Создать сервер и позволить циклу завершить сопрограмму, прежде
# чем запустить реальный цикл событий
factory = event_loop.create_server(EchoServer, *SERVER_ADDRESS)
server = event_loop.run_until_complete(factory)
log.debug('starting up on {} port {}'.format(*SERVER_ADDRESS))
```

На следующем этапе должен быть запущен цикл событий, который будет обрабатывать события и клиентские запросы. Для длительно выполняющихся служб это проще всего обеспечить, вызвав метод `run_forever()`. В случае прекращения выполнения цикла событий из кода приложения или посредством посылки сигнала процессу сервер можно закрыть для освобождения неиспользуемых сокетом ресурсов. После этого можно закрыть цикл событий, чтобы завершить обработку других сопрограмм перед выходом из программы.

```
# Войти в бесконечный цикл событий для обработки всех соединений
try:
    event_loop.run_forever()
finally:
    log.debug('closing server')
    server.close()
    event_loop.run_until_complete(server.wait_closed())
    log.debug('closing event loop')
    event_loop.close()
```

10.5.8.2. Эхо-клиент

Создание клиента с использованием класса протокола очень похоже на создание сервера. Код клиента также начинается с импорта модулей `asyncio` и `logging` (раздел 14.80) с последующим созданием объекта цикла событий.

Листинг 10.99. `asyncio_echo_client_protocol.py`

```
import asyncio
import functools
import logging
import sys

MESSAGES = [
    b'This is the message. ',
    b'It will be sent ',
    b'in parts.',
]
SERVER_ADDRESS = ('localhost', 10000)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
    stream=sys.stderr,
)
log = logging.getLogger('main')

event_loop = asyncio.get_event_loop()
```

Клиентский класс протокола определяет те же методы, что и серверный, но с другой реализацией. Конструктор класса получает два аргумента: список сообщений, подлежащих отправке, и экземпляр `Future`, используемый для отправки сигнала, означающего, что клиент завершил рабочий цикл, получив ответ от сервера.

```
class EchoClient(asyncio.Protocol):

    def __init__(self, messages, future):
        super().__init__()
        self.messages = messages
        self.log = logging.getLogger('EchoClient')
        self.f = future
```

После успешного соединения с сервером клиент немедленно приступает к обмену данными. Сообщения отправляются по одному за раз, хотя базовый сетевой код может объединить несколько сообщений в одной пересылке. Когда вся последовательность сообщений исчерпывается, посылается метка конца файла EOF.

Несмотря на то что создается впечатление, будто все данные отправляются немедленно, в действительности объект `transport` буферизует исходящие данные и устанавливает функцию обратного вызова, которая осуществляет фактическую передачу данных, когда буфер сокета готов их принять. Эта обработка выполняется прозрачным образом, поэтому код приложения может быть написан так, словно операции ввода-вывода выполняются немедленно.

```
    def connection_made(self, transport):
        self.transport = transport
        self.address = transport.get_extra_info('peername')
        self.log.debug(
            'connecting to {} port {}'.format(*self.address)
        )
        # Здесь можно было бы использовать метод
        # transport.writelines(), но это затруднило бы
        # представление каждой части отправляемого сообщения
        for msg in self.messages:
            transport.write(msg)
            self.log.debug('sending {!r}'.format(msg))
        if transport.can_write_eof():
            transport.write_eof()
```

Полученный от сервера ответ протоколируется.

```
    def data_received(self, data):
        self.log.debug('received {!r}'.format(data))
```

Наконец, если получена метка конца файла или соединение закрыто на стороне сервера, локальный объект `transport` закрывается, а объект фьючерса помечается как завершивший работу вызовом метода `set_result()`.

```
    def eof_received(self):
        self.log.debug('received EOF')
        self.transport.close()
```

```
        if not self.f.done():
            self.f.set_result(True)

    def connection_lost(self, exc):
        self.log.debug('server closed connection')
        self.transport.close()
        if not self.f.done():
            self.f.set_result(True)
        super().connection_lost(exc)
```

Обычно соединение создается путем передачи класса протокола циклу событий. В данном случае, поскольку цикл событий не имеет возможности передать дополнительные аргументы конструктору протокола, необходимо обернуть класс клиента с помощью функции `partial()`, передав ей в качестве аргументов список сообщений и экземпляр `Future`. Далее этот новый вызываемый объект может быть использован вместо класса при вызове метода `create_connection()` для подключения клиента.

```
client_completed = asyncio.Future()

client_factory = functools.partial(
    EchoClient,
    messages=MESSAGES,
    future=client_completed,
)
factory_coroutine = event_loop.create_connection(
    client_factory,
    *SERVER_ADDRESS,
)
```

Чтобы инициировать выполнение клиента, цикл событий запускается один раз с сопрограммой для создания клиента, а второй — с экземпляром `Future`, передаваемым клиенту для обмена данными после завершения работы. Использование двух вызовов позволяет избежать создания бесконечного цикла в клиентской программе, которая, вероятно, захочет завершить выполнение после обмена данными с сервером. Если бы использовался только первый вызов, ожидающий создания клиента сопрограммой, то не исключено, что он не смог бы обработать все данные ответа и закрыть соединение с сервером надлежащим образом.

```
log.debug('waiting for client to complete')
try:
    event_loop.run_until_complete(factory_coroutine)
    event_loop.run_until_complete(client_completed)
finally:
    log.debug('closing event loop')
    event_loop.close()
```

10.5.8.3. Вывод

Выполнив серверную программу в одном окне, а клиентскую — в другом, можно получить следующий вывод.

```
$ python3 asyncio_echo_client_protocol.py
asyncio: Using selector: KqueueSelector
main: waiting for client to complete
EchoClient: connecting to ::1 port 10000
EchoClient: sending b'This is the message. '
EchoClient: sending b'It will be sent '
EchoClient: sending b'in parts.'
EchoClient: received b'This is the message. It will be sent in
parts.'
EchoClient: received EOF
EchoClient: server closed connection
main: closing event loop

$ python3 asyncio_echo_client_protocol.py
asyncio: Using selector: KqueueSelector
main: waiting for client to complete
EchoClient: connecting to ::1 port 10000
EchoClient: sending b'This is the message. '
EchoClient: sending b'It will be sent '
EchoClient: sending b'in parts.'
EchoClient: received b'This is the message. It will be sent in
parts.'
EchoClient: received EOF
EchoClient: server closed connection
main: closing event loop

$ python3 asyncio_echo_client_protocol.py
asyncio: Using selector: KqueueSelector
main: waiting for client to complete
EchoClient: connecting to ::1 port 10000
EchoClient: sending b'This is the message. '
EchoClient: sending b'It will be sent '
EchoClient: sending b'in parts.'
EchoClient: received b'This is the message. It will be sent in
parts.'
EchoClient: received EOF
EchoClient: server closed connection
main: closing event loop
```

Несмотря на то что клиент всегда отправляет сообщения по отдельности, при первом запуске клиента сервер получает одно длинное сообщение и отсылает его обратно клиенту. При последующих запусках клиента результаты могут выглядеть несколько иначе, в зависимости от загруженности сети, а также от того, сбрасываются ли сетевые буферы, прежде чем подготовятся все данные.

```
$ python3 asyncio_echo_server_protocol.py
asyncio: Using selector: KqueueSelector
main: starting up on localhost port 10000
EchoServer_::1_63347: connection accepted
EchoServer_::1_63347: received b'This is the message. It will
be sent in parts.'
EchoServer_::1_63347: sent b'This is the message. It will be
```

```
sent in parts.'
EchoServer_::1_63347: received EOF
EchoServer_::1_63347: closing

EchoServer_::1_63387: connection accepted
EchoServer_::1_63387: received b'This is the message. '
EchoServer_::1_63387: sent b'This is the message. '
EchoServer_::1_63387: received b'It will be sent in parts.'
EchoServer_::1_63387: sent b'It will be sent in parts.'
EchoServer_::1_63387: received EOF
EchoServer_::1_63387: closing

EchoServer_::1_63389: connection accepted
EchoServer_::1_63389: received b'This is the message. It will
be sent '
EchoServer_::1_63389: sent b'This is the message. It will be sent '
EchoServer_::1_63389: received b'in parts.'
EchoServer_::1_63389: sent b'in parts.'
EchoServer_::1_63389: received EOF
EchoServer_::1_63389: closing
```

10.5.9. Асинхронные операции ввода-вывода с использованием сопрограмм и потоков

В этом разделе исследуются альтернативные версии простого эхо-сервера и эхо-клиента, в которых вместо абстракций классов `protocol` и `transport` используются сопрограммы и Stream API модуля `asyncio`. Приведенные ниже примеры основаны на более низком уровне абстракции, чем API семейства абстрактных классов `Protocol`, но события обрабатываются похожим образом.

10.5.9.1. Эхо-сервер

Код эхо-сервера начинается с импорта модулей `asyncio` и `logging` (раздел 14.80) и создания объекта цикла событий.

Листинг 10.100. `asyncio_echo_server_coroutine.py`

```
import asyncio
import logging
import sys

SERVER_ADDRESS = ('localhost', 10000)
logging.basicConfig(
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
    stream=sys.stderr,
)
log = logging.getLogger('main')

event_loop = asyncio.get_event_loop()
```

Далее определяется сопрограмма, обеспечивающая взаимодействие с клиентом. Каждый раз, когда клиент запрашивает соединение, вызывается новый

экземпляр сопрограммы; поэтому код в теле функции всегда взаимодействует только с одним клиентом за раз. Состоянием каждого экземпляра сопрограммы управляет среда времени выполнения Python, поэтому никакие дополнительные структуры данных, которые отслеживали бы отдельных клиентов в коде приложения, не требуются.

Аргументами сопрограммы являются экземпляры `StreamReader` и `StreamWriter`, связанные с новым соединением. Как и в случае класса `Transport`, адрес клиента можно получить с помощью метода `get_extra_info()` объекта записи.

```
async def echo(reader, writer):
    address = writer.get_extra_info('peername')
    log = logging.getLogger('echo_{}_{}'.format(*address))
    log.debug('connection accepted')
```

Несмотря на то что сопрограмма вызывается, когда соединение уже установлено, данные, которые должны быть прочитаны, к этому времени еще могут отсутствовать. Чтобы избежать блокирования других операций при чтении данных, сопрограмма использует ключевое слово `await` совместно с вызовом метода `read()`, предоставляя циклу событий возможность выполнять другие задачи, пока не появятся данные для чтения.

```
    while True:
        data = await reader.read(128)
```

Если клиент отправил данные, `await` возвращает их, и они могут быть отправлены обратно клиенту путем передачи объекту записи. Исходящие данные могут буферизоваться с помощью многократных вызовов метода `write()` с последующим вызовом метода `drain()` для сброса результатов, оставшихся в буфере. Поскольку при сетевом вводе-выводе сброс буфера может блокировать другие операции, для возврата управления циклу событий вновь используется ключевое слово `await`, обеспечивающее мониторинг состояния объекта записи и его вызов при наличии данных, подлежащих отправке.

```
        if data:
            log.debug('received {!r}'.format(data))
            writer.write(data)
            await writer.drain()
            log.debug('sent {!r}'.format(data))
```

Если клиент не отправил никаких данных, то метод `read()` возвращает пустую байтовую строку, указывающую на то, что соединение закрывается. Сервер должен закрыть сокет, после чего сопрограмма может выполнить возврат, свидетельствующий о завершении ее выполнения.

```
        else:
            log.debug('closing')
            writer.close()
            return
```

Запуск сервера осуществляется в два этапа. Во-первых, приложение информирует цикл событий о том, что необходимо создать новый объект сервера с ис-

пользованием указанной сопрограммы, а также имени хоста и сокета, который будет прослушиваться. Метод `start_server()` сам является сопрограммой, поэтому для фактического запуска сервера результаты должны быть обработаны циклом событий. В результате выполнения этой сопрограммы создается экземпляр `asyncio.Server`, связанный с циклом событий.

```
# Создать сервер и позволить циклу завершить сопрограмму, прежде
# чем запустить реальный цикл событий
factory = asyncio.start_server(echo, *SERVER_ADDRESS)
server = event_loop.run_until_complete(factory)
log.debug('starting up on {} port {}'.format(*SERVER_ADDRESS))
```

На следующем этапе должен быть запущен цикл событий, который будет обрабатывать события и клиентские запросы. Для длительно выполняющихся служб это проще всего обеспечить, вызвав метод `run_forever()`. В случае прекращения цикла событий из кода приложения или посредством отправки сигнала процессу сервер можно закрыть для освобождения ресурсов, неиспользуемых сокетом. После этого можно закрыть цикл событий, чтобы завершить обработку других сопрограмм перед выходом из программы.

```
# Войти в бесконечный цикл событий для обработки всех соединений
try:
    event_loop.run_forever()
except KeyboardInterrupt:
    pass
finally:
    log.debug('closing server')
    server.close()
    event_loop.run_until_complete(server.wait_closed())
    log.debug('closing event loop')
    event_loop.close()
```

10.5.9.2. Эхо-клиент

Создание клиента с использованием сопрограммы очень напоминает создание сервера. Код клиента также начинается с импорта модулей `asyncio` и `logging` (раздел 14.80) с последующим созданием объекта цикла событий.

Листинг 10.101. `asyncio_echo_client_coroutine.py`

```
import asyncio
import logging
import sys

MESSAGES = [
    b'This is the message. ',
    b'It will be sent ',
    b'in parts.',
]
SERVER_ADDRESS = ('localhost', 10000)

logging.basicConfig(
```

```
    level=logging.DEBUG,
    format='%(name)s: %(message)s',
    stream=sys.stderr,
)

log = logging.getLogger('main')

event_loop = asyncio.get_event_loop()
```

Сопрограмма `echo_client` получает аргументы, содержащие информацию о расположении сервера и отправляемых сообщениях.

```
async def echo_client(address, messages):
```

Сопрограмма вызывается при запуске задачи, но в условиях отсутствия соединения, с которым можно работать. Поэтому первым шагом является установление клиентом собственного соединения. Чтобы избежать блокирования других операций во время выполнения сопрограммы `open_connection()`, используется ключевое слово `await`.

```
    log = logging.getLogger('echo_client')

    log.debug('connecting to {} port {}'.format(*address))
    reader, writer = await asyncio.open_connection(*address)
```

Сопрограмма `open_connection()` возвращает экземпляры `StreamReader` и `StreamWriter`, связанные с новым сокетом. Следующим шагом является отправка данных на сервер с помощью объекта записи. Как и в случае сервера, объект записи будет буферизовать исходящие данные до тех пор, пока сокет не перейдет в состояние готовности или пока метод `drain()` не будет использован для сброса результатов, оставшихся в буфере. Поскольку при сетевом вводе-выводе сброс буфера может блокировать другие операции, для возврата управления циклу событий вновь используется ключевое слово `await`, обеспечивающее мониторинг состояния объекта записи и его вызов при наличии данных, подлежащих отправке.

```
    # Здесь можно было бы использовать метод
    # transport.writelines(), но это затруднило бы
    # представление каждой части отправляемого сообщения
    for msg in messages:
        writer.write(msg)
        log.debug('sending {!r}'.format(msg))
    if writer.can_write_eof():
        writer.write_eof()
    await writer.drain()
```

Далее клиент получает ответ сервера, пытаясь читать данные до тех пор, пока будет что читать. Чтобы избежать блокирования на одном отдельном вызове метода `read()`, управление возвращается циклу событий с помощью ключевого слова `await`. Отправленные сервером данные протоколируются. Если сервер не отправил никаких данных, то метод `read()` возвращает пустую байтовую строку, указывающую на то, что соединение закрывается. Клиент должен сначала за-

крыть сокет, а затем выполнить возврат, свидетельствующий о завершении выполнения.

```
    log.debug('waiting for response')
    while True:
        data = await reader.read(128)
        if data:
            log.debug('received {!r}'.format(data))
        else:
            log.debug('closing')
            writer.close()
            return
```

Клиент запускается посредством вызова цикла событий с сопрограммой для создания клиента. Использование метода `run_until_complete()` для этой цели позволяет избежать создания бесконечного цикла в клиентской программе. В отличие от примера, в котором использовался класс протокола, создавать отдельный фьючерс, сигнализирующий о завершении работы сопрограммы, не требуется, поскольку функция `echo_client()` содержит всю клиентскую логику и не возвращает управление до тех пор, пока не получит ответ и не закроет соединение с сервером.

```
try:
    event_loop.run_until_complete(
        echo_client(SERVER_ADDRESS, MESSAGES)
    )
finally:
    log.debug('closing event loop')
    event_loop.close()
```

10.5.9.3. Вывод

Выполнив серверную программу в одном окне, а клиентскую — в другом, можно получить следующий вывод.

```
$ python3 asyncio_echo_client_coroutine.py
asyncio: Using selector: KqueueSelector
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. It will be sent in
parts.'
echo_client: closing
main: closing event loop

$ python3 asyncio_echo_client_coroutine.py
asyncio: Using selector: KqueueSelector
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
```

```
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. It will be sent in
parts.'
echo_client: closing
main: closing event loop

$ python3 asyncio_echo_client_coroutine.py
asyncio: Using selector: KqueueSelector
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. It will be sent '
echo_client: received b'in parts.'
echo_client: closing
main: closing event loop
```

Несмотря на то что клиент отправляет сообщения по отдельности, при первых двух запусках клиента сервер получает одно длинное сообщение и отсылает его обратно клиенту. При последующих запусках клиента результаты могут выглядеть несколько иначе, в зависимости от загруженности сети, а также от того, сбрасываются ли сетевые буферы до того, как будут подготовлены все данные.

```
$ python3 asyncio_echo_server_coroutine.py
asyncio: Using selector: KqueueSelector
main: starting up on localhost port 10000
echo_::1_64624: connection accepted
echo_::1_64624: received b'This is the message. It will be sent
in parts.'
echo_::1_64624: sent b'This is the message. It will be sent in parts.'
echo_::1_64624: closing

echo_::1_64626: connection accepted
echo_::1_64626: received b'This is the message. It will be sent
in parts.'
echo_::1_64626: sent b'This is the message. It will be sent in parts.'
echo_::1_64626: closing

echo_::1_64627: connection accepted
echo_::1_64627: received b'This is the message. It will be sent '
echo_::1_64627: sent b'This is the message. It will be sent '
echo_::1_64627: received b'in parts.'
echo_::1_64627: sent b'in parts.'
echo_::1_64627: closing
```

10.5.10. Использование SSL

В модуле `asyncio` предусмотрена встроенная поддержка протокола SSL при обмене данными через сокеты. Передача экземпляра `SSLContext` сопрограммам, создающим серверные или клиентские соединения, активизирует эту поддержку

и гарантирует настройку параметров SSL-протокола, прежде чем сокет будет предоставлен приложению для использования.

Рассмотренные в предыдущем разделе эхо-сервер и эхо-клиент на основе сопрограмм можно легко обновить для использования протокола SSL, внеся незначительные изменения. Первый шаг заключается в создании файлов сертификата и ключа. Команда наподобие следующей позволяет создать самоподписанный сертификат.

```
$ openssl req -newkey rsa:2048 -nodes -keyout pymotw.key \
-x509 -days 365 -out pymotw.crt
```

Команда `openssl` запрашивает несколько значений, которые используются для генерации сертификата, и создает запрошенные выходные файлы.

Слушающий сокет, который создавался в предыдущем примере сервера с помощью функции `start_server()`, не был безопасным.

```
factory = asyncio.start_server(echo, *SERVER_ADDRESS)
server = event_loop.run_until_complete(factory)
```

Чтобы добавить шифрование, следует создать экземпляр `SSLContext` с только что сгенерированными сертификатом и ключом и передать этот контекст функции `start_server()`.

```
# Сертификат создается с использованием имени хоста pymotw.com.
# При выполнении примера на другом хосте это имя не будет
# корректным, поэтому следует отменить его проверку.
ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_context.check_hostname = False
ssl_context.load_cert_chain('pymotw.crt', 'pymotw.key')

# Создать сервер и позволить ему завершить сопрограмму до
# запуска реального цикла событий
factory = asyncio.start_server(echo, *SERVER_ADDRESS,
                               ssl=ssl_context)
```

Аналогичные изменения должны быть внесены также в клиентскую программу. В прежней версии для создания соединения сокета с сервером используется функция `open_connection()`.

```
reader, writer = await asyncio.open_connection(*address)
```

Для создания безопасного сокета на стороне клиента следует вновь использовать экземпляр `SSLContext`. Поскольку аутентификация клиента не навязывается, необходимо загрузить только сертификат.

```
# Сертификат создается с использованием имени хоста pymotw.com.
# При выполнении примера на другом хосте это имя не будет
# корректным, поэтому следует отменить его проверку.
ssl_context = ssl.create_default_context(
    ssl.Purpose.SERVER_AUTH,
)
ssl_context.check_hostname = False
```

```
ssl_context.load_verify_locations('pymotw.crt')
reader, writer = await asyncio.open_connection(
    *server_address, ssl=ssl_context)
```

В клиенте требуется внесение еще одного небольшого изменения. Поскольку SSL-соединение не поддерживает отправку метки конца файла (EOF), клиент использует байт NULL в качестве завершающего символа сообщения. В прежней версии цикла отправки сообщений для этой цели используется метод `write_eof()`.

```
# Здесь можно было бы использовать метод
# writer.writelines(), но это затруднило бы
# представление каждой части отправляемого сообщения
for msg in messages:
    writer.write(msg)
    log.debug('sending {!r}'.format(msg))
if writer.can_write_eof():
    writer.write_eof()
await writer.drain()
```

Новая версия отправляет нулевой байт (`b'\x00'`) в качестве признака конца сообщения.

```
# Здесь можно было бы использовать метод
# writer.writelines(), но это затруднило бы
# представление каждой части отправляемого сообщения
for msg in messages:
    writer.write(msg)
    log.debug('sending {!r}'.format(msg))
# SSL не поддерживает метку EOF, поэтому для обозначения
# конца сообщения используется нулевой байт
writer.write(b'\x00')
await writer.drain()
```

Сопрограмма `echo()` на стороне сервера должна отслеживать байт NULL и при его получении закрывать соединение с клиентом.

```
async def echo(reader, writer):
    address = writer.get_extra_info('peername')
    log = logging.getLogger('echo_{}_{}'.format(*address))
    log.debug('connection accepted')
    while True:
        data = await reader.read(128)
        terminate = data.endswith(b'\x00')
        data = data.rstrip(b'\x00')
        if data:
            log.debug('received {!r}'.format(data))
            writer.write(data)
            await writer.drain()
            log.debug('sent {!r}'.format(data))
        if not data or terminate:
            log.debug('message terminated, closing connection')
            writer.close()
            return
```

Выполнив серверную программу в одном окне, а клиентскую — в другом, можно получить следующий вывод.

```
$ python3 asyncio_echo_server_ssl.py
asyncio: Using selector: KqueueSelector
main: starting up on localhost port 10000
echo_::1_53957: connection accepted
echo_::1_53957: received b'This is the message. '
echo_::1_53957: sent b'This is the message. '
echo_::1_53957: received b'It will be sent in parts.'
echo_::1_53957: sent b'It will be sent in parts.'
echo_::1_53957: message terminated, closing connection
```

```
$ python3 asyncio_echo_client_ssl.py
asyncio: Using selector: KqueueSelector
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. '
echo_client: received b'It will be sent in parts.'
echo_client: closing
main: closing event loop
```

10.5.11. Взаимодействие со службами DNS

Приложения используют сетевые соединения с серверами для взаимодействия с системой доменных имен (DNS) с целью выполнения таких операций, как взаимные преобразования между именами хостов и IP-адресами. Цикл событий модуля `asyncio` включает вспомогательные методы, позволяющие выполнять эти операции в фоновом режиме, чтобы избежать блокирования других операций во время обслуживания запросов.

10.5.11.1. Получение IP-адреса по имени хоста

Сопрограмма `getaddrinfo()` преобразует имя хоста и номер порта в IP- или IPv6-адрес. Как и в случае версии этой функции, содержащейся в модуле `socket` (раздел 11.2), возвращаемое значение представляет собой список кортежей, содержащих следующие пять элементов информации:

- семейство адресов;
- тип адреса;
- протокол;
- каноническое имя сервера;
- кортеж адреса сокета, пригодный для открытия соединения с сервером через первоначально указанный порт.

Запросы могут фильтроваться протоколом. В следующем примере фильтр гарантирует, что возвращаться будут только ответы, использующие протокол TCP.

Листинг 10.102. asyncio_getaddrinfo.py

```
import asyncio
import logging
import socket
import sys


TARGETS = [
    ('pymotw.com', 'https'),
    ('doughellmann.com', 'https'),
    ('python.org', 'https'),
]


async def main(loop, targets):
    for target in targets:
        info = await loop.getaddrinfo(
            *target,
            proto=socket.IPPROTO_TCP,
        )

        for host in info:
            print('{:20}: {}'.format(target[0], host[4][0]))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop, TARGETS))
finally:
    event_loop.close()
```

В приведенном примере программа преобразует имя хоста и имя протокола в IP-адрес и номер порта.

```
$ python3 asyncio_getaddrinfo.py

pymotw.com           : 66.33.211.242
doughellmann.com     : 66.33.211.240
python.org           : 23.253.135.79
python.org           : 2001:4802:7901::e60a:1375:0:6
```

10.5.11.2. Получение имени хоста по IP-адресу

Сопрограмма `getnameinfo()` работает в обратном направлении, преобразуя IP-адрес в имя хоста, а номер порта — в имя протокола, если это возможно.

Листинг 10.103. asyncio_getnameinfo.py

```
import asyncio
import logging
import socket
import sys
```

```
TARGETS = [
    ('66.33.211.242', 443),
    ('104.130.43.121', 443),
]


async def main(loop, targets):
    for target in targets:
        info = await loop.getnameinfo(target)
        print('{:15}: {} {}'.format(target[0], *info))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop, TARGETS))
finally:
    event_loop.close()
```

Как показывают результаты выполнения этого примера, IP-адрес для сайта `pymotw.com` ссылается на сервер компании DreamHost – хостинговой компании, предоставившей свою техническую площадку для размещения данного сайта. Второй из исследуемых IP-адресов относится к сайту `python.org` и не разрешается в имя хоста.

```
$ python3 asyncio_getnameinfo.py

66.33.211.242  : apache2-echo.catalina.dreamhost.com https
104.130.43.121 : 104.130.43.121 https
```

Дополнительные ссылки

- Обсуждение модуля `socket` (раздел 11.2), включающее более подробное рассмотрение описанных операций.

10.5.12. Работа с подпроцессами

Часто требуется работать с другими программами и процессами так, чтобы можно было воспользоваться существующим кодом, не переписывая его, или получить доступ к библиотекам или средствам, недоступным в Python. Как и в случае сетевых операций ввода-вывода, модуль `asyncio` включает две абстракции для запуска другой программы и последующего взаимодействия с ней.

10.5.12.1. Использование абстракции `Protocol` с подпроцессами

В следующем примере сопрограмма запускает процесс, выполняющий команду `df` в Unix, которая определяет размер свободного пространства на локальных дисках. Для запуска процесса и его связывания с классом `Protocol`, которому известно, как прочитать вывод команды `df` и выполнить его синтаксический анализ, используется метод `subprocess_exec()`. Методы класса `Protocol` вызываются автоматически на основании событий ввода-вывода подпроцесса. Поскольку для аргументов `stdin` и `stderr` установлены значения `None`, эти каналы передачи данных не подключены к новому процессу.

Листинг 10.104. `asyncio_subprocess_protocol.py`

```
import asyncio
import functools


async def run_df(loop):
    print('in run_df')

    cmd_done = asyncio.Future(loop=loop)
    factory = functools.partial(DFProtocol, cmd_done)
    proc = loop.subprocess_exec(
        factory,
        'df', '-hl',
        stdin=None,
        stderr=None,
    )
    try:
        print('launching process')
        transport, protocol = await proc
        print('waiting for process to complete')
        await cmd_done
    finally:
        transport.close()

    return cmd_done.result()
```

Класс `DFProtocol` происходит от класса `SubprocessProtocol`, определяющего API, который позволяет классу обмениваться данными с другим процессом посредством каналов. В качестве аргумента `done` ожидается экземпляр `Future`, который будет использоваться вызывающим кодом для наблюдения за завершением процесса.

```
class DFProtocol(asyncio.SubprocessProtocol):

    FD_NAMES = ['stdin', 'stdout', 'stderr']

    def __init__(self, done_future):
        self.done = done_future
        self.buffer = bytearray()
        super().__init__()
```

Как и в случае обмена данными через сокет, метод `connection_made()` вызывается при установлении входных каналов связи с новым процессом. Аргумент `transport` является экземпляром подкласса `BaseSubprocessTransport`. Он может читать выходные данные процесса и записывать данные в поток ввода процесса, если процесс был сконфигурирован для получения входных данных.

```
    def connection_made(self, transport):
        print('process started {}'.format(transport.get_pid()))
        self.transport = transport
```

Если процесс сгенерировал вывод, вызывается метод `pipe_data_received()` с дескриптором файла, в который были записаны данные, и данные читаются из канала. Класс `Protocol` сохраняет данные, полученные из канала стандартного вывода процесса, в буфере для последующей обработки.

```
    def pipe_data_received(self, fd, data):
        print('read {} bytes from {}'.format(len(data),
                                             self.FD_NAMES[fd]))
        if fd == 1:
            self.buffer.extend(data)
```

При завершении процесса вызывается метод `process_exited()`. Код завершения можно получить с помощью объекта `transport`, вызвав метод `get_returncode()`. В этом случае, при условии отсутствия ошибок, доступный вывод декодируется и анализируется, прежде чем будет возвращен через экземпляр `Future`. Если же возникают ошибки, то предполагается, что результат пуст. Установка результата фьючерса информирует сопрограмму `run_df()` о завершении процесса, после чего закрываются ресурсы и возвращается результат.

```
    def process_exited(self):
        print('process exited')
        return_code = self.transport.get_returncode()
        print('return code {}'.format(return_code))
        if not return_code:
            cmd_output = bytes(self.buffer).decode()
            results = self._parse_results(cmd_output)
        else:
            results = []
        self.done.set_result((return_code, results))
```

Вывод команды разбирается в последовательность словарей, сопоставляющих имена заголовков с их значениями для каждой строки вывода. Результирующий список является возвращаемым значением.

```
def _parse_results(self, output):
    print('parsing results')
    # Вывод содержит одну строку заголовков, каждый из
    # которых представляет собой одно слово. Каждая
    # последующая строка соответствует отдельной файловой
    # системе, а столбцы соответствуют заголовкам
    # (предполагается, что имена точек монтирования не
    # содержат пробелов).
    if not output:
        return []
    lines = output.splitlines()
    headers = lines[0].split()
    devices = lines[1:]
    results = [
        dict(zip(headers, line.split()))
        for line in devices
    ]
    return results
```

Сопрограмма `run_df()` выполняется с помощью метода `run_until_complete()`. Вывод содержит размер свободного пространства на каждом устройстве.

```
event_loop = asyncio.get_event_loop()
try:
    return_code, results = event_loop.run_until_complete(
        run_df(event_loop)
    )
finally:
    event_loop.close()

if return_code:
    print('error exit {}'.format(return_code))
else:
    print('\nFree space:')
    for r in results:
        print('{Mounted:25}: {Avail}'.format(**r))
```

Представленный ниже вывод показывает последовательность выполняемых действий и размер свободного пространства для каждого из дисков системы, в которой выполняется данная программа.

```
$ python3 asyncio_subprocess_protocol.py
in run_df
launching process
process started 49675
waiting for process to complete
read 332 bytes from stdout
process exited
return code 0
parsing results

Free space:

/                         : 233Gi
/Volumes/hubertinternal   : 157Gi
/Volumes/hubert-tm        : 2.3Ti
```

10.5.12.2. Вызов подпроцессов с использованием сопрограмм и потоков ввода-вывода

Вместо получения доступа к процессам посредством подкласса `Protocol` их можно запускать непосредственно с помощью сопрограмм путем вызова функции `create_subprocess_exec()`, указав ей, следует ли подключать к каналам потоки ввода-вывода `stdout`, `stderr` и `stdin`. Результатом порождения подпроцесса сопрограммой является экземпляр `Process`, который можно использовать для манипулирования подпроцессом и обмена данными с ним.

Листинг 10.105. `asyncio_subprocess_coroutine.py`

```
import asyncio
import asyncio.subprocess
```

```
async def run_df():
    print('in run_df')

    buffer = bytearray()

    create = asyncio.create_subprocess_exec(
        'df', '-hl',
        stdout=asyncio.subprocess.PIPE,
    )
    print('launching process')
    proc = await create
    print('process started {}'.format(proc.pid))
```

В этом примере команда `df` не нуждается в иных входных данных, кроме аргументов командной строки, поэтому следующий шаг заключается в чтении всего вывода. В случае экземпляров `Protocol` невозможно контролировать, какой объем данных читается за один раз. В этом примере для чтения данных используется метод `readline()`, но для чтения данных, не разбитых на строки, также можно было бы использовать метод `read()`. Выходные результаты команды буферизуются, как и в примере, в котором использовался объект протокола, поэтому их синтаксический анализ может быть выполнен позднее.

```
    while True:
        line = await proc.stdout.readline()
        print('read {!r}'.format(line))
        if not line:
            print('no more output from command')
            break
        buffer.extend(line)
```

Если больше нечего читать, поскольку программа закончила свою работу, то метод `readline()` возвращает пустую байтовую строку. Чтобы гарантировать надлежащее закрытие ресурсов процессом, организуется ожидание его полного завершения с помощью ключевого слова `await`.

```
    print('waiting for process to complete')
    await proc.wait()
```

В этот момент можно проверить код завершения, чтобы определить, какие именно действия следует выполнить после того, как процесс перестал предоставлять результаты: выполнить синтаксический анализ вывода или обработать ошибку. Логика анализа вывода команды остается той же, что и в предыдущем примере, но она вынесена в отдельную функцию (здесь не представлена) ввиду отсутствия класса протокола, в котором ее можно было бы скрыть. После выполнения анализа данных результаты и код завершения возвращаются вызывающему коду.

```
    return_code = proc.returncode
    print('return code {}'.format(return_code))
    if not return_code:
        cmd_output = bytes(buffer).decode()
        results = _parse_results(cmd_output)
```

```
    else:
        results = []

    return (return_code, results)
```

Основная программа аналогична программе из примера, основанного на использовании объекта протокола, поскольку изменения в реализации изолированы в сопрограмме `run_df()`.

```
event_loop = asyncio.get_event_loop()
try:
    return_code, results = event_loop.run_until_complete(
        run_df()
    )
finally:
    event_loop.close()

if return_code:
    print('error exit {}'.format(return_code))
else:
    print('\nFree space:')
    for r in results:
        print('{Mounted:25}: {Avail}'.format(**r))
```

Поскольку вывод команды `df` можно читать по одной строке за раз, он отображается на консоли для того, чтобы можно было контролировать ход выполнения программы. Во всем остальном результаты совпадают с теми, которые были получены в предыдущем примере.

```
$ python3 asyncio_subprocess_coroutine.py

in run_df
launching process
process started 49678
read b'Filesystem Size Used Avail Capacity iused
ifree %iused Mounted on\n'
read b'/dev/disk2s2 446Gi 213Gi 233Gi 48% 55955082
61015132 48% /\n'
read b'/dev/disk1 465Gi 307Gi 157Gi 67% 80514922
41281172 66% /Volumes/hubertinternal\n'
read b'/dev/disk3s2 3.6Ti 1.4Ti 2.3Ti 38% 181837749
306480579 37% /Volumes/hubert-tm\n'
read b''
no more output from command
waiting for process to complete
return code 0
parsing results

Free space:
/                         : 233Gi
/Volumes/hubertinternal   : 157Gi
/Volumes/hubert-tm        : 2.3Ti
```

10.5.12.3. Отправка данных подпроцессу

В обоих предыдущих примерах использовался только один канал связи, который обеспечивал чтение данных из второго процесса. Часто возникает необходимость в передаче данных команде для последующей обработки. Следующий пример определяет сопрограмму для выполнения команды `tr` в Unix, которая преобразует символы, поступающие в ее входной поток. В данном случае команда `tr` используется для преобразования букв из нижнего регистра в верхний.

Сопрограмма `to_upper()` получает в качестве аргумента входную строку и порождает второй процесс, выполняющий команду `"tr [:lower:] [:upper:]"`.

Листинг 10.106. asyncio_subprocess_coroutine_write.py

```
import asyncio
import asyncio.subprocess


async def to_upper(input):
    print('in to_upper')

    create = asyncio.create_subprocess_exec(
        'tr', '[:lower:]', '[:upper:]',
        stdout=asyncio.subprocess.PIPE,
        stdin=asyncio.subprocess.PIPE,
    )
    print('launching process')
    proc = await create
    print('pid {}'.format(proc.pid))
```

Затем сопрограмма `to_upper()` использует метод `communicate()` экземпляра `Process` для отправки входной строки команде и читает весь результирующий вывод в асинхронном режиме. Как и в случае версии того же метода для экземпляра `subprocess.Popen`, метод `communicate()` возвращает все байтовые строки, выводимые этим методом. Если же команда может выводить слишком большой объем данных, которые будут потреблять много памяти, или же входные данные не могут быть предоставлены все сразу, или вывод должен обрабатываться инкрементно, то, возможно, вместо вызова метода `communicate()` лучше использовать непосредственно дескрипторы `stdin`, `stdout` и `stderr` экземпляра `Process`.

```
    print('communicating with process')
    stdout, stderr = await proc.communicate(input.encode())
```

После выполнения операций ввода-вывода организуется ожидание полного завершения процесса, что гарантирует корректное выполнение операций по освобождению ресурсов.

```
    print('waiting for process to complete')
    await proc.wait()
```

Далее можно проверить код завершения процесса, декодировать выходную байтовую строку и сформировать на основании этого результат, возвращаемый сопрограммой.

```
    return_code = proc.returncode
    print('return code {}'.format(return_code))
    if not return_code:
        results = bytes(stdout).decode()
    else:
        results = ''

    return (return_code, results)
```

В основной части программы задается строка сообщения, которую необходимо преобразовать, создается цикл событий для выполнения сопрограммы `to_upper()` и выводятся результаты.

```
MESSAGE = """
This message will be converted
to all caps.
"""

event_loop = asyncio.get_event_loop()
try:
    return_code, results = event_loop.run_until_complete(
        to_upper(MESSAGE)
    )
finally:
    event_loop.close()

if return_code:
    print('error exit {}'.format(return_code))
else:
    print('Original: {!r}'.format(MESSAGE))
    print('Changed : {!r}'.format(results))
```

Вывод отображает последовательность выполняемых операций, а также исходное и преобразованное сообщения.

```
$ python3 asyncio_subprocess_coroutine_write.py

in to_upper
launching process
pid 49684
communicating with process
waiting for process to complete
return code 0
Original: '\nThis message will be converted\nto all caps.\n'
Changed : '\nTHIS MESSAGE WILL BE CONVERTED\nTO ALL CAPS.\n'
```

10.5.13. Получение сигналов Unix

Обычно уведомления о системных событиях Unix прерывают работу приложений, запуская соответствующие обработчики событий. При работе с модулем `asyncio` выполнение обратных вызовов обработчиков сигналов чередуется с выполнением других сопрограмм и функций обратного вызова, управляемых ци-

клом событий. Подобная интеграция приводит к уменьшению числа прерываемых функций и минимизирует потребность в специальных мерах по защите ресурсов в случае прерывания операций.

Обработчики сигналов должны быть обычными вызываемыми объектами, а не сопрограммами.

Листинг 10.107. asyncio_signal.py

```
import asyncio
import functools
import os
import signal


def signal_handler(name):
    print('signal_handler({!r})'.format(name))
```

Обработчики сигналов регистрируются с помощью метода `add_signal_handler()`. Первый аргумент — это сигнал, второй — функция обратного вызова. Функциям обратного вызова не передаются никакие аргументы, поэтому, если необходимо передавать им аргументы, их следует обертывать функцией `functools.partial()`.

```
event_loop = asyncio.get_event_loop()

event_loop.add_signal_handler(
    signal.SIGHUP,
    functools.partial(signal_handler, name='SIGHUP'),
)
event_loop.add_signal_handler(
    signal.SIGUSR1,
    functools.partial(signal_handler, name='SIGUSR1'),
)
event_loop.add_signal_handler(
    signal.SIGINT,
    functools.partial(signal_handler, name='SIGINT'),
)
```

В этом примере программа использует сопрограмму для отправки сигналов самой себе посредством вызова `os.kill()`. После отправки каждого сигнала программа уступает управление, обеспечивая возможность выполнения обработчика. В реальном приложении код будет содержать больше мест, в которых управление будет уступаться циклу событий, поэтому предпринимать какие-либо искусственные меры для этого (как в данном примере) не потребуется.

```
async def send_signals():
    pid = os.getpid()
    print('starting send_signals for {}'.format(pid))

    for name in ['SIGHUP', 'SIGHUP', 'SIGUSR1', 'SIGINT']:
        print('sending {}'.format(name))
        os.kill(pid, getattr(signal, name))
        # Уступить управление, чтобы позволить выполняться
```

```
        # обработчику сигнала, поскольку сам по себе сигнал
        # не прерывает работу программы
        print('yielding control')
        await asyncio.sleep(0.01)
    return
```

Основная программа выполняет сопрограмму `send_signals()` до тех пор, пока не отправит все сигналы.

```
try:
    event_loop.run_until_complete(send_signals())
finally:
    event_loop.close()
```

Как следует из приведенного ниже вывода, обработчики вызываются тогда, когда сопрограмма `send_signals()` уступает управление после отправки сигнала.

```
$ python3 asyncio_signal.py

starting send_signals for 21772
sending SIGHUP
yielding control
signal_handler('SIGHUP')
sending SIGHUP
yielding control
signal_handler('SIGHUP')
sending SIGUSR1
yielding control
signal_handler('SIGUSR1')
sending SIGINT
yielding control
signal_handler('SIGINT')
```

Дополнительные ссылки

- `signal` (раздел 10.2). Получение уведомлений об асинхронных системных событиях.

10.5.14. Сочетание сопрограмм с потоками и процессами

Многие предопределенные библиотеки сами по себе не готовы к непосредственному использованию совместно с модулем `asyncio`. Они могут блокироваться или зависеть от средств параллелизма, недоступных через этот модуль. Однако возможность работы приложений на основе модуля `asyncio` с такими библиотеками все-таки можно обеспечить за счет использования *объекта-исполнителя* (раздел 10.6), запускающего код либо в отдельном потоке, либо в отдельном процессе.

10.5.14.1. Потоки

Метод `run_in_executor()` цикла событий получает в качестве аргументов экземпляр `executor`, который обычно является вызываемым объектом (функцией), и аргументы, которые должны быть переданы этому вызываемому объекту. Возвращаемым значением метода `run_in_executor()` является экземпляр

Future, который можно использовать для ожидания завершения работы указанной функции и возврата соответствующего значения. Если экземпляр executor не предоставляется, то создается экземпляр ThreadPoolExecutor. В следующем примере явным образом создается объект executor с ограниченным количеством доступных ему потоков.

Экземпляр ThreadPoolExecutor запускает свои рабочие потоки и вызывает в каждом из них предоставленные экземпляры функции. В этом примере показано, каким образом совместное использование методов run_in_executor() и wait() позволяет сопрограмме сначала уступать управление циклу событий, пока блокирующие функции выполняются в отдельных потоках, а затем пробуждаться, когда эти функции завершают свою работу.

Листинг 10.108. asyncio_executor_thread.py

```
import asyncio
import concurrent.futures
import logging
import sys
import time


def blocks(n):
    log = logging.getLogger('blocks({})'.format(n))
    log.info('running')
    time.sleep(0.1)
    log.info('done')
    return n ** 2


async def run_blocking_tasks(executor):
    log = logging.getLogger('run_blocking_tasks')
    log.info('starting')

    log.info('creating executor tasks')
    loop = asyncio.get_event_loop()
    blocking_tasks = [
        loop.run_in_executor(executor, blocks, i)
        for i in range(6)
    ]
    log.info('waiting for executor tasks')
    completed, pending = await asyncio.wait(blocking_tasks)
    results = [t.result() for t in completed]
    log.info('results: {!r}'.format(results))

    log.info('exiting')


if __name__ == '__main__':
    # Сконфигурировать протоколирование для отображения имени
    # потока, из которого поступило сообщение
    logging.basicConfig(
        level=logging.INFO,
```

```
        format='%(threadName)10s %(name)18s: %(message)s',
        stream=sys.stderr,
    )

    # Создать ограниченный пул потоков
    executor = concurrent.futures.ThreadPoolExecutor(
        max_workers=3,
    )

    event_loop = asyncio.get_event_loop()
    try:
        event_loop.run_until_complete(
            run_blocking_tasks(executor)
        )
    finally:
        event_loop.close()
```

Использование модуля `logging` (раздел 14.80) в сценарии `asyncio_executor_thread.py` позволяет удобным образом проследить за тем, с каким потоком и какой функцией связано каждое сообщение. Поскольку в каждом вызове `blocks()` используется свой объект `log`, вывод отчетливо показывает, что одни и те же потоки повторно используются для вызова нескольких экземпляров функций с различными аргументами.

```
$ python3 asyncio_executor_thread.py

MainThread run_blocking_tasks: starting
MainThread run_blocking_tasks: creating executor tasks
Thread-1 blocks(0): running
Thread-2 blocks(1): running
Thread-3 blocks(2): running
MainThread run_blocking_tasks: waiting for executor tasks
Thread-1 blocks(0): done
Thread-3 blocks(2): done
Thread-1 blocks(3): running
Thread-2 blocks(1): done
Thread-3 blocks(4): running
Thread-2 blocks(5): running
Thread-1 blocks(3): done
Thread-2 blocks(5): done
Thread-3 blocks(4): done
MainThread run_blocking_tasks: results: [16, 4, 1, 0, 25, 9]
MainThread run_blocking_tasks: exiting
```

10.5.14.2. Процессы

Экземпляр `ProcessPoolExecutor` работает во многом аналогично экземпляру `ThreadPoolExecutor`, создавая набор рабочих процессов вместо рабочих потоков. Несмотря на то что использование отдельных процессов требует большего количества системных ресурсов, для интенсивных вычислительных операций целесообразно выполнять отдельные задачи на каждом ядре CPU.

Листинг 10.109. asyncio_executor_process.py

```
# Изменения по сравнению со сценарием asyncio_executor_thread.py

if __name__ == '__main__':
    # Сконфигурировать протоколирование для отображения
    # идентификатора процесса, из которого поступило сообщение
    logging.basicConfig(
        level=logging.INFO,
        format='PID %(process)5s %(name)18s: %(message)s',
        stream=sys.stderr,
    )

    # Создать ограниченный пул процессов
    executor = concurrent.futures.ProcessPoolExecutor(
        max_workers=3,
    )

    event_loop = asyncio.get_event_loop()
    try:
        event_loop.run_until_complete(
            run_blocking_tasks(executor)
        )
    finally:
        event_loop.close()
```

Единственное изменение, которое потребовалось внести для перехода от потоков к процессам, заключается в создании объекта-исполнителя другого типа. Кроме того, в этом примере в строке формата вместо имени потока используется идентификатор процесса, чтобы продемонстрировать, что задачи действительно выполняются в отдельных процессах.

```
$ python3 asyncio_executor_process.py

PID 16429 run_blocking_tasks: starting
PID 16429 run_blocking_tasks: creating executor tasks
PID 16429 run_blocking_tasks: waiting for executor tasks
PID 16430 blocks(0): running
PID 16431 blocks(1): running
PID 16432 blocks(2): running
PID 16430 blocks(0): done
PID 16432 blocks(2): done
PID 16431 blocks(1): done
PID 16430 blocks(3): running
PID 16432 blocks(4): running
PID 16431 blocks(5): running
PID 16431 blocks(5): done
PID 16432 blocks(4): done
PID 16430 blocks(3): done
PID 16429 run_blocking_tasks: results: [4, 0, 16, 1, 9, 25]
PID 16429 run_blocking_tasks: exiting
```

10.5.15. Отладка с помощью модуля asyncio

В модуле asyncio предусмотрены полезные встроенные возможности отладки. Например, цикл событий может использовать модуль logging (раздел 14.80) для генерации сообщений о состоянии во время выполнения. Одни из этих сообщений доступны, если в приложении были активизированы средства протоколирования, другие могут быть включены непосредственным уведомлением цикла о необходимости генерации дополнительных отладочных сообщений. Передав методу set_debug() булево значение, можно указать, должна ли быть включена отладка.

Поскольку приложения, созданные на основе модуля asyncio, крайне чувствительны к сопрограммам, жадно потребляющим ресурсы и при этом не уступающим управления, в цикл событий встроена поддержка, обеспечивающая обнаружение медленных функций обратного вызова. Можно активизировать это средство, включив отладку, и управлять определением того, что считать "медленным", путем задания для свойства slow_callback_duration длительности периода выполнения в секундах, в случае превышения которого должно выводиться предупреждение.

Наконец, если приложение, использующее модуль asyncio, завершается без выполнения надлежащих операций по закрытию сопрограмм и других ресурсов, то такое поведение может указывать на наличие логической ошибки, препятствующей нормальному выполнению кода. Включение уровня детализации сообщений ResourceWarning обеспечит вывод соответствующих предупреждений при завершении работы программы.

Листинг 10.110. asyncio_debug.py

```
import argparse
import asyncio
import logging
import sys
import time
import warnings

parser = argparse.ArgumentParser('debugging asyncio')
parser.add_argument(
    '-v',
    dest='verbose',
    default=False,
    action='store_true',
)
args = parser.parse_args()

logging.basicConfig(
    level=logging.DEBUG,
    format='%(levelname)7s: %(message)s',
    stream=sys.stderr,
)
LOG = logging.getLogger('')
```

```
async def inner():
    LOG.info('inner starting')
    # Использовать блокирующую паузу для
    # имитации выполнения работы функцией
    time.sleep(0.1)
    LOG.info('inner completed')


async def outer(loop):
    LOG.info('outer starting')
    await asyncio.ensure_future(loop.create_task(inner()))
    LOG.info('outer completed')


event_loop = asyncio.get_event_loop()
if args.verbose:
    LOG.info('enabling debugging')

    # Включить отладку
    event_loop.set_debug(True)

    # В целях иллюстрации установить очень низкий порог для
    # "медленных" задач. По умолчанию он равен 0.1, или
    # 100 миллисекунд.
    event_loop.slow_callback_duration = 0.001

    # Сообщить обо всех ошибках управления асинхронными
    # ресурсами
    warnings.simplefilter('always', ResourceWarning)

LOG.info('entering event loop')
event_loop.run_until_complete(outer(event_loop))
```

Результаты работы этого сценария с отключенной отладкой не дают никакого повода для беспокойства.

```
$ python3 asyncio_debug.py

  DEBUG: Using selector: KqueueSelector
   INFO: entering event loop
   INFO: outer starting
   INFO: inner starting
   INFO: inner completed
   INFO: outer completed
```

Однако включение режима отладки позволяет выявить наличие некоторых проблем в приложении. Например, несмотря на то что сопрограмма `inner()` завершается нормально, для этого ей потребовалось время, которое превышает предельное значение, установленное в свойстве `slow_callback_duration`. Кроме того, цикл событий не был надлежащим образом закрыт при завершении программы.

```
$ python3 asyncio_debug.py -v

  DEBUG: Using selector: KqueueSelector
   INFO: enabling debugging
   INFO: entering event loop
   INFO: outer starting
   INFO: inner starting
   INFO: inner completed
WARNING: Executing <Task finished coro=<inner() done, defined at
asyncio_debug.py:34> result=None created at asyncio_debug.py:44>
took 0.102 seconds
   INFO: outer completed
.../lib/python3.5/asyncio/base_events.py:429: ResourceWarning:
unclosed event loop <_UnixSelectorEventLoop running=False
closed=False debug=True>
  DEBUG: Close <_UnixSelectorEventLoop running=False
closed=False debug=True>
```

Примечание

В Python 3.5 модуль `asyncio` все еще является экспериментальным. Его API был стабилизирован в версии Python 3.6, и для большинства изменений, вносимых в последующие выпуски Python 3.5, была обеспечена ретроподдержка. В результате этого данный модуль может работать несколько иначе в различных версиях Python 3.5.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `asyncio`[10].
- **PEP 3156**[11]. *Asynchronous IO Support Rebooted: The "asyncio" Module.*
- **PEP 380**[12]. *Syntax for Delegating to a Subgenerator.*
- **PEP 492**[13]. *Coroutines with async and await syntax.*
- `concurrent.futures` (раздел 10.6). Управление пулами параллельных задач.
- `socket` (раздел 11.2). Низкоуровневый сетевой обмен данными.
- `select` (раздел 11.4). Низкоуровневые средства асинхронного ввода-вывода.
- `socketserver` (раздел 11.5). Фреймворк для создания сетевых серверов.
- *What's New in Python 3.6: asyncio*[14]. Сводка изменений, внесенных в модуль `asyncio` в результате стабилизации API в версии Python 3.6.
- `trollius`[15]. Интерфейс Tulip — первоначальная версия модуля `asyncio` для Python 2.
- *The New asyncio Module in Python 3.4: Event Loops* (Gaston Hillar)[16]. Статья, опубликованная в журнале *Dr. Dobb's* и содержащая обсуждение модуля `asyncio`.

[10] https://docs.python.org/3.5/library/asyncio.html

[11] www.python.org/dev/peps/pep-3156

[12] www.python.org/dev/peps/pep-0380

[13] www.python.org/dev/peps/pep-0492

[14] https://docs.python.org/3/whatsnew/3.6.html#asyncio

[15] https://pypi.python.org/pypi/trollius

[16] www.drdobbs.com/open-source/the-new-asyncio-module-in-python-34-even/240168401

- *Exploring Python 3's Asyncio by Example* (Chad Lung)[17]. Статья в блоге с примерами использования модуля *asyncio*.
- *A Web Crawler with Asyncio Coroutines* (A. Jesse Jiryu Davis, Guido van Rossum)[18]. Статья на сайте *The Architecture of Open Source Applications*.
- *Playing with asyncio* (Nathan Hoad)[19]. Статья в блоге, посвященная использованию модуля `asyncio`.
- *Async I/O and Python* (Mark McLoughlin)[20]. Статья в блоге, в которой обсуждается выполнение асинхронных операций ввода-вывода средствами Python.
- *A Curious Course on Coroutines and Concurrency* (David Beazley)[21]. Практическое руководство, представленное на конференции PyCon 2009.
- *How the heck does async/await work in Python 3.5?* (Brett Cannon)[22]. Статья в блоге.
- *Unix Network Programming, Volume 1. The Sockets Networking API, Third Edition*, by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Addison-Wesley Professional, 2004. ISBN-10: 0131411551.
- *Foundations of Python Network Programming, Third Edition*, by Brandon Rhodes and John Goerzen. Apress, 2014. ISBN-10: 1430258543.

10.6. `concurrent.futures`: управление пулами параллельных задач

Модуль `concurrent.futures` предоставляет интерфейсы для выполнения задач с использованием пулов рабочих потоков или процессов. Эти интерфейсы одинаковы для обеих вариантов, поэтому переход от использования потоков к использованию процессов и наоборот требует внесения лишь минимальных изменений в приложения.

Данный модуль предоставляет два класса для взаимодействия с пулами. Для управления пулами рабочих объектов используются объекты-исполнители, а для управления вычисленными с их помощью результатами — объекты-фьючерсы. Чтобы использовать пул рабочих потоков или процессов, приложение создает экземпляр исполнителя соответствующего класса и передает ему задачи для выполнения. При запуске каждой задачи возвращается экземпляр `Future` (фьючерс). Когда потребуется результат выполнения задачи, приложение может использовать фьючерс в целях блокирования до тех пор, пока результат не станет доступным. Модуль предоставляет различные программные интерфейсы, обеспечивающие удобные способы ожидания завершения задач, поэтому непосредственное управление объектами `Future` не требуется.

---

[17] www.giantflyingsaucer.com/blog/?p=5557

[18] http://aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html

[19] www.getoffmalawn.com/blog/playing-with-asyncio

[20] https://blogs.gnome.org/markmc/2013/06/04/async-io-and-python/

[21] www.dabeaz.com/coroutines/

[22] www.snarky.ca/how-the-heck-does-async-await-work-in-python-3-5

10.6.1. Использование метода map() с базовым пулом потоков

Класс ThreadPoolExecutor управляет набором рабочих потоков, передавая им задачи по мере того, как они освобождаются для выполнения очередной порции работы. В следующем примере с помощью метода map() организуется получение набора результатов параллельной обработки входного итерируемого объекта. С помощью метода time.sleep() в задачах создаются паузы различной длительности, чтобы продемонстрировать, что метод map() всегда возвращает результаты в соответствии с порядком поступления входных данных, независимо от порядка завершения параллельных задач.

Листинг 10.111. futures_thread_pool_map.py

```
from concurrent import futures
import threading
import time


def task(n):
    print('{}: sleeping {}'.format(
        threading.current_thread().name,
        n)
    )
    time.sleep(n / 10)
    print('{}: done with {}'.format(
        threading.current_thread().name,
        n)
    )
    return n / 10


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
results = ex.map(task, range(5, 0, -1))
print('main: unprocessed results {}'.format(results))
print('main: waiting for real results')
real_results = list(results)
print('main: results: {}'.format(real_results))
```

Возвращаемое методом map() значение в действительности является специальным типом итератора, которому известно, как следует дожидаться очередного ответа по мере того, как основная программа выполняет итерации по нему.

```
$ python3 futures_thread_pool_map.py

main: starting
Thread-1: sleeping 5
Thread-2: sleeping 4
main: unprocessed results <generator object
Executor.map.<locals>.result_iterator at 0x1013c80a0>
main: waiting for real results
Thread-2: done with 4
```

```
Thread-2: sleeping 3
Thread-1: done with 5
Thread-1: sleeping 2
Thread-1: done with 2
Thread-1: sleeping 1
Thread-2: done with 3
Thread-1: done with 1
main: results: [0.5, 0.4, 0.3, 0.2, 0.1]
```

10.6.2. Планирование индивидуальных задач

Помимо метода `map()` существует еще одна возможность планировать выполнение индивидуальных задач с помощью объекта-исполнителя, используя метод `submit()`. Возвращаемый им экземпляр `Future` может быть далее использован для ожидания результатов выполнения конкретной задачи.

Листинг 10.112. `futures_thread_pool_submit.py`

```
from concurrent import futures
import threading
import time


def task(n):
    print('{}: sleeping {}'.format(
        threading.current_thread().name,
        n)
    )
    time.sleep(n / 10)
    print('{}: done with {}'.format(
        threading.current_thread().name,
        n)
    )
    return n / 10


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
f = ex.submit(task, 5)
print('main: future: {}'.format(f))
print('main: waiting for results')
result = f.result()
print('main: result: {}'.format(result))
print('main: future after result: {}'.format(f))
```

Изменение состояния объекта `Future` происходит лишь после того, как завершаются все задачи и результат становится доступным.

```
$ python3 futures_thread_pool_submit.py

main: starting
Thread-1: sleeping 5
main: future: <Future at 0x1010e6080 state=running>
```

```
main: waiting for results
Thread-1: done with 5
main: result: 0.5
main: future after result: <Future at 0x1010e6080 state=finished
 returned float>
```

10.6.3. Ожидание завершения задач в произвольном порядке

Вызов метода `result()` объекта `Future` блокируется до тех пор, пока не завершатся все задачи (либо посредством возврата значения, либо посредством возбуждения исключения), или пока он не будет отменен. Результаты нескольких задач становятся доступными в том порядке, в каком их выполнение было запланировано с помощью метода `map()`. Если порядок, в каком должны обрабатываться результаты, не имеет значения, то метод `as_completed()` позволяет обрабатывать их по мере завершения каждой задачи.

Листинг 10.113. `futures_as_completed.py`

```
from concurrent import futures
import random
import time


def task(n):
    time.sleep(random.random())
    return (n, n / 10)


ex = futures.ThreadPoolExecutor(max_workers=5)
print('main: starting')

wait_for = [
    ex.submit(task, i)
    for i in range(5, 0, -1)
]

for f in futures.as_completed(wait_for):
    print('main: result: {}'.format(f.result()))
```

Поскольку в пуле содержится столько потоков, сколько имеется задач, можно запустить все задачи. Задачи завершаются в случайном порядке, поэтому значения, генерируемые методом `as_completed()`, будут отличаться от одного запуска программы к другому.

```
$ python3 futures_as_completed.py

main: starting
main: result: (3, 0.3)
main: result: (5, 0.5)
main: result: (4, 0.4)
main: result: (2, 0.2)
main: result: (1, 0.1)
```

10.6.4. Обратные вызовы с использованием экземпляров `Future`

Чтобы предпринять некоторое действие по завершении задачи без явного ожидания результата, можно указать с помощью метода `add_done_callback()` другую функцию, которая должна быть вызвана, когда экземпляр `Future` перейдет в состояние готовности. Функция обратного вызова должна быть вызываемым объектом, получающим единственный аргумент: экземпляр `Future`.

Листинг 10.114. `futures_future_callback.py`

```
from concurrent import futures
import time


def task(n):
    print('{}: sleeping'.format(n))
    time.sleep(0.5)
    print('{}: done'.format(n))
    return n / 10


def done(fn):
    if fn.cancelled():
        print('{}: canceled'.format(fn.arg))
    elif fn.done():
        error = fn.exception()
        if error:
            print('{}: error returned: {}'.format(
                fn.arg, error))
        else:
            result = fn.result()
            print('{}: value returned: {}'.format(
                fn.arg, result))


if __name__ == '__main__':
    ex = futures.ThreadPoolExecutor(max_workers=2)
    print('main: starting')
    f = ex.submit(task, 5)
    f.arg = 5
    f.add_done_callback(done)
    result = f.result()
```

Функция обратного вызова запускается независимо от причины, по которой экземпляр переходит в состояние готовности, поэтому необходимо проверять состояние передаваемого ей объекта перед тем, как использовать его.

```
$ python3 futures_future_callback.py

main: starting
5: sleeping
5: done
5: value returned: 0.5
```

10.6.5. Отмена выполнения задач

Действие экземпляра Future можно отменить, если он был предоставлен, но не запущен, вызвав его метод cancel().

Листинг 10.115. futures_future_callback_cancel.py

```
from concurrent import futures
import time


def task(n):
    print('{}: sleeping'.format(n))
    time.sleep(0.5)
    print('{}: done'.format(n))
    return n / 10


def done(fn):
    if fn.cancelled():
        print('{}: canceled'.format(fn.arg))
    elif fn.done():
        print('{}: not canceled'.format(fn.arg))


if __name__ == '__main__':
    ex = futures.ThreadPoolExecutor(max_workers=2)
    print('main: starting')
    tasks = []

    for i in range(10, 0, -1):
        print('main: submitting {}'.format(i))
        f = ex.submit(task, i)
        f.arg = i
        f.add_done_callback(done)
        tasks.append((i, f))

    for i, t in reversed(tasks):
        if not t.cancel():
            print('main: did not cancel {}'.format(i))

    ex.shutdown()
```

Метод cancel() возвращает булево значение, указывающее на то, была ли задача отменена.

```
$ python3 futures_future_callback_cancel.py

main: starting
main: submitting 10
10: sleeping
main: submitting 9
9: sleeping
main: submitting 8
```

```
main: submitting 7
main: submitting 6
main: submitting 5
main: submitting 4
main: submitting 3
main: submitting 2
main: submitting 1
1: canceled
2: canceled
3: canceled
4: canceled
5: canceled
6: canceled
7: canceled
8: canceled
main: did not cancel 9
main: did not cancel 10
10: done
10: not canceled
9: done
9: not canceled
```

10.6.6. Возбуждение исключений в задачах

Если задача генерирует необрабатываемое исключение, то оно сохраняется в экземпляре `Future` для данной задачи и становится доступным через вызов метода `result()` или `exception()`.

Листинг 10.116. futures_future_exception.py

```
from concurrent import futures


def task(n):
    print('{}: starting'.format(n))
    raise ValueError('the value {} is no good'.format(n))


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
f = ex.submit(task, 5)

error = f.exception()
print('main: error: {}'.format(error))

try:
    result = f.result()
except ValueError as e:
    print('main: saw error "{}" when accessing result'.format(e))
```

Если метод `result()` вызывается после возбуждения необрабатываемого исключения в функции задачи, то же самое исключение возбуждается в текущем контексте.

```
$ python3 futures_future_exception.py

main: starting
5: starting
main: error: the value 5 is no good
main: saw error "the value 5 is no good" when accessing result
```

10.6.7. Менеджер контекста

Объекты-исполнители работают как менеджеры контекста, выполняя задачи в параллельном режиме и ожидая, пока все они не завершатся. Если менеджер контекста существует, то вызывается метод `shutdown()` объекта-исполнителя.

Листинг 10.117. `futures_context_manager.py`

```
from concurrent import futures


def task(n):
    print(n)


with futures.ThreadPoolExecutor(max_workers=2) as ex:
    print('main: starting')
    ex.submit(task, 1)
    ex.submit(task, 2)
    ex.submit(task, 3)
    ex.submit(task, 4)

print('main: done')
```

Этот режим использования объекта-исполнителя может быть полезен, если ресурсы потока или процесса должны освобождаться, когда поток управления покидает текущую область видимости.

```
$ python3 futures_context_manager.py

main: starting
1
2
3
4
main: done
```

10.6.8. Пулы процессов

Экземпляры `ProcessPoolExecutor` работают аналогично экземплярам `ThreadPoolExecutor`, но используют процессы вместо потоков. Этот подход позволяет интенсивным вычислительным операциям использовать отдельные CPU, не подвергаясь глобальной блокировке интерпретатора CPython.

Листинг 10.118. futures_process_pool_map.py

```
from concurrent import futures
import os


def task(n):
    return (n, os.getpid())


ex = futures.ProcessPoolExecutor(max_workers=2)
results = ex.map(task, range(5, 0, -1))
for n, pid in results:
    print('ran task {} in process {}'.format(n, pid))
```

Как и в случае пула потоков, отдельные рабочие процессы повторно используются для выполнения множества задач.

```
$ python3 futures_process_pool_map.py

ran task 5 in process 60245
ran task 4 in process 60246
ran task 3 in process 60245
ran task 2 in process 60245
ran task 1 in process 60245
```

Если с одним из рабочих процессов происходит нечто, что приводит к его непредвиденному завершению, то работа экземпляра `ProcessPoolExecutor` считается прерванной, и он больше не используется для планирования выполнения задач.

Листинг 10.119. futures_process_pool_broken.py

```
from concurrent import futures
import os
import signal


with futures.ProcessPoolExecutor(max_workers=2) as ex:
    print('getting the pid for one worker')
    f1 = ex.submit(os.getpid)
    pid1 = f1.result()

    print('killing process {}'.format(pid1))
    os.kill(pid1, signal.SIGHUP)

    print('submitting another task')
    f2 = ex.submit(os.getpid)
    try:
        pid2 = f2.result()
    except futures.process.BrokenProcessPool as e:
        print('could not start new tasks: {}'.format(e))
```

Фактическое возбуждение исключения `BrokenProcessPool` происходит при обработке результатов, а не при предоставлении новой задачи.

```
$ python3 futures_process_pool_broken.py

getting the pid for one worker
killing process 62059
submitting another task
could not start new tasks: A process in the process pool was
terminated abruptly while the future was running or pending.
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `concurrent.futures`[23].
- **PEP 3148**[24]. *The proposal for creating the concurrent.futures feature set.*
- Раздел 10.5.14.
- `threading` (раздел 10.3).
- `multiprocessing` (раздел 10.4).

[23] https://docs.python.org/3.5/library/concurrent.futures.html

[24] www.python.org/dev/peps/pep-3148

Глава 11

Обмен данными по сети

Взаимодействие по сети используется с целью получения данных, необходимых для локального выполнения алгоритмов, совместного использования информации для распределенных вычислений и управления облачными службами. Стандартная библиотека Python комплектуется модулями, необходимыми как для создания сетевых служб, так и для удаленного доступа к существующим службам.

Модуль ipaddress (раздел 11.1) включает классы, обеспечивающие проверку, сравнение и другие операции над сетевыми адресами IPv4 и IPv-.

Низкоуровневая библиотека socket (раздел 11.2) предоставляет непосредственный доступ к библиотеке socket языка C и может использоваться для взаимодействия с любой сетевой службой. Модуль selectors (раздел 11.3) предоставляет высокоуровневый интерфейс для одновременного наблюдения за несколькими сокетами и обеспечивает возможность взаимодействия сетевых серверов с несколькими клиентами одновременно. Модуль select (раздел 11.4) предоставляет низкоуровневые программные интерфейсы, используемые модулем selectors (раздел 11.3).

Фреймворки, входящие в состав модуля socketserver (раздел 11.5), абстрагируют значительную часть рутинной работы, которую приходится выполнять при создании нового сетевого сервера. Комбинируя предоставляемые ими классы, можно создавать серверы, порождающие или использующие потоки и поддерживающие протоколы TCP и UDP. Приложению остается позаботиться лишь о фактической обработке сообщений.

11.1. ipaddress: интернет-адреса

Модуль ipaddress включает классы для работы с сетевыми адресами, соответствующими протоколам IPv4 и IPv6. Эти классы поддерживают проверку и поиск адресов и хостов в сети, а также другие часто используемые операции.

11.1.1. Адреса

Простейшим объектом является собственно сетевой адрес. Чтобы создать сетевой адрес, следует передать функции ip_address() аргумент в виде строки, целого числа или последовательности байтов. В зависимости от типа используемого адреса эта функция возвращает экземпляр IPv4Address или IPv6Address.

Листинг 11.1. ipaddress_addresses.py

```
import binascii
import ipaddress


ADDRESSES = [
```

```
    '10.9.0.6',
    'fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa',
]

for ip in ADDRESSES:
    addr = ipaddress.ip_address(ip)
    print('{!r}'.format(addr))
    print('   IP version:', addr.version)
    print('   is private:', addr.is_private)
    print('  packed form:', binascii.hexlify(addr.packed))
    print('      integer:', int(addr))
    print()
```

Оба класса обеспечивают различные представления адреса для различных целей и позволяют использовать утверждения с целью проверки того, зарезервирован ли адрес для многоадресатного вещания, принадлежит ли он частной сети и т.д.

```
$ python3 ipaddress_addresses.py

IPv4Address('10.9.0.6')
   IP version: 4
   is private: True
  packed form: b'0a090006'
      integer: 168361990

IPv6Address('fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa')
   IP version: 6
   is private: True
  packed form: b'fdfd87b5b4755e3eb1bce121a8eb14aa'
      integer: 337611086560236126439725644408160982186
```

11.1.2. Сети

Сеть определяется диапазоном адресов. Обычно диапазон адресов записывается в виде базового адреса и маски сети, указывающей, какие части адреса представляют сеть, а какие — адреса хостов в этой сети. Маска может быть выражена либо явным образом, либо с использованием системы обозначений "префикс/число битов", как показано в следующем примере.

Листинг 11.2. `ipaddress_networks.py`

```
import ipaddress

NETWORKS = [
    '10.9.0.0/24',
    'fdfd:87b5:b475:5e3e::/64',
]

for n in NETWORKS:
    net = ipaddress.ip_network(n)
    print('{!r}'.format(net))
```

```
    print('     is private:', net.is_private)
    print('      broadcast:', net.broadcast_address)
    print('     compressed:', net.compressed)
    print('   with netmask:', net.with_netmask)
    print('  with hostmask:', net.with_hostmask)
    print('  num addresses:', net.num_addresses)
    print()
```

Что касается адресов, то существуют два сетевых класса, которые соответствуют сетям IPv4 и IPv6. Каждый из них предоставляет свойства или методы для доступа к значениям, относящимся к сети, таким как широковещательный адрес и адреса в сети, которые можно использовать для хостов.

```
$ python3 ipaddress_networks.py

IPv4Network('10.9.0.0/24')
     is private: True
      broadcast: 10.9.0.255
     compressed: 10.9.0.0/24
   with netmask: 10.9.0.0/255.255.255.0
  with hostmask: 10.9.0.0/0.0.0.255
  num addresses: 256

IPv6Network('fdfd:87b5:b475:5e3e::/64')
     is private: True
      broadcast: fdfd:87b5:b475:5e3e:ffff:ffff:ffff:ffff
     compressed: fdfd:87b5:b475:5e3e::/64
   with netmask: fdfd:87b5:b475:5e3e::/ffff:ffff:ffff:ffff::
  with hostmask: fdfd:87b5:b475:5e3e::/::ffff:ffff:ffff:ffff
  num addresses: 18446744073709551616
```

Экземпляр сети является итерируемым объектом, который возвращает сетевые адреса.

Листинг 11.3. ipaddress_network_iterate.py

```
import ipaddress

NETWORKS = [
    '10.9.0.0/24',
    'fdfd:87b5:b475:5e3e::/64',
]

for n in NETWORKS:
    net = ipaddress.ip_network(n)
    print('{!r}'.format(net))
    for i, ip in zip(range(3), net):
        print(ip)
    print()
```

В этом примере выводятся не все адреса, поскольку полное количество адресов сети IPv6 настолько велико, что их вывод был бы слишком длинным.

```
$ python3 ipaddress_network_iterate.py

IPv4Network('10.9.0.0/24')
10.9.0.0
10.9.0.1
10.9.0.2

IPv6Network('fdfd:87b5:b475:5e3e::/64')
fdfd:87b5:b475:5e3e::
fdfd:87b5:b475:5e3e::1
fdfd:87b5:b475:5e3e::2
```

Итерирование по экземпляру сети возвращает адреса, но не все они являются корректными адресами хостов. Например, в число этих адресов входят базовый и широковещательный адреса сети. Для нахождения адресов, которые могут использоваться обычными сетевыми хостами, следует использовать метод `hosts()`, возвращающий генератор.

Листинг 11.4. `ipaddress_network_iterate_hosts.py`

```
import ipaddress

NETWORKS = [
    '10.9.0.0/24',
    'fdfd:87b5:b475:5e3e::/64',
]

for n in NETWORKS:
    net = ipaddress.ip_network(n)
    print('{!r}'.format(net))
    for i, ip in zip(range(3), net.hosts()):
        print(ip)
    print()
```

Сравнение с выводом предыдущего примера показывает, что адреса хостов не включают первые значения, получаемые при итерировании по всей сети.

```
$ python3 ipaddress_network_iterate_hosts.py

IPv4Network('10.9.0.0/24')
10.9.0.1
10.9.0.2
10.9.0.3

IPv6Network('fdfd:87b5:b475:5e3e::/64')
fdfd:87b5:b475:5e3e::1
fdfd:87b5:b475:5e3e::2
fdfd:87b5:b475:5e3e::3
```

Кроме протокола итератора экземпляры сети поддерживают оператор `in`, позволяющий тестировать принадлежность адреса к данной сети.

Листинг 11.5. ipaddress_network_membership.py

```
import ipaddress


NETWORKS = [
    ipaddress.ip_network('10.9.0.0/24'),
    ipaddress.ip_network('fdfd:87b5:b475:5e3e::/64'),
]

ADDRESSES = [
    ipaddress.ip_address('10.9.0.6'),
    ipaddress.ip_address('10.7.0.31'),
    ipaddress.ip_address(
        'fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa'
    ),
    ipaddress.ip_address('fe80::3840:c439:b25e:63b0'),
]


for ip in ADDRESSES:
    for net in NETWORKS:
        if ip in net:
            print('{}\nis on {}'.format(ip, net))
            break
    else:
        print('{}\nis not on a known network'.format(ip))
    print()
```

Для тестирования адреса реализация оператора `in` использует маску сети, что гораздо эффективнее использования полного списка сетевых адресов.

```
$ python3 ipaddress_network_membership.py

10.9.0.6
is on 10.9.0.0/24

10.7.0.31
is not on a known network

fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa
is on fdfd:87b5:b475:5e3e::/64

fe80::3840:c439:b25e:63b0
is not on a known network
```

11.1.3. Интерфейсы

Сетевой интерфейс является особым сетевым адресом и может быть представлен в виде адреса хоста и сетевого префикса или маски сети.

Листинг 11.6. ipaddress_interfaces.py

```
import ipaddress


ADDRESSES = [
    '10.9.0.6/24',
    'fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa/64',
]


for ip in ADDRESSES:
    iface = ipaddress.ip_interface(ip)
    print('{!r}'.format(iface))
    print('network:\n  ', iface.network)
    print('ip:\n  ', iface.ip)
    print('IP with prefixlen:\n  ', iface.with_prefixlen)
    print('netmask:\n  ', iface.with_netmask)
    print('hostmask:\n  ', iface.with_hostmask)
    print()
```

Объект интерфейса имеет свойства, обеспечивающие доступ к полной сети и адресу по отдельности, а также поддерживает несколько различных способов представления адреса и маски сети.

```
IPv4Interface('10.9.0.6/24')
network:
   10.9.0.0/24
ip:
   10.9.0.6
IP with prefixlen:
   10.9.0.6/24
netmask:
   10.9.0.6/255.255.255.0
hostmask:
   10.9.0.6/0.0.0.255

IPv6Interface('fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa/64')
network:
   fdfd:87b5:b475:5e3e::/64
ip:
   fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa
IP with prefixlen:
   fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa/64
netmask:
   fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa/ffff:ffff:ffff:ffff::
hostmask:
   fdfd:87b5:b475:5e3e:b1bc:e121:a8eb:14aa/::ffff:ffff:ffff:ffff
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `ipaddress`[1].
- **PEP 3144**[2]. *IP Address Manipulation Library for the Python Standard Library*.
- *An introduction to the ipaddress module*[3].
- Википедия: *IP-адрес*[4]. Введение в сети и сетевые адреса.
- *Computer Networks, Fifth Edition*, by Andrew S. Tanenbaum and David J. Wetherall. Pearson, 2010. ISBN-10: 0132126958.

11.2. `socket`: сетевое взаимодействие

Модуль `socket` предоставляет низкоуровневый API языка C для взаимодействия по сети с использованием сокетов BSD. Он включает класс `socket`, предназначенный для работы с фактическим каналом передачи данных, а также функции для решения таких задач, связанных с сетями, как преобразование имени сервера в адрес и форматирование данных, передаваемых по сети.

11.2.1. Адресация, семейства протоколов и типы сокетов

Сокет — это конечная точка соединения, используемого программами как для локального обмена данными между процессами, так и для обмена через Интернет. За управление процессом передачи данных отвечают два основных свойства сокетов: *семейство адресов*, задающее используемый сетевой протокол модели OSI, и *тип сокета*, соответствующий протоколу транспортного уровня.

Python поддерживает три семейства адресов. Наиболее распространенное из них — `AF_INET`, которое используется для IPv4-адресации в Интернете. Адреса IPv4 имеют длину 4 байта и обычно представляются в виде последовательности из четырех чисел, по одному на октет, разделенных точками (например, 10.1.1.5 или 127.0.0.1). Именно такие значения чаще всего имеют в виду, когда говорят об IP-адресах. В настоящее время почти вся работа в Интернете ведется с использованием адресов IPv4.

Семейство адресов `AF_INET6` используется для IPv6-адресации в Интернете. Протокол IPv6 — это следующее поколение интернет-протокола, обеспечивающее поддержку 128-битовых адресов, шейпинг трафика и средства маршрутизации, недоступные в протоколе IPv4. Внедрение IPv6-адресации идет ускоренными темпами, особенно с появлением облачных технологий и добавлением в сеть новых устройств в рамках проектов *интернета вещей*.

Семейство адресов `AF_UNIX` применяется при использовании *сокетов домена Unix* (Unix Domain Sockets — UDS) — протокола межпроцессного взаимодействия, доступного в POSIX-совместимых системах. В типичных случаях реализация UDS позволяет операционной системе передавать данные непосредственно из процесса в процесс без прохождения сетевого стека. Такой подход обеспечивает большую эффективность по сравнению с использованием `AF_INET`, но ввиду того, что

[1] `https://docs.python.org/3.5/library/ipaddress.html`

[2] `www.python.org/dev/peps/pep-3144`

[3] `https://docs.python.org/3.5/howto/ipaddress.html#ipaddress-howto`

[4] `https://ru.wikipedia.org/wiki/IP_адрес`

в качестве пространства имен для адресации используется файловая система, применение UDS ограничено процессами, выполняющимися в одной и той же системе. Привлекательность использования UDS по сравнению с другими механизмами межпроцессного взаимодействия, такими как именованные каналы или разделяемая память, заключается в том, что программный интерфейс остается тем же, что и при работе с IP-сетями, благодаря чему приложение может воспользоваться всеми преимуществами эффективной передачи данных при выполнении программы на отдельном хосте, но при этом использовать тот же код для передачи данных по сети.

Примечание

Константа `AF_UNIX` определена лишь в системах, поддерживающих UDS.

Обычно типом сокета является либо `SOCK_DGRAM`, предназначенный для передачи датаграмм без предварительного установления соединения, либо `SOCK_STREAM`, предназначенный для передачи потока байтов с поддержкой логического соединения. Сокеты, работающие с датаграммами, чаще всего ассоциируются с *протоколом пользовательских датаграмм* (User Datagram Protocol – UDP). Сокеты этого типа не в состоянии обеспечить высокую надежность передачи данных. Потокоориентированные сокеты типа `SOCK_STREAM` ассоциируются с *протоколом управления передачей* (Transport Control Protocol – TCP). Они обеспечивают передачу байтовых потоков между сервером и клиентом и гарантируют доставку сообщений или уведомлений о неудачной попытке доставки, используя управление тайм-аутами, повторную передачу данных и другие возможности.

Большинство протоколов прикладного уровня, обеспечивающих доставку больших объемов данных, такие как HTTP, располагаются в стеке поверх протокола TCP, поскольку создание сложных приложений упрощается, если упорядочение и доставка сообщений осуществляются автоматически. Обычно UDP используют при работе с протоколами, в которых порядок получения сообщений не особенно существен (поскольку каждое сообщение является самодостаточным и имеет небольшой размер, как, например, в случае получения имени хоста с помощью службы DNS), а также при *многоадресатном вещании* (когда одни и те же данные передаются нескольким хостам). Оба протокола, UDP и TCP, могут использоваться как с IPv4-, так и с IPv6-адресацией.

Примечание

Модуль `socket` в Python поддерживает и другие типы сокетов, но они используются не так часто и поэтому здесь не рассматриваются. Для получения более подробной информации по этому вопросу обратитесь к документации стандартной библиотеки.

11.2.1.1. Поиск хостов в сети

Модуль `socket` включает функции, образующие программный интерфейс к сетевым службам доменных имен, благодаря чему программа может преобразовать имя серверного хоста в числовой сетевой адрес. Приложениям не нужно выполнять явное преобразование адресов до того, как они будут использованы для установления соединения с сервером. Тем не менее при выводе сообщений об ошибках в них целесообразно включать как числовой адрес, так и используемое имя сервера.

Имя хоста локального компьютера можно определить с помощью функции gethostname().

Листинг 11.7. socket_gethostname.py

```
import socket

print(socket.gethostname())
```

Возвращаемое имя зависит от сетевых настроек текущей системы и может изменяться при подключении к другой сети (например, при подключении ноутбука к беспроводной сети).

```
$ python3 socket_gethostname.py

apu.hellfly.net
```

Функция gethostbyname() позволяет запросить у операционной системы преобразование имени сервера в его числовой адрес.

Листинг 11.8. socket_gethostbyname.py

```
import socket

HOSTS = [
    'apu',
    'pymotw.com',
    'www.python.org',
    'nosuchname',
]

for host in HOSTS:
    try:
        print('{} : {}'.format(host, socket.gethostbyname(host)))
    except socket.error as msg:
        print('{} : {}'.format(host, msg))
```

Если DNS-конфигурация локальной системы включает в область поиска один или несколько доменов, то аргумент, представляющий имя, не обязан быть полным (уточненным) именем (т.е. не обязан включать имя домена наряду с базовым именем хоста). Если указанное имя не удастся найти, возбуждается исключение socket.error.

```
$ python3 socket_gethostbyname.py

apu : 10.9.0.10
pymotw.com : 66.33.211.242
www.python.org : 151.101.32.223
nosuchname : [Errno 8] nodename nor servname provided, or not
known
```

Функция gethostbyname_ex() позволяет получить более полную информацию о сервере. Она возвращает основное имя сервера, список альтернативных

имен хоста (псевдонимов), а также список всех доступных адресов, которые можно использовать для подключения к этому серверу.

Листинг 11.9. socket_gethostbyname_ex.py

```
import socket

HOSTS = [
    'apu',
    'pymotw.com',
    'www.python.org',
    'nosuchname',
]

for host in HOSTS:
    print(host)
    try:
        name, aliases, addresses = socket.gethostbyname_ex(host)
        print('  Hostname:', name)
        print('  Aliases :', aliases)
        print(' Addresses:', addresses)
    except socket.error as msg:
        print('ERROR:', msg)
    print()
```

Знание всех известных IP-адресов сервера позволяет клиенту реализовывать собственные отказоустойчивые или балансирующие нагрузку алгоритмы.

```
$ python3 socket_gethostbyname_ex.py

apu
  Hostname: apu.hellfly.net
  Aliases : ['apu']
 Addresses: ['10.9.0.10']


pymotw.com
  Hostname: pymotw.com
  Aliases : []
 Addresses: ['66.33.211.242']


www.python.org
  Hostname: prod.python.map.fastlylb.net
  Aliases : ['www.python.org', 'python.map.fastly.net']
 Addresses: ['151.101.32.223']


nosuchname
ERROR: [Errno 8] nodename nor servname provided, or not known
```

Функция `getfqdn()` преобразует неполное имя домена в уточненное имя.

Листинг 11.10. socket_getfqdn.py

```
import socket

for host in ['apu', 'pymotw.com']:
    print('{:>10} : {}'.format(host, socket.getfqdn(host)))
```

Возвращенное имя не обязательно будет совпадать с входным аргументом, если он является псевдонимом.

```
$ python3 socket_getfqdn.py

       apu : apu.hellfly.net
pymotw.com : apache2-echo.catalina.dreamhost.com
```

Если известен адрес сервера, то функция `gethostbyaddr()` позволяет выполнить обратный поиск для нахождения имени.

Листинг 11.11. socket_gethostbyaddr.py

```
import socket

hostname, aliases, addresses = socket.gethostbyaddr('10.9.0.10')

print('Hostname :', hostname)
print('Aliases  :', aliases)
print('Addresses:', addresses)
```

Возвращаемое значение представляет собой кортеж, содержащий полное имя хоста, список его альтернативных имен и список IP-адресов, связанных с этим именем.

```
$ python3 socket_gethostbyaddr.py

Hostname : apu.hellfly.net
Aliases  : ['apu']
Addresses: ['10.9.0.10']
```

11.2.1.2. Получение информации о сетевых службах

Помимо IP-адреса каждый адрес сокета включает целочисленный номер порта. Многие приложения могут выполняться на одном и том же компьютере, прослушивая один и тот же IP-адрес, но в каждый момент времени только один сокет может использовать порт по этому адресу. Комбинация IP-адреса, протокола и номера порта однозначно идентифицирует канал передачи данных и гарантирует, что сообщения, отправленные через сокет, будут корректно доставлены в пункт назначения.

Некоторые номера портов предопределены для конкретных протоколов. Например, связь между почтовыми серверами посредством протокола SMTP осуществляется через порт 25 с использованием протокола TCP, а веб-клиенты и веб-серверы используют порт 80 для HTTP. Номера портов для сетевых служб, имеющих стандартизированные имена, можно найти с помощью функции `getservbyname()`.

Листинг 11.12. socket_getservbyname.py

```
import socket
from urllib.parse import urlparse

URLS = [
    'http://www.python.org',
    'https://www.mybank.com',
    'ftp://prep.ai.mit.edu',
    'gopher://gopher.micro.umn.edu',
    'smtp://mail.example.com',
    'imap://mail.example.com',
    'imaps://mail.example.com',
    'pop3://pop.example.com',
    'pop3s://pop.example.com',
]

for url in URLS:
    parsed_url = urlparse(url)
    port = socket.getservbyname(parsed_url.scheme)
    print('{:>6} : {}'.format(parsed_url.scheme, port))
```

Несмотря на то что вероятность изменения номера порта, используемого стандартизированной службой, весьма мала, использование системных вызовов для определения номера порта обеспечивает более гибкое решение в том случае, если в будущем планируется добавление новых служб.

```
$ python3 socket_getservbyname.py

  http : 80
 https : 443
   ftp : 21
gopher : 70
  smtp : 25
  imap : 143
 imaps : 993
  pop3 : 110
 pop3s : 995
```

Чтобы выполнить обратный поиск по номерам портов, следует использовать функцию `getservbyport()`.

Листинг 11.13. socket_getservbyport.py

```
import socket
from urllib.parse import urlunparse

for port in [80, 443, 21, 70, 25, 143, 993, 110, 995]:
    url = '{}://example.com/'.format(socket.getservbyport(port))
    print(url)
```

Обратный поиск полезен для конструирования URL-адресов на основе произвольных адресов.

```
$ python3 socket_getservbyport.py

http://example.com/
https://example.com/
ftp://example.com/
gopher://example.com/
smtp://example.com/
imap://example.com/
imaps://example.com/
pop3://example.com/
pop3s://example.com/
```

Для получения номера порта, назначенного транспортному протоколу, следует использовать функцию `getprotobyname()`.

Листинг 11.14. socket_getprotobyname.py

```
import socket


def get_constants(prefix):
    """Создать словарь, сопоставляющий константы
    модуля socket с их именами.
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
    }


protocols = get_constants('IPPROTO_')

for name in ['icmp', 'udp', 'tcp']:
    proto_num = socket.getprotobyname(name)
    const_name = protocols[proto_num]
    print('{:>4} -> {:2d} (socket.{:<12} = {:2d})'.format(
        name, proto_num, const_name,
        getattr(socket, const_name)))
```

Значения номеров портов для протоколов стандартизированы и определены в модуле `socket` в виде констант с префиксом `IPPROTO_`.

```
$ python3 socket_getprotobyname.py

icmp ->  1 (socket.IPPROTO_ICMP =  1)
 udp -> 17 (socket.IPPROTO_UDP  = 17)
 tcp ->  6 (socket.IPPROTO_TCP  =  6)
```

11.2.1.3. Получение информации об адресе сервера

Функция `getaddrinfo()` преобразует базовый адрес сервера в список кортежей, содержащих всю необходимую информацию для установления соединения. Каждый кортеж может содержать различные семейства адресов или протоколы.

Листинг 11.15. `socket_getaddrinfo.py`

```
import socket


def get_constants(prefix):
    """Создать словарь, сопоставляющий константы
    модуля socket с их именами.
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
    }


families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')

for response in socket.getaddrinfo('www.python.org', 'http'):

    # Распаковка кортежа ответа
    family, socktype, proto, canonname, sockaddr = response

    print('Family        :', families[family])
    print('Type          :', types[socktype])
    print('Protocol      :', protocols[proto])
    print('Canonical name:', canonname)
    print('Socket address:', sockaddr)
    print()
```

Этот пример демонстрирует, как получить информацию о возможных соединениях с сервером `www.python.org`.

```
$ python3 socket_getaddrinfo.py

Family        : AF_INET
Type          : SOCK_DGRAM
Protocol      : IPPROTO_UDP
Canonical name:
Socket address: ('151.101.32.223', 80)

Family        : AF_INET
Type          : SOCK_STREAM
Protocol      : IPPROTO_TCP
Canonical name:
Socket address: ('151.101.32.223', 80)
```

```
Family        : AF_INET6
Type          : SOCK_DGRAM
Protocol      : IPPROTO_UDP
Canonical name:
Socket address: ('2a04:4e42:8::223', 80, 0, 0)

Family        : AF_INET6
Type          : SOCK_STREAM
Protocol      : IPPROTO_TCP
Canonical name:
Socket address: ('2a04:4e42:8::223', 80, 0, 0)
```

Функция `getaddrinfo()` получает несколько аргументов, обеспечивающих фильтрацию результирующего списка. Значения параметров `host` и `port`, представленные в примере, являются обязательными аргументами. Необязательные аргументы — `family`, `socktype`, `proto` и `flags`. Значением необязательных аргументов может быть `0` или одна из констант, определенных в модуле `socket`.

Листинг 11.16. socket_getaddrinfo_extra_args.py

```
import socket


def get_constants(prefix):
    """Создать словарь, сопоставляющий константы
    модуля socket с их именами.
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
    }


families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')

responses = socket.getaddrinfo(
    host='www.python.org',
    port='http',
    family=socket.AF_INET,
    type=socket.SOCK_STREAM,
    proto=socket.IPPROTO_TCP,
    flags=socket.AI_CANONNAME,
)

for response in responses:
    # Распаковка кортежа ответа
    family, socktype, proto, canonname, sockaddr = response

    print('Family        :', families[family])
```

```
    print('Type          :', types[socktype])
    print('Protocol      :', protocols[proto])
    print('Canonical name:', canonname)
    print('Socket address:', sockaddr)
    print()
```

Поскольку аргумент `flags` включает флаг `AI_CANONNAME`, то на этот раз основное (каноническое) имя сервера, которое может отличаться от значения, используемого для поиска при наличии альтернативных имен, включается в выводимые результаты. Без этого флага поле "Canonical name" осталось бы пустым.

```
$ python3 socket_getaddrinfo_extra_args.py

Family        : AF_INET
Type          : SOCK_STREAM
Protocol      : IPPROTO_TCP
Canonical name: prod.python.map.fastlylb.net
Socket address: ('151.101.32.223', 80)
```

11.2.1.4. Представление IP-адресов

Сетевые программы, написанные на языке C, используют тип данных `struct sockaddr` для представления IP-адресов в виде двоичных значений (а не в виде строк, обычно применяемых для этой цели в программах на языке Python). Для выполнения взаимных преобразований адресов IPv4 между их представлениями в Python и C следует использовать функции `inet_aton()` и `inet_ntoa()`.

Листинг 11.17. `socket_address_packing.py`

```
import binascii
import socket
import struct
import sys

for string_address in ['192.168.1.1', '127.0.0.1']:
    packed = socket.inet_aton(string_address)
    print('Original:', string_address)
    print('Packed  :', binascii.hexlify(packed))
    print('Unpacked:', socket.inet_ntoa(packed))
    print()
```

Четыре байта в упакованном формате могут использоваться библиотеками C, безопасно передаваться по сети или компактно храниться в базе данных.

```
$ python3 socket_address_packing.py
Original: 192.168.1.1
Packed  : b'c0a80101'
Unpacked: 192.168.1.1

Original: 127.0.0.1
Packed  : b'7f000001'
Unpacked: 127.0.0.1
```

Родственные функции `inet_pton()` и `inet_ntop()` работают как с адресами IPv4, так и с адресами IPv6, создавая подходящий выходной формат на основании переданного значения аргумента `family`.

Листинг 11.18. `socket_ipv6_address_packing.py`

```
import binascii
import socket
import struct
import sys

string_address = '2002:ac10:10a:1234:21e:52ff:fe74:40e'
packed = socket.inet_pton(socket.AF_INET6, string_address)

print('Original:', string_address)
print('Packed  :', binascii.hexlify(packed))
print('Unpacked:', socket.inet_ntop(socket.AF_INET6, packed))
```

Адрес IPv6 является уже шестнадцатеричным значением, поэтому в результате преобразования упакованной версии в серию шестнадцатеричных чисел получается строка, похожая на исходное значение.

```
$ python3 socket_ipv6_address_packing.py

Original: 2002:ac10:10a:1234:21e:52ff:fe74:40e
Packed  : b'2002ac10010a1234021e52fffe74040e'
Unpacked: 2002:ac10:10a:1234:21e:52ff:fe74:40e
```

Дополнительные ссылки

- Википедия: *IPv6*[5]. Обсуждение IPv6 (версии 6 интернет-протокола).
- Википедия: *Сетевая модель OSI*[6]. Описание семиуровневой модели сетевого стека протоколов.
- *Assigned Internet Protocol Numbers*[7]. Список имен и номеров стандартных протоколов.

11.2.2. Клиент и сервер TCP/IP

В зависимости от того, как сконфигурирован сокет, он может функционировать в качестве сервера, слушая входящие сообщения, или в качестве клиента, устанавливая соединение с другим приложением. После того как между обеими конечными точками устанавливается соединение, канал связи становится двусторонним.

11.2.2.1. Эхо-сервер

Представленная ниже программа, которая основана на одном из примеров, приведенных в документации стандартной библиотеки, получает входящие сообщения и посылает их обратно отправителю. Вначале программа создает сокет

[5] https://ru.wikipedia.org/wiki/IPv6

[6] https://ru.wikipedia.org/wiki/Сетевая_модель_OSI

[7] www.iana.org/assignments/protocol-numbers/protocol-numbers.xml

TCP/IP, а затем связывает сокет с адресом сервера с помощью метода `bind()`. В данном случае используется адрес сервера `localhost`, ссылающийся на локальный хост, и порт `10000`.

Листинг 11.19. `socket_echo_server.py`

```
import socket
import sys

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Привязать сокет к прослушиваемому порту
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address))
sock.bind(server_address)

# Слушать входящие соединения
sock.listen(1)

while True:
    # Ждать соединения
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('connection from', client_address)

        # Получать данные небольшими порциями и
        # отправлять их обратно
        while True:
            data = connection.recv(16)
            print('received {!r}'.format(data))
            if data:
                print('sending data back to the client')
                connection.sendall(data)
            else:
                print('no data from', client_address)
                break

    finally:
        # Закрыть соединение
        connection.close()
```

Вызов метода `listen()` переводит сокет в режим сервера, а вызов метода `accept()` блокирует сокет до получения входящего соединения. Целочисленный аргумент задает количество соединений, которые система должна принимать в очередь в фоновом режиме, прежде чем отвергать новых клиентов. В данном случае программа работает не более чем с одним клиентом одновременно.

Метод `accept()` возвращает открытое соединение между сервером и клиентом вместе с адресом клиента. На самом деле это соединение осуществляется через другой сокет с использованием другого порта (назначаемого ядром). Данные

читаются из соединения с помощью метода `recv()` и передаются клиенту с помощью метода `sendall()`.

Если связь с клиентом прекращается, то соединение должно освободить неиспользуемые ресурсы с помощью метода `close()`. В данном примере вызов метода `close()`, даже в случае возникновения ошибки, гарантируется использованием блока `try:finally`.

11.2.2.2. Эхо-клиент

Клиентская программа настраивает свой сокет иначе, чем серверная. Вместо привязки к порту и его прослушивания она использует метод `connect()` для непосредственного подключения сокета к удаленному адресу.

Листинг 11.20. `socket_echo_client.py`

```
import socket
import sys

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Подключить сокет к порту, который прослушивается сервером
server_address = ('localhost', 10000)
print('connecting to {} port {}'.format(*server_address))
sock.connect(server_address)

try:

    # Отправить данные
    message = b'This is the message.  It will be repeated.'
    print('sending {!r}'.format(message))
    sock.sendall(message)

    # Ждать ответа
    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data))

finally:
    print('closing socket')
    sock.close()
```

После установления соединения клиент может отправлять данные через сокет с помощью метода `sendall()` и получать ответы с помощью метода `recv()`, точно так же, как и в случае сервера. После отправки всего сообщения и получения его копии сокет закрывается, освобождая порт.

11.2.2.3. Взаимодействие клиента и сервера

Чтобы клиент и сервер могли общаться между собой, они должны выполняться в разных окнах терминала. В окне сервера отображаются входящее соединение и полученные данные, а также ответ, посылаемый обратно клиенту.

```
$ python3 socket_echo_server.py
starting up on localhost port 10000
waiting for a connection
connection from ('127.0.0.1', 65141)
received b'This is the mess'
sending data back to the client
received b'age.  It will be'
sending data back to the client
received b' repeated.'
sending data back to the client
received b''
no data from ('127.0.0.1', 65141)
waiting for a connection
```

В окне клиента отображаются исходящее сообщение и ответ, полученный от сервера.

```
$ python3 socket_echo_client.py
connecting to localhost port 10000
sending b'This is the message.  It will be repeated.'
received b'This is the mess'
received b'age.  It will be'
received b' repeated.'
closing socket
```

11.2.2.4. Упрощенный способ подключения клиента

Клиенты TCP/IP могут сэкономить несколько операций, используя вспомогательную функцию `create_connection()` для соединения с сервером. Эта функция имеет один аргумент — кортеж из двух элементов, содержащий адрес сервера, — и определяет наилучший адрес, который можно использовать для соединения.

Листинг 11.21. `socket_echo_client_easy.py`

```
import socket
import sys


def get_constants(prefix):
    """Создать словарь, сопоставляющий константы
    модуля socket с их именами.
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
```

```
    }


families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')

# Создать сокет TCP/IP
sock = socket.create_connection(('localhost', 10000))

print('Family  :', families[sock.family])
print('Type    :', types[sock.type])
print('Protocol:', protocols[sock.proto])
print()

try:

    # Отправить данные
    message = b'This is the message.  It will be repeated.'
    print('sending {!r}'.format(message))
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data))

finally:
    print('closing socket')
    sock.close()
```

Функция `create_connection()` использует функцию `getaddrinfo()` для определения параметров соединения-кандидата и возвращает сокет, открытый с использованием первой конфигурации, которая приводит к успешному созданию соединения. Тип возвращенного сокета можно определить по значениям атрибутов `family`, `type` и `proto`.

```
$ python3 socket_echo_client_easy.py

Family  : AF_INET
Type    : SOCK_STREAM
Protocol: IPPROTO_IP

sending b'This is the message.  It will be repeated.'
received b'This is the mess'
received b'age.  It will be'
received b' repeated.'
closing socket
```

11.2.2.5. Выбор адреса для прослушивания

Очень важно привязать сервер к корректному адресу, чтобы клиенты могли общаться с ним. Во всех предыдущих примерах в качестве IP-адреса использовался адрес `'localhost'`, с которым могут соединяться только клиенты, выполняющиеся на том же сервере. Использование публичного адреса сервера, как тот, который возвращает функция `gethostname()`, позволяет связываться с ним другим хостам. В следующем примере демонстрируется видоизмененная версия эхо-сервера, который прослушивает адрес, задаваемый в качестве аргумента в командной строке.

Листинг 11.22. `socket_echo_server_explicit.py`

```
import socket
import sys

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Привязать сокет к адресу, указанному в командной строке
server_name = sys.argv[1]
server_address = (server_name, 10000)
print('starting up on {} port {}'.format(*server_address))
sock.bind(server_address)
sock.listen(1)

while True:
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('client connected:', client_address)
        while True:
            data = connection.recv(16)
            print('received {!r}'.format(data))
            if data:
                connection.sendall(data)
            else:
                break
    finally:
        connection.close()
```

Для тестирования сервера потребуется аналогичная модифицированная версия клиентской программы.

Листинг 11.23. `socket_echo_client_explicit.py`

```
import socket
import sys

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Подключить сокет к порту на сервере, указанном вызывающим кодом
server_address = (sys.argv[1], 10000)
```

```
print('connecting to {} port {}'.format(*server_address))
sock.connect(server_address)

try:

    message = b'This is the message.  It will be repeated.'
    print('sending {!r}'.format(message))
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)
    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data))

finally:
    sock.close()
```

Выполнение команды `netstat` после запуска сервера с аргументом `hubert` позволяет убедиться в прослушивании адреса указанного хоста.

```
$ host hubert.hellfly.net

hubert.hellfly.net has address 10.9.0.6

$ netstat -an | grep 10000

Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address      Foreign Address    (state)
...
tcp4       0      0  10.9.0.6.10000     *.*                LISTEN
...
```

Выполнение клиента на другом хосте с передачей ему `hubert.hellfly.net` в качестве адреса хоста, на котором выполняется сервер, приводит к следующим результатам.

```
$ hostname

apu

$ python3 ./socket_echo_client_explicit.py hubert.hellfly.net
connecting to hubert.hellfly.net port 10000
sending b'This is the message. It will be repeated.'
received b'This is the mess'
received b'age. It will be'
received b' repeated.'
```

На сервере выводятся следующие результаты.

```
$ python3 socket_echo_server_explicit.py hubert.hellfly.net
starting up on hubert.hellfly.net port 10000
```

```
waiting for a connection
client connected: ('10.9.0.10', 33139)
received b''
waiting for a connection
client connected: ('10.9.0.10', 33140)
received b'This is the mess'
received b'age. It will be'
received b' repeated.'
received b''
waiting for a connection
```

Многие серверы имеют более одного сетевого интерфейса, а значит, более чем один IP-адрес. Вместо того чтобы выполнять отдельные экземпляры службы, по одному на каждый IP-адрес, можно использовать специальный адрес `INADDR_ANY`, позволяющий прослушивать одновременно все адреса. Несмотря на то что константа `INADDR_ANY` определена в модуле `socket`, ее значение задано в виде целого числа, и это значение, прежде чем быть переданным методу `bind()`, необходимо преобразовать в строку адреса, для которого используется точечная нотация. Вместо выполнения этого преобразования используем адрес `0.0.0.0`, которому соответствует пустая строка (`''`).

Листинг 11.24. `socket_echo_server_any.py`

```
import socket
import sys

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Привязать сокет к адресу, указанному в командной строке
server_address = ('', 10000)
sock.bind(server_address)
print('starting up on {} port {}'.format(*sock.getsockname()))
sock.listen(1)

while True:
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('client connected:', client_address)
        while True:
            data = connection.recv(16)
            print('received {!r}'.format(data))
            if data:
                connection.sendall(data)
            else:
                break
    finally:
        connection.close()
```

Чтобы увидеть фактический адрес, используемый сокетом, следует вызвать его метод `getsockname()`. Вновь выполнив команду `netstat` после запуска серве-

ра, можно убедиться в том, что он прослушивает входящие соединения по любому из адресов.

```
$ netstat -an

Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address    Foreign Address   (state)
...
tcp4       0      0  *.10000          *.*               LISTEN
...
```

11.2.3. Клиент и сервер UDP

Протокол пользовательских датаграмм (UDP) работает не так, как TCP/IP. В то время как TCP — *потокоориентированный* протокол, гарантирующий получение данных в том порядке, в каком они были переданы, UDP ориентирован на *обработку сообщений*. С одной стороны, UDP не требует создания долговременных соединений, и поэтому настройка UDP-сокетов выполняется немного проще. С другой стороны, сообщения UDP должны умещаться в одной датаграмме (для IPv4 это означает, что они могут содержать не более 65507 байтов, поскольку пакет размером 65535 байтов включает также заголовок), и их доставка, в отличие от TCP, не гарантируется.

11.2.3.1. Эхо-сервер

Поскольку в данном случае соединение как таковое отсутствует, сервер не должен прослушивать и принимать входящие соединения. Вместо этого он просто использует метод `bind()` для привязки сокета к порту и ожидает поступления отдельных сообщений.

Листинг 11.25. `socket_echo_server_dgram.py`

```
import socket
import sys

# Создать сокет UDP
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Привязать сокет к порту
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address))
sock.bind(server_address)

while True:
    print('\nwaiting to receive message')
    data, address = sock.recvfrom(4096)

    print('received {} bytes from {}'.format(
        len(data), address))
    print(data)

    if data:
```

```
        sent = sock.sendto(data, address)
        print('sent {} bytes back to {}'.format(
            sent, address))
```

Сообщения читаются из сокета с помощью метода `recvfrom()`, который возвращает данные, а также адрес отправившего их клиента.

11.2.3.2. Эхо-клиент

Эхо-клиент UDP аналогичен серверу, но не использует метод `bind()` для привязки своего сокета к адресу. Он использует метод `sendto()` для непосредственной доставки сообщения серверу и метод `recvfrom()` для получения ответа.

Листинг 11.26. socket_echo_client_dgram.py

```
import socket
import sys

# Создать сокет UDP
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

server_address = ('localhost', 10000)
message = b'This is the message.  It will be repeated.'

try:

    # Отправить данные
    print('sending {!r}'.format(message))
    sent = sock.sendto(message, server_address)

    # Получить ответ
    print('waiting to receive')
    data, server = sock.recvfrom(4096)
    print('received {!r}'.format(data))

finally:
    print('closing socket')
    sock.close()
```

11.2.3.3. Совместная работа сервера и клиента

При запуске сервера выводится следующая информация.

```
$ python3 socket_echo_server_dgram.py
starting up on localhost port 10000

waiting to receive message
received 42 bytes from ('127.0.0.1', 57870)
b'This is the message. It will be repeated.'
sent 42 bytes back to ('127.0.0.1', 57870)

waiting to receive message
```

На стороне клиента вывод выглядит так.

```
$ python3 socket_echo_client_dgram.py

sending b'This is the message.  It will be repeated.'
waiting to receive
received b'This is the message.  It will be repeated.'
closing socket
```

11.2.4. Сокеты домена Unix

С точки зрения программиста сокет домена Unix (UDS) отличается от сокета TCP/IP в двух отношениях. Во-первых, адресом сокета является путь в файловой системе, а не имя сервера и порт. Во-вторых, узел, созданный в файловой системе для представления сокета, существует постоянно даже после его закрытия, поэтому его необходимо удалять каждый раз, когда запускается сервер. Приведенный ранее пример с эхо-сервером можно обновить для использования протокола UDS, внеся несколько изменений.

Сокет должен создаваться с адресом семейства `AF_UNIX`. Привязка сокета и управление входящими соединениями осуществляются точно так же, как в случае сокетов TCP/IP.

Листинг 11.27. `socket_echo_server_uds.py`

```
import socket
import sys
import os

server_address = './uds_socket'

# Убедиться в том, что сокет еще не существует
try:
    os.unlink(server_address)
except OSError:
    if os.path.exists(server_address):
        raise

# Создать сокет UDS
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

# Привязать сокет к адресу
print('starting up on {}'.format(server_address))
sock.bind(server_address)

# Слушать входящие соединения
sock.listen(1)

while True:
    # Ждать соединения
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('connection from', client_address)
```

```
        # Получать данные небольшими порциями и
        # отправлять их обратно
        while True:
            data = connection.recv(16)
            print('received {!r}'.format(data))
            if data:
                print('sending data back to the client')
                connection.sendall(data)
            else:
                print('no data from', client_address)
                break

    finally:
        # Закрыть соединение
        connection.close()
```

Также необходимо изменить параметры клиента, чтобы он мог работать с протоколом UDS. Клиент должен предполагать, что узел файловой системы для сокета существует, поскольку сервер создает его, когда привязывается к адресу. Отправка и получение данных клиентом UDS осуществляются точно так же, как и в описанном ранее случае клиента TCP/IP.

Листинг 11.28. `socket_echo_client_uds.py`

```
import socket
import sys

# Создать сокет UDS
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

# Подключить сокет к порту, который прослушивается сервером
server_address = './uds_socket'
print('connecting to {}'.format(server_address))
try:
    sock.connect(server_address)
except socket.error as msg:
    print(msg)
    sys.exit(1)

try:

    # Отправить данные
    message = b'This is the message.  It will be repeated.'
    print('sending {!r}'.format(message))
    sock.sendall(message)

    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data))
```

```
finally:
    print('closing socket')
    sock.close()
```

Не считая соответствующих обновлений, касающихся адреса, программа в основном выводит ту же информацию, что и прежде. Сервер отображает полученные сообщения, а также сообщения, отправляемые обратно клиенту.

```
$ python3 socket_echo_server_uds.py

starting up on ./uds_socket
waiting for a connection
connection from
received b'This is the mess'
sending data back to the client
received b'age. It will be'
sending data back to the client
received b' repeated.'
sending data back to the client
received b''
no data from
waiting for a connection
```

Клиент отправляет все сообщение целиком, но получает обратное сообщение по частям.

```
$ python3 socket_echo_client_uds.py
connecting to ./uds_socket
sending b'This is the message. It will be repeated.'
received b'This is the mess'
received b'age. It will be'
received b' repeated.'
closing socket
```

11.2.4.1. Права доступа

Поскольку сокет UDS представляется узлом в файловой системе, для управления доступом к серверу могут быть использованы стандартные права доступа к файлу.

```
$ ls -l ./uds_socket

srwxr-xr-x 1 dhellmann dhellmann 0 Aug 21 11:19 uds_socket

$ sudo chown root ./uds_socket

$ ls -l ./uds_socket

srwxr-xr-x 1 root dhellmann 0 Aug 21 11:19 uds_socket
```

Теперь выполнение клиента от имени пользователя, не обладающего административными привилегиями, приводит к ошибке, поскольку процесс не имеет разрешения на открытие сокета.

```
$ python3 socket_echo_client_uds.py

connecting to ./uds_socket
[Errno 13] Permission denied
```

11.2.4.2. Обмен данными между родительскими и дочерними процессами

Функция `socketpair()` позволяет настраивать сокеты UDS для межпроцессного взаимодействия при работе под управлением Unix. Она создает пару соединенных сокетов, которые могут быть использованы для взаимодействия между родительским и дочерним процессом.

Листинг 11.29. `socket_socketpair.py`

```
import socket
import os

parent, child = socket.socketpair()

pid = os.fork()

if pid:
    print('in parent, sending message')
    child.close()
    parent.sendall(b'ping')
    response = parent.recv(1024)
    print('response from child:', response)
    parent.close()

else:
    print('in child, waiting for message')
    parent.close()
    message = child.recv(1024)
    print('message from parent:', message)
    child.sendall(b'pong')
    child.close()
```

По умолчанию создается сокет UDS, однако вызывающий код может определить параметры создаваемых сокетов, указав семейство адресов, тип сокета и даже протокол.

```
$ python3 -u socket_socketpair.py

in parent, sending message
in child, waiting for message
message from parent: b'ping'
response from child: b'pong'
```

11.2.5. Многоадресатное вещание

Во многих случаях двухточечных соединений, или соединений "точка – точка", вполне хватает для того, чтобы обеспечить обмен данными, но передача

одной и той же информации многим равноправным узлам становится все более затруднительной по мере роста количества прямых соединений. Отправка сообщений каждой конечной точке требует дополнительных затрат процессорного времени и полосы пропускания, что может создавать проблемы при передаче потоковой видео- или аудиоинформации. Использование *многоадресатного вещания*, или *мультивещания* (multicasting), для доставки сообщений более чем одной конечной точке за один раз обеспечивает более высокую эффективность, поскольку сетевая инфраструктура гарантирует доставку пакетов всем получателям.

Для рассылки многоадресатных сообщений всегда используется протокол UDP, поскольку протокол TCP предполагает наличие двух взаимодействующих подсистем. Адреса, используемые для отправки многоадресатных сообщений и называемые *мультикастными группами*, представляют собой подмножество обычного адресного диапазона IPv4 (от 224.0.0.0 до 230.255.255.255), зарезервированное для многоадресатного трафика. Сетевые маршрутизаторы и коммутаторы обрабатывают эти адреса специальным образом, поэтому сообщения, посланные группе, могут распространяться по Интернету и доставляться всем получателям, присоединившимся к данной группе.

Примечание

В некоторых маршрутизаторах и коммутаторах многоадресатный трафик по умолчанию отключен. Если у вас возникнут проблемы с выполнением приведенных ниже примеров, проверьте конфигурацию своей сети.

11.2.5.1. Передача многоадресатных сообщений

В следующем примере модифицированная версия эхо-клиента посылает сообщение мультикастной группе, а затем выводит отчет обо всех полученных ответах. Поскольку количество ответов, которые будут получены, неизвестно заранее, используется тайм-аут, позволяющий избежать бесконечной блокировки сокета в ожидании ответа.

Сокет также должен быть сконфигурирован с определенным временем жизни пакетов данных в системе (time-to-live, TTL). Параметр TTL управляет количеством сетей, которые получат пакет, и устанавливается посредством передачи опции `IP_MULTICAST_TTL` методу `setsockopt()`. Используемое по умолчанию значение 1 означает, что пакеты не будут направляться маршрутизатором за пределы текущего сегмента сети. Значение параметра TTL не может превышать 255 и должно упаковываться в один байт.

Листинг 11.30. `socket_multicast_sender.py`

```
import socket
import struct
import sys

message = b'very important data'
multicast_group = ('224.3.29.71', 10000)

# Создать сокет для датаграмм
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
# Задать тайм-аут, чтобы избежать бесконечной блокировки сокета
# при попытках получения данных
sock.settimeout(0.2)

# Установить для сообщений значение ttl, равное 1, чтобы
# они не выходили за пределы сегмента локальной сети
ttl = struct.pack('b', 1)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)

try:

    # Послать данные мультикастной группе
    print('sending {!r}'.format(message))
    sent = sock.sendto(message, multicast_group)

    # Ожидать ответы от всех получателей
    while True:
        print('waiting to receive')
        try:
            data, server = sock.recvfrom(16)
        except socket.timeout:
            print('timed out, no more responses')
            break
        else:
            print('received {!r} from {}'.format(
                data, server))

finally:
    print('closing socket')
    sock.close()
```

Остальная часть кода отправителя сообщений напоминает код эхо-клиента UDP, за исключением того, что она ожидает получения множества ответов. Метод `recvfrom()` вызывается в цикле до тех пор, пока не истечет время тайм-аута.

11.2.5.2. Получение многоадресатных сообщений

Первое, что необходимо сделать для создания получателя сообщений, включаемого в многоадресатную группу, — это создать UDP-сокет. Создав обычный сокет и привязав его к порту, можно добавить этот сокет в мультикастную группу с помощью метода `setsockopt()` для изменения опции `IP_ADD_MEMBERSHIP`. Значением этой опции является упакованное 8-байтовое представление адреса мультикастной группы, за которым следует сетевой интерфейс, используемый сервером для прослушивания трафика и идентифицируемый по IP-адресу. В данном случае получатель прослушивает все интерфейсы, используя опцию `INADDR_ANY`.

Листинг 11.31. `socket_multicast_receiver.py`

```
import socket
import struct
import sys

multicast_group = '224.3.29.71'
```

```
server_address = ('', 10000)

# Создать сокет
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Привязать сокет к адресу сервера
sock.bind(server_address)

# Приказать операционной системе добавить сокет в
# мультикастную группу на всех интерфейсах
group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(
    socket.IPPROTO_IP,
    socket.IP_ADD_MEMBERSHIP,
    mreq)

# Цикл получения сообщений и отправки ответов
while True:
    print('\nwaiting to receive message')
    data, address = sock.recvfrom(1024)

    print('received {} bytes from {}'.format(
        len(data), address))
    print(data)

    print('sending acknowledgement to', address)
    sock.sendto(b'ack', address)
```

Основной цикл для получателя выглядит точно так же, как и для обычного сервера UDP.

11.2.5.3. Выходная информация примера

Ниже приведены результаты, выводимые получателями многоадресатных сообщений на двух хостах: A с адресом 192.168.1.13 и B с адресом 192.168.1.14.

```
[A]$ python3 socket_multicast_receiver.py

waiting to receive message
received 19 bytes from ('192.168.1.14', 62650)
b'very important data'
sending acknowledgement to ('192.168.1.14', 62650)

waiting to receive message

[B]$ python3 source/socket/socket_multicast_receiver.py

waiting to receive message
received 19 bytes from ('192.168.1.14', 64288)
b'very important data'
sending acknowledgement to ('192.168.1.14', 64288)

waiting to receive message
```

Отправитель сообщений выполняется на хосте B.

```
[B]$ python3 socket_multicast_sender.py
sending b'very important data'
waiting to receive
received b'ack' from ('192.168.1.14', 10000)
waiting to receive
received b'ack' from ('192.168.1.13', 10000)
waiting to receive
timed out, no more responses
closing socket
```

Сообщение было отправлено один раз, и его получение было подтверждено двумя хостами, A и B.

Дополнительные ссылки

- Википедия: *Мультивещание*[8]. Статья, содержащая технические детали многоадресатного вещания.
- Википедия: *IP multicast*[9]. Статья о многоадресатном вещании в Интернете, содержащая информацию о механизмах адресации.

11.2.6. Отправка двоичных данных

Сокеты передают потоки байтов. Эти байты могут содержать текстовые сообщения, закодированные в виде байтов, как в предыдущих примерах, или двоичные данные, упакованные в буфер с помощью модуля `struct` (раздел 2.7) для подготовки к передаче.

Представленная ниже клиентская программа упаковывает целое число, строку из двух символов и число с плавающей точкой в последовательность байтов, принимаемую сокетом для последующей передачи.

Листинг 11.32. `socket_binary_client.py`

```
import binascii
import socket
import struct
import sys

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('localhost', 10000)
sock.connect(server_address)

values = (1, b'ab', 2.7)
packer = struct.Struct('I 2s f')
packed_data = packer.pack(*values)

print('values =', values)
```

[8] https://ru.wikipedia.org/wiki/Мультивещание

[9] https://en.wikipedia.org/wiki/IP_multicast

```
try:
    # Отправить данные
    print('sending {!r}'.format(binascii.hexlify(packed_data)))
    sock.sendall(packed_data)
finally:
    print('closing socket')
    sock.close()
```

При обмене мультибайтовыми данными между двумя системами очень важно убедиться в том, что обеим сторонам соединения известен порядок следования байтов в них, а также корректный порядок их обратной сборки, соответствующий локальной архитектуре. Серверная программа использует тот же спецификатор `Struct` для распаковки полученных байтов, чтобы они могли быть интерпретированы в корректном порядке.

Листинг 11.33. `socket_binary_server.py`

```
import binascii
import socket
import struct
import sys

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('localhost', 10000)
sock.bind(server_address)
sock.listen(1)

unpacker = struct.Struct('I 2s f')

while True:
    print('\nwaiting for a connection')
    connection, client_address = sock.accept()
    try:
        data = connection.recv(unpacker.size)
        print('received {!r}'.format(binascii.hexlify(data)))

        unpacked_data = unpacker.unpack(data)
        print('unpacked:', unpacked_data)

    finally:
        connection.close()
```

Выполнение этого клиента приводит к следующим результатам.

```
$ python3 source/socket/socket_binary_client.py
values = (1, b'ab', 2.7)
sending b'0100000061620000cdcc2c40'
closing socket
```

Сервер отображает полученные данные.

```
$ python3 socket_binary_server.py

waiting for a connection
received b'0100000061620000cdcc2c40'
unpacked: (1, b'ab', 2.700000047683716)

waiting for a connection
```

Значение с плавающей точкой несколько теряет в точности в процессе упаковки и распаковки, однако во всем остальном данные передаются так, как и ожидалось. Важный момент, о котором не следует забывать, заключается в том, что в зависимости от того, какова величина целого числа, его передача в текстовом виде может оказаться более эффективной, чем использование упакованной структуры. Например, для целого числа 1 требуется 1 байт, если оно представляется в виде строки, и 4 байта, если оно упаковывается в структуру.

Дополнительные ссылки

- `struct` (раздел 11.7). Взаимные преобразования между строками и другими типами данных.

11.2.7. Неблокирующее взаимодействие и тайм-ауты

По умолчанию сокет конфигурируется для выполнения блокирующего ввода-вывода, т.е. выполнение программы приостанавливается до тех пор, пока сокет не закончит свою работу. Вызовы метода `send()` ожидают доступности исходящих данных, а вызовы метода `recv()` ожидают, пока другая программа не отправит данные, которые можно будет прочитать. Такую форму операций ввода-вывода несложно понять, но она может приводить к неэффективному выполнению операций и даже взаимоблокировкам, когда две программы блокируют друг друга, ожидая отправки или получения данных одна от другой.

Из этой ситуации возможно несколько выходов. Один из них заключается в том, чтобы использовать отдельный поток для каждого сокета. Однако это может породить другие проблемы, связанные с взаимодействием потоков. При другом подходе блокирование вообще исключается посредством перевода сокета в неблокирующий режим, когда он немедленно возвращает управление, если не находит данных для выполнения операции. Чтобы перевести сокет в неблокирующий режим, следует изменить состояние его флага блокировки с помощью метода `setblocking()`. По умолчанию этот флаг имеет значение `1`, что соответствует блокирующему режиму; значение `0` отключает блокирование. Если блокирование сокета отключено и он не готов к выполнению операции, то возбуждается исключение `socket.error`.

Компромиссное решение заключается в том, чтобы задать тайм-аут (предельное время ожидания) для сокета. Метод `settimeout()` позволяет установить значение тайм-аута в виде числа с плавающей точкой, представляющего длительность интервала ожидания в секундах, в течение которого сокет будет блокироваться до принятия решения о том, что он не готов к выполнению операции. По истечении тайм-аута возбуждается исключение `socket.timeout`.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `socket`[10].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `socket` (раздел A.6.39).
- `select` (раздел 11.4). Тестирование готовности сокета к чтению или записи в режиме неблокирующего ввода-вывода.
- `SocketServer`. Фреймворк для создания сетевых серверов.
- `asyncio` (раздел 10.5). Асинхронный ввод-вывод и инструменты параллельных вычислений.
- `urllib` и `urllib2`. Большинство сетевых клиентов должно использовать более удобные библиотеки для доступа к удаленным ресурсам посредством URL-адресов.
- *Socket Programming HOWTO* (Gordon McMillan)[11]. Учебное руководство, включенное в документацию стандартной библиотеки.
- *Foundations of Python Network Programming, Third Edition*, by Brandon Rhodes and John Goerzen. Apress, 2014. ISBN-10: 1430258543.
- *Unix Network Programming, Volume 1: The Sockets Networking API, Third Edition*, by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Addison-Wesley, 2004. ISBN-10: 0131411551.

11.3. `selectors`: абстракции мультиплексирования ввода-вывода

Модуль `selectors` предоставляет платформонезависимый абстрактный слой, располагающийся поверх специфических для каждой платформы функций мониторинга ввода-вывода, содержащихся в модуле `select` (раздел 11.4).

11.3.1. Рабочая модель

Предлагаемые модулем `selectors` программные интерфейсы основаны на событиях, как и примитивы модуля `select`. Они существуют в нескольких реализациях, и модуль автоматически настраивает экземпляр селектора с псевдонимом `DefaultSelector`, ссылающимся на ту реализацию, которая наиболее эффективна для текущей конфигурации системы.

Объект селектора имеет методы, позволяющие указать, какие события, связанные с сокетом, необходимо отслеживать, и предоставляет вызывающему объекту возможность ожидать наступления событий платформонезависимым способом. В результате регистрации объекта файла, события которого будут отслеживаться, создается экземпляр именованного кортежа `SelectorKey`, связывающий объект файла с его дескриптором файла, выбранной маской событий и необязательными данными приложения. Владелец объекта селектора вызывает метод `select()` для получения информации о событиях. Возвращаемым значением является список кортежей, каждый из которых содержит экземпляр `SelectorKey`, соответствующий доступному объекту файла, и битовую маску, которая указывает на

[10] `https://docs.python.org/3.5/library/socket.html`

[11] `https://docs.python.org/3/howto/sockets.html`

ожидающие обработки события. Программа, использующая данный селектор, должна повторно вызывать метод `select()` и соответствующим образом обрабатывать события.

11.3.2. Эхо-сервер

В приведенном ниже примере эхо-сервера данные приложения, содержащиеся в экземпляре `SelectorKey`, используются для регистрации функции обратного вызова, которая должна вызываться для обработки нового события. Основной цикл получает функцию обратного вызова из экземпляра `SelectorKey` и передает ему объект сокета и маску событий. При запуске сервера он регистрирует функцию `accept()`, которая будет вызываться для событий чтения на основном сокете сервера. В результате принятия соединения создается новый сокет, который затем регистрируется с функцией `read()` в качестве функции обратного вызова для событий чтения.

Листинг 11.34. `selectors_echo_server.py`

```
import selectors
import socket

mysel = selectors.DefaultSelector()
keep_running = True


def read(connection, mask):
    "Функция обратного вызова для событий чтения"
    global keep_running

    client_address = connection.getpeername()
    print('read({})'.format(client_address))
    data = connection.recv(1024)
    if data:
        # Сокет клиента, предназначенный для чтения,
        # содержит данные
        print('  received {!r}'.format(data))
        connection.sendall(data)
    else:
        # Интерпретировать пустой результат как закрытое
        # соединение
        print('  closing')
        mysel.unregister(connection)
        connection.close()
        # Приказать основному циклу приостановить выполнение
        keep_running = False


def accept(sock, mask):
    "Функция обратного вызова для новых подключений"
    new_connection, addr = sock.accept()
    print('accept({})'.format(addr))
    new_connection.setblocking(False)
    mysel.register(new_connection, selectors.EVENT_READ, read)
```

```
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address))
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(False)
server.bind(server_address)
server.listen(5)

mysel.register(server, selectors.EVENT_READ, accept)

while keep_running:
    print('waiting for I/O')
    for key, mask in mysel.select(timeout=1):
        callback = key.data
        callback(key.fileobj, mask)

print('shutting down')
mysel.close()
```

Если метод `read()` не находит данных для чтения из сокета, то он интерпретирует событие чтения как сигнал о том, что другая сторона закрывает соединение. В связи с этим он удаляет сокет из селектора и закрывает его. Поскольку это всего лишь пример, работа данного сервера прекращается сразу же после завершения обмена данными с единственным клиентом.

11.3.3. Эхо-клиент

В приведенном ниже примере эхо-клиента все события ввода-вывода обрабатываются в основном цикле, а не с помощью функций обратного вызова. Программа настраивает селектор для отслеживания событий чтения сокетом, а также событий готовности сокета к отправке данных. Поскольку отслеживаются два типа событий, клиент должен проверять, какое именно из них произошло, используя для этого значение маски. После отправки всех исходящих данных конфигурация селектора изменяется таким образом, чтобы отслеживалось лишь наличие данных для чтения.

Листинг 11.35. `selectors_echo_client.py`

```
import selectors
import socket

mysel = selectors.DefaultSelector()
keep_running = True
outgoing = [
    b'It will be repeated.',
    b'This is the message.  ',
]
bytes_sent = 0
bytes_received = 0

# Установление соединения - блокирующая операция, поэтому
# после ее выполнения следует вызвать метод setblocking()
server_address = ('localhost', 10000)
print('connecting to {} port {}'.format(*server_address))
```

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(server_address)
sock.setblocking(False)

# Настроить селектор для отслеживания готовности сокета к отправке
# данных, а также наличия данных для чтения
mysel.register(
    sock,
    selectors.EVENT_READ | selectors.EVENT_WRITE,
)

while keep_running:
    print('waiting for I/O')
    for key, mask in mysel.select(timeout=1):
        connection = key.fileobj
        client_address = connection.getpeername()
        print('client({})'.format(client_address))

        if mask & selectors.EVENT_READ:
            print('  ready to read')
            data = connection.recv(1024)
            if data:
                # Сокет клиента, предназначенный для чтения,
                # получил данные
                print('  received {!r}'.format(data))
                bytes_received += len(data)

            # Прекратить выполнение при получении пустого
            # результата, указывающего на закрытие соединения,
            # а также после получения копии всех отправленных
            # данных
            keep_running = not (
                data or
                (bytes_received and
                 (bytes_received == bytes_sent))
            )

        if mask & selectors.EVENT_WRITE:
            print('  ready to write')
            if not outgoing:
                # Сообщения отсутствуют, поэтому записывать больше
                # нечего. Изменить регистрацию, оставив только
                # чтение ответов от сервера.
                print('  switching to read-only')
                mysel.modify(sock, selectors.EVENT_READ)
            else:
                # Отправить следующее сообщение
                next_msg = outgoing.pop()
                print('  sending {!r}'.format(next_msg))
                sock.sendall(next_msg)
                bytes_sent += len(next_msg)

print('shutting down')
```

```
mysel.unregister(connection)
connection.close()
mysel.close()
```

Клиент отслеживает объем как отправленных, так и полученных им данных. Если эти значения совпадают и не равны нулю, то клиент выходит из цикла обработки и корректно завершает работу, удаляя сокет из селектора и закрывая и сокет, и селектор.

11.3.4. Совместная работа сервера и клиента

Чтобы клиент и сервер могли взаимодействовать, они должны выполняться в разных окнах терминала. В окне сервера отображаются входящее соединение и данные, а также ответ, отправляемый обратно клиенту.

```
$ python3 source/selectors/selectors_echo_server.py
starting up on localhost port 10000
waiting for I/O
waiting for I/O
accept(('127.0.0.1', 59850))
waiting for I/O
read(('127.0.0.1', 59850))
received b'This is the message. It will be repeated.'
waiting for I/O
read(('127.0.0.1', 59850))
closing
shutting down
```

В окне клиента отображаются исходящее сообщение и ответ, полученный от сервера.

```
$ python3 source/selectors/selectors_echo_client.py
connecting to localhost port 10000
waiting for I/O
client(('127.0.0.1', 10000))
ready to write
sending b'This is the message. '
waiting for I/O
client(('127.0.0.1', 10000))
ready to write
sending b'It will be repeated.'
waiting for I/O
client(('127.0.0.1', 10000))
ready to write
switching to read-only
waiting for I/O
client(('127.0.0.1', 10000))
ready to read
received b'This is the message. It will be repeated.'
shutting down
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `selectors`[12].
- `select` (раздел 11.4). Низкоуровневые программные интерфейсы для организации эффективного ввода-вывода.

11.4. `select`: эффективное ожидание завершения ввода-вывода

Модуль `select` представляет доступ к платформозависимым функциям отслеживания операций ввода-вывода. Наилучшую портируемость обеспечивает интерфейс, предлагаемый функцией `select()` стандарта POSIX, которая доступна на платформах Unix и Windows. Данный модуль включает также функцию `poll()` (доступна только на платформе UNIX) и несколько опций, которые работают только со специфическими вариантами Unix.

Примечание

Новый модуль `selectors` (раздел 11.3) предоставляет высокоуровневый интерфейс, построенный поверх программных интерфейсов модуля `select`. Модуль `selectors` упрощает написание переносимого кода, поэтому всегда отдавайте предпочтение ему, за исключением тех случаев, когда без использования низкоуровневых интерфейсов, предлагаемых модулем `select`, невозможно обойтись.

11.4.1. Использование функции `select()`

Функция `select()` в Python — непосредственный интерфейс к системному вызову `select()` в Unix. Она может работать с сокетами, открытыми файлами и каналами, т.е. с любыми объектами, обладающими методом `fileno()`, который возвращает действительный дескриптор файла. Функция `select()` упрощает одновременный мониторинг нескольких соединений и более эффективна, чем цикл опроса Python с использованием тайм-аутов, поскольку мониторинг осуществляется на сетевом уровне операционной системы, а не в интерпретаторе.

Примечание

Использование файловых объектов Python с функцией `select()` возможно в Unix, но не поддерживается в Windows.

Пример эхо-сервера, приведенный при описании модуля `socket` (раздел 11.2), можно расширить для одновременного отслеживания нескольких соединений с помощью функции `select()`. Новая версия кода начинается с создания неблокирующего сокета TCP/IP и его конфигурирования для прослушивания адреса.

Листинг 11.36. `select_echo_server.py`

```
import select
import socket
import sys
import queue
```

[12] https://docs.python.org/3.5/library/selectors.html

```
# Создать сокет TCP/IP
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

# Привязать сокет к порту
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address),
      file=sys.stderr)
server.bind(server_address)

# Слушать входящие соединения
server.listen(5)
```

Аргументами функции `select()` являются три списка, которые содержат отслеживаемые коммуникационные каналы. Первый из них перечисляет объекты, проверяемые на готовность к получению входящих данных, второй — объекты, проверяемые на готовность получать исходящие данные, когда будет достаточно места в их буфере, а третий — объекты, проверяемые на наличие исключительных ситуаций (обычно используется комбинация объектов входных и выходных каналов). Следующим шагом является конфигурирование списков, содержащих источники входных данных и пункты назначения выходных данных, для передачи функции `select()`.

```
# Сокеты, из которых ожидается чтение данных
inputs = [server]

# Сокеты, в которые предполагается записывать данные
outputs = []
```

Соединения добавляются в списки и удаляются из них в основном цикле сервера. Поскольку данная версия сервера вместо немедленной отправки ответа будет дожидаться, пока сокет не станет доступным для записи, прежде чем отправлять данные, каждое выходное соединение нуждается в очереди, которая будет служить буфером для отправляемых посредством нее данных.

```
# Очереди исходящих сообщений (socket:Queue)
message_queues = {}
```

Основная часть программы выполняется в цикле, вызывая метод `select()` для блокирования и ожидания активности сети.

```
while inputs:

    # Ждать, пока по крайней мере один из сокетов
    # не будет готов к обработке
    print('waiting for the next event', file=sys.stderr)
    readable, writable, exceptional = select.select(inputs,
                                                    outputs,
                                                    inputs)
```

Функция `select()` возвращает три новых списка, содержащих подмножества содержимого переданных ей списков. В список читаемых сокетов входят сокеты,

которые имеют буферизованные данные, доступные для чтения. В список записываемых сокетов входят сокеты, имеющие свободное буферное пространство, в которое могут быть записаны данные. А в список сокетов, в которых возникли исключительные ситуации, входят сокеты, в которых возникли ошибки (точный смысл определения "исключительная ситуация" зависит от платформы).

Читаемые сокеты представляют три возможных случая. Если сокет является основным серверным сокетом (т.е. тем, который используется для прослушивания соединений), то условие "читаемости" означает, что он готов принять еще одно входящее соединение. Кроме добавления нового соединения в список отслеживаемых входов этот раздел кода устанавливает для клиентского сокета неблокирующий режим.

```
# Обработать входные данные
for s in readable:

    if s is server:
        # Читаемый сокет готов к принятию соединения
        connection, client_address = s.accept()
        print('  connection from', client_address,
              file=sys.stderr)
        connection.setblocking(0)
        inputs.append(connection)

        # Предоставить соединению очередь для
        # буферизации отправляемых данных
        message_queues[connection] = queue.Queue()
```

Следующий случай представляет соединение с клиентом, который отправил данные. Данные читаются с помощью метода `recv()`, а затем помещаются в очередь для отправки через сокет обратно клиенту.

```
    else:
        data = s.recv(1024)
        if data:
            # Читаемый клиентский сокет имеет данные для
            # чтения
            print('  received {!r} from {}'.format(
                data, s.getpeername()), file=sys.stderr,
            )
            message_queues[s].put(data)
            # Добавить выходной канал для отправки ответа
            if s not in outputs:
                outputs.append(s)
```

Читаемый сокет, который не возвращает данные с помощью метода `recv()`, соответствует клиенту, который разорвал соединение, и теперь поток готов к закрытию.

```
        else:
            # Интерпретировать пустой результат как закрытие
            # соединения
            print('  closing', client_address,
```

```
                file=sys.stderr)
            # Прекратить прослушивание входного канала для
            # данного соединения
            if s in outputs:
                outputs.remove(s)
            inputs.remove(s)
            s.close()

            # Удалить очередь сообщений
            del message_queues[s]
```

Для записываемых сокетов количество различных возможных случаев меньше. Если очередь содержит данные, предназначенные для отправки через соединение, то отправляется следующее сообщение. В противном случае соединение удаляется из списка выходных соединений, так что при прохождении цикла в следующий раз функция `select()` не укажет, что данный сокет готов к отправке данных.

```
# Обработать выходные данные
for s in writable:
    try:
        next_msg = message_queues[s].get_nowait()
    except queue.Empty:
        # Ввиду отсутствия сообщений, ожидающих обработки,
        # прекратить проверку возможности выполнения записи
        print('  ', s.getpeername(), 'queue empty',
              file=sys.stderr)
        outputs.remove(s)
    else:
        print('  sending {!r} to {}'.format(next_msg,
                                            s.getpeername()),
              file=sys.stderr)
        s.send(next_msg)
```

Наконец, сокеты, входящие в исключительный список, закрываются.

```
# Обработать "исключительные условия"
for s in exceptional:
    print('exception condition on', s.getpeername(),
          file=sys.stderr)
    # Прекратить прослушивание входного канала для
    # данного соединения
    inputs.remove(s)
    if s in outputs:
        outputs.remove(s)
    s.close()

    # Удалить очередь сообщений
    del message_queues[s]
```

В данном примере клиентская программа использует два сокета для демонстрации того, каким образом сервер управляет одновременно несколькими сое-

динениями с помощью функции `select()`. Клиентский код начинается с подключения каждого сокета TCP/IP к серверу.

Листинг 11.37. `select_echo_multiclient.py`

```
import socket
import sys

messages = [
    'This is the message. ',
    'It will be sent ',
    'in parts.',
]
server_address = ('localhost', 10000)

# Создать сокет TCP/IP
socks = [
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
]

# Подключить сокет к порту, который прослушивается сервером
print('connecting to {} port {}'.format(*server_address),
      file=sys.stderr)
for s in socks:
    s.connect(server_address)
```

Далее клиент отправляет по одной порции сообщения за раз через каждый сокет и читает доступные ответы после записи новых данных.

```
for message in messages:
    outgoing_data = message.encode()

    # Послать сообщения в оба сокета
    for s in socks:
        print('{}: sending {!r}'.format(s.getsockname(),
                                        outgoing_data),
              file=sys.stderr)
        s.send(outgoing_data)

    # Прочитать сообщения в обоих сокетах
    for s in socks:
        data = s.recv(1024)
        print('{}: received {!r}'.format(s.getsockname(),
                                         data),
              file=sys.stderr)
        if not data:
            print('closing socket', s.getsockname(),
                  file=sys.stderr)
            s.close()
```

Запустите программы сервера и клиента в разных окнах. Вывод будет выглядеть примерно так, как показано ниже, хотя номера портов могут отличаться.

```
$ python3 select_echo_server.py
starting up on localhost port 10000
waiting for the next event
  connection from ('127.0.0.1', 61003)
waiting for the next event
  connection from ('127.0.0.1', 61004)
waiting for the next event
  received b'This is the message. ' from ('127.0.0.1', 61003)
  received b'This is the message. ' from ('127.0.0.1', 61004)
waiting for the next event
  sending b'This is the message. ' to ('127.0.0.1', 61003)
  sending b'This is the message. ' to ('127.0.0.1', 61004)
waiting for the next event
   ('127.0.0.1', 61003) queue empty
   ('127.0.0.1', 61004) queue empty
waiting for the next event
  received b'It will be sent ' from ('127.0.0.1', 61003)
  received b'It will be sent ' from ('127.0.0.1', 61004)
waiting for the next event
  sending b'It will be sent ' to ('127.0.0.1', 61003)
  sending b'It will be sent ' to ('127.0.0.1', 61004)
waiting for the next event
   ('127.0.0.1', 61003) queue empty
   ('127.0.0.1', 61004) queue empty
waiting for the next event
  received b'in parts.' from ('127.0.0.1', 61003)
waiting for the next event
  received b'in parts.' from ('127.0.0.1', 61004)
  sending b'in parts.' to ('127.0.0.1', 61003)
waiting for the next event
   ('127.0.0.1', 61003) queue empty
sending b'in parts.' to ('127.0.0.1', 61004)
waiting for the next event
   ('127.0.0.1', 61004) queue empty
waiting for the next event
  closing ('127.0.0.1', 61004)
  closing ('127.0.0.1', 61004)
waiting for the next event
```

В окне клиента отображаются данные, отправляемые и получаемые обоими сокетами.

```
$ python3 select_echo_multiclient.py
connecting to localhost port 10000
('127.0.0.1', 61003): sending b'This is the message. '
('127.0.0.1', 61004): sending b'This is the message. '
('127.0.0.1', 61003): received b'This is the message. '
('127.0.0.1', 61004): received b'This is the message. '
('127.0.0.1', 61003): sending b'It will be sent '
('127.0.0.1', 61004): sending b'It will be sent '
('127.0.0.1', 61003): received b'It will be sent '
('127.0.0.1', 61004): received b'It will be sent '
('127.0.0.1', 61003): sending b'in parts.'
```

```
('127.0.0.1', 61004): sending b'in parts.'
('127.0.0.1', 61003): received b'in parts.'
('127.0.0.1', 61004): received b'in parts.'
```

11.4.2. Неблокирующий ввод-вывод с тайм-аутами

Функция `select()` также имеет необязательный четвертый параметр — `timeout` (тайм-аут), т.е. предельное время ожидания в секундах, в течение которого необходимо проверять, не появился ли доступный активный канал. Использование этого параметра позволяет основной программе вызывать функцию `select()` в качестве части более широкого цикла обработки, предпринимая другие действия в промежутках между проверками активности сети.

По истечении предельного интервала ожидания функция `select()` возвращает три пустых списка. Чтобы обновить пример серверной программы для использования тайм-аута, необходимо добавить дополнительный аргумент в вызов функции `select()` и обработать возвращаемые ею пустые списки.

Листинг 11.38. `select_echo_server_timeout.py`

```
    readable, writable, exceptional = select.select(inputs,
                                                    outputs,
                                                    inputs,
                                                    timeout)

    if not (readable or writable or exceptional):
        print('  timed out, do some other work here',
              file=sys.stderr)
        continue
```

Приведенная ниже "медленная" версия клиентской программы приостанавливает выполнение после отправки каждого сообщения с целью имитации задержки при передаче сообщений.

Листинг 11.39. `select_echo_slow_client.py`

```
import socket
import sys
import time

# Создать сокет TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Подключить сокет к порту, прослушиваемому сервером
server_address = ('localhost', 10000)
print('connecting to {} port {}'.format(*server_address),
      file=sys.stderr)
sock.connect(server_address)

time.sleep(1)

messages = [
    'Part one of the message.',
    'Part two of the message.',
```

```
]
amount_expected = len(''.join(messages))

try:

    # Отправить данные
    for message in messages:
        data = message.encode()
        print('sending {!r}'.format(data), file=sys.stderr)
        sock.sendall(data)
        time.sleep(1.5)

    # Ожидание ответа
    amount_received = 0

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received {!r}'.format(data), file=sys.stderr)

finally:
    print('closing socket', file=sys.stderr)
    sock.close()
```

Выполнение новой версии сервера с медленным клиентом приводит к следующим результатам.

```
$ python3 select_echo_server_timeout.py
starting up on localhost port 10000
waiting for the next event
  timed out, do some other work here
waiting for the next event
  connection from ('127.0.0.1', 61144)
waiting for the next event
  timed out, do some other work here
waiting for the next event
  received b'Part one of the message.' from ('127.0.0.1', 61144)
waiting for the next event
   b'Part one of the message.' to ('127.0.0.1', 61144)
waiting for the next event
('127.0.0.1', 61144) queue empty
waiting for the next event
  timed out, do some other work here
waiting for the next event
  received b'Part two of the message.' from ('127.0.0.1', 61144)
waiting for the next event
  sending b'Part two of the message.' to ('127.0.0.1', 61144)
waiting for the next event
('127.0.0.1', 61144) queue empty
waiting for the next event
  timed out, do some other work here
waiting for the next event
closing ('127.0.0.1', 61144)
```

```
waiting for the next event
  timed out, do some other work here
```

Ниже приведен вывод на стороне клиента.

```
$ python3 select_echo_slow_client.py
connecting to localhost port 10000
sending b'Part one of the message.'
sending b'Part two of the message.'
received b'Part one of the '
received b'message.Part two'
received b' of the message.'
closing socket
```

11.4.3. Использование функции `poll()`

Функция `poll()` предоставляет возможности, аналогичные тем, которые предлагает функция `select()`, но с более эффективной базовой реализацией. Недостатком функции `poll()` является то, что она не поддерживается на платформах Windows, и поэтому переносимость использующих ее программ ограничена.

Как и в предыдущих примерах, код эхо-сервера на основе функции `poll()` начинается с конфигурирования сокета.

Листинг 11.40. `select_poll_echo_server.py`

```
import select
import socket
import sys
import queue

# Создать сокет TCP/IP
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

# Привязать сокет к порту
server_address = ('localhost', 10000)
print('starting up on {} port {}'.format(*server_address),
      file=sys.stderr)
server.bind(server_address)

# Слушать входящие соединения
server.listen(5)

# Очереди исходящих сообщений
message_queues = {}
```

Длительность тайм-аута, передаваемая в качестве аргумента функции `poll()`, задается не в секундах, а в миллисекундах. Таким образом, паузе длительностью одна секунда соответствует значение 1000.

```
# Предотвратить бесконечное блокирование(миллисекунды)
TIMEOUT = 1000
```

Python реализует функцию `poll()` с классом, который управляет наблюдаемыми зарегистрированными каналами данных. Каналы добавляются посредством вызова функции `register()` с флагами, указывающими, какие события представляют интерес для этого канала. Полный набор флагов приведен в табл. 11.1.

Таблица 11.1. Флаги событий для функции `poll()`

Событиет	Описание
`POLLIN`	Имеются данные, доступные для чтения
`POLLPRI`	Имеются приоритетные данные, доступные для чтения
`POLLOUT`	Готовность к получению исходящих данных
`POLLERR`	Ошибка
`POLLHUP`	Канал закрыт
`POLLNVAL`	Канал не открыт

Эхо-сервер будет настраивать одни сокеты только для чтения, другие — только для записи. Подходящая комбинация флагов сохраняется в локальных переменных `READ_ONLY` и `READ_WRITE` соответственно.

```
# Обычно используемые установки флагов
READ_ONLY = (
    select.POLLIN |
    select.POLLPRI |
    select.POLLHUP |
    select.POLLERR
)
READ_WRITE = READ_ONLY | select.POLLOUT
```

Сокет сервера регистрируется таким образом, чтобы любые входящие соединения или данные запускали событие.

```
# Зарегистрировать объект, выполняющий опрос
poller = select.poll()
poller.register(server, READ_ONLY)
```

Поскольку функция `poll()` возвращает список кортежей, каждый из которых содержит дескриптор файла для сокета и флаг события, необходимо сопоставить числовые дескрипторы с объектами, чтобы извлечь сокет для чтения или записи.

```
# Сопоставить дескрипторы файлов с объектами сокетов
fd_to_socket = {
    server.fileno(): server,
}
```

Цикл сервера вызывает функцию `poll()`, а затем обрабатывает возвращенные события, выполняя поиск сокета и предпринимая необходимые действия в соответствии с флагом события.

```
while True:

    # Ждать, пока по крайней мере один из сокетов не перейдет
```

```
# в состояние готовности к выполнению обработки
print('waiting for the next event', file=sys.stderr)
events = poller.poll(TIMEOUT)

for fd, flag in events:

    # Получить фактический сокет по его дескриптору файла
    s = fd_to_socket[fd]
```

Как и в случае функции `select()`, если основной сокет сервера — читаемый, то в действительности это означает, что имеется входящий запрос на подключение от клиента. Новое соединение регистрируется с флагом `READ_ONLY` для отслеживания поступающих через него новых данных.

```
    # Обработать входные данные
    if flag & (select.POLLIN | select.POLLPRI):

        if s is server:
            # Читаемый сокет готов принять соединение
            connection, client_address = s.accept()
            print('  connection', client_address,
                  file=sys.stderr)
            connection.setblocking(0)
            fd_to_socket[connection.fileno()] = connection
            poller.register(connection, READ_ONLY)

            # Предоставить соединению очередь для буферизации
            # отправляемых данных
            message_queues[connection] = queue.Queue()
```

Сокеты, отличные от серверного, — это существующие клиенты, и для доступа к данным, ожидающим чтения, используется метод `recv()`.

```
        else:
            data = s.recv(1024)
```

Если метод `recv()` возвращает данные, то они помещаются в очередь исходящих сообщений для сокета. Затем флаги для этого сокета изменяются с помощью метода `modify()` таким образом, чтобы функция `poll()` отслеживала готовность сокета к получению данных.

```
            if data:
                # Читаемый клиентский сокет имеет данные для чтения
                print('  received {!r} from {}'.format(
                    data, s.getpeername()), file=sys.stderr,
                )
                message_queues[s].put(data)
                # Добавить выходной канал для отправки ответа
                poller.modify(s, READ_WRITE)
```

Возвращение методом `recv()` пустой строки означает разрыв соединения с клиентом, и в этом случае объект, выполняющий опрос, информируется о том,

что данный сокет следует игнорировать, посредством отмены регистрации сокета с помощью метода `unregister()`.

```
            else:
                # Интерпретировать пустой результат как
                # закрытие соединения
                print('  closing', client_address,
                      file=sys.stderr)
                # Прекратить прослушивание входных данных
                # для этого соединения
                poller.unregister(s)
                s.close()

                # Удалить очередь сообщений
                del message_queues[s]
```

Флагом `POLLHUP` отмечается клиент, который "отключился", не закрыв соединение корректным образом. Сервер прекращает опрос таких "исчезнувших" клиентов.

```
        elif flag & select.POLLHUP:
            # Клиент отключился
            print('  closing', client_address, '(HUP)',
                  file=sys.stderr)
            # Прекратить прослушивание входных данных
            # для этого соединения
            poller.unregister(s)
            s.close()
```

Для записываемых сокетов код обработки аналогичен коду, который использовался в примере с функцией `select()`, за исключением того, что вместо удаления сокета из списка выходных сокетов изменяются его флаги событий в объекте, выполняющем опрос, что делается с помощью метода `modify()`.

```
        elif flag & select.POLLOUT:
            # Сокет готов к отправке данных,
            # если таковые имеются
            try:
                next_msg = message_queues[s].get_nowait()
            except queue.Empty:
                # Ввиду отсутствия сообщений, ожидающих
                # обработки, прекратить проверку готовности
                # сокета к выполнению записи
                print(s.getpeername(), 'queue empty',
                      file=sys.stderr)
                poller.modify(s, READ_ONLY)
            else:
                print('  sending {!r} to {}'.format(
                    next_msg, s.getpeername()), file=sys.stderr,
                )
                s.send(next_msg)
```

Наконец, любые события с флагом POLLERR приводят к закрытию сокета сервером.

```
        elif flag & select.POLLERR:
            print('  exception on', s.getpeername(),
                  file=sys.stderr)
            # Прекратить прослушивание входных данных для
            # этого соединения
            poller.unregister(s)
            s.close()

            # Удалить очередь сообщений
            del message_queues[s]
```

В результате совместного выполнения сервера, основанного на использовании опроса, и клиента select_echo_multiclient.py (клиентская программа, в которой используется несколько сокетов) выводится следующая информация.

```
$ python3 select_poll_echo_server.py
starting up on localhost port 10000
waiting for the next event
waiting for the next event
waiting for the next event
waiting for the next event
  connection ('127.0.0.1', 61253)
waiting for the next event
  connection ('127.0.0.1', 61254)
waiting for the next event
  received b'This is the message. ' from ('127.0.0.1', 61253)
  received b'This is the message. ' from ('127.0.0.1', 61254)
waiting for the next event
  sending b'This is the message. ' to ('127.0.0.1', 61253)
  sending b'This is the message. ' to ('127.0.0.1', 61254)
waiting for the next event
('127.0.0.1', 61253) queue empty
('127.0.0.1', 61254) queue empty
waiting for the next event
  received b'It will be sent ' from ('127.0.0.1', 61253)
  received b'It will be sent ' from ('127.0.0.1', 61254)
waiting for the next event
  sending b'It will be sent ' to ('127.0.0.1', 61253)
  sending b'It will be sent ' to ('127.0.0.1', 61254)
waiting for the next event
('127.0.0.1', 61253) queue empty
('127.0.0.1', 61254) queue empty
waiting for the next event
  received b'in parts.' from ('127.0.0.1', 61253)
  received b'in parts.' from ('127.0.0.1', 61254)
waiting for the next event
  sending b'in parts.' to ('127.0.0.1', 61253)
  sending b'in parts.' to ('127.0.0.1', 61254)
waiting for the next event
('127.0.0.1', 61253) queue empty
```

```
('127.0.0.1', 61254) queue empty
waiting for the next event
  closing ('127.0.0.1', 61254)
waiting for the next event
  closing ('127.0.0.1', 61254)
waiting for the next event
```

11.4.4. Опции, специфические для платформы

Модуль `select` также предоставляет специфические для конкретных систем возможности, такие как `epoll` (от англ. *Edge and Trigger Polling*) — механизм отслеживания событий по перепаду и по значению, поддерживаемый в Linux, а также `kqueue` (от англ. *kernel queue*) — возможности доступа к очереди ядра и `kevent` (от англ. *kernel event*) — возможности доступа к событиям ядра, поддерживаемые в системах BSD. Для получения более подробной информации по этому вопросу обращайтесь к документации операционной системы.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `select`[13].
- `selectors` (раздел 11.3). Высокоуровневая абстракция, построенная поверх модуля `select`.
- *Socket Programming HOWTO*[14] (Gordon McMillan). Учебное руководство, включенное в документацию стандартной библиотеки.
- `socket` (раздел 11.2). Низкоуровневое сетевое взаимодействие.
- `SocketServer`. Фреймворк, предназначенный для создания сетевых серверных приложений.
- `asyncio` (раздел 10.5). Библиотека средств асинхронного ввода-вывода.
- *Unix Network Programming, Volume 1: The Sockets Networking API, Third Edition*, by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Addison-Wesley, 2004. ISBN-10: 0131411551.
- *Foundations of Python Network Programming, Third Edition*, by Brandon Rhodes and John Goerzen. Apress, 2014. ISBN-10: 1430258543.

11.5. `socketserver`: создание сетевых серверов

Модуль `socketserver` — это фреймворк, предназначенный для создания сетевых серверов. Он определяет классы, упрощающие обработку синхронных сетевых запросов (обработчик запросов блокируется до полного завершения обработки запроса) с использованием сокетов TCP, UDP, а также потоков и датаграмм Unix. Кроме того, он предоставляет примесные классы, позволяющие легко перестраивать серверы для работы с использованием отдельного потока или процесса для каждого запроса.

Ответственность за обработку запроса распределяется между классом сервера и классом обработчика запросов. Сервер отвечает за осуществление взаимо-

[13] `https://docs.python.org/3.5/library/select.html`

[14] `https://docs.python.org/howto/sockets.html`

действия (прослушивание сокета, принятие соединений и т.д.), а обработчик запросов — за протокол (интерпретация входящих данных, их обработка, а также отправка данных клиенту). Подобное разделение ответственности означает, что многие приложения могут использовать один из существующих серверных классов без каких-либо изменений и предоставлять класс обработчика запросов для работы с пользовательским протоколом.

11.5.1. Типы серверов

В модуле `socketserver` определены пять классов серверов. Класс `BaseServer` определяет API и не предназначен для инстанциализации и непосредственного использования. Класс `TCPServer` использует для взаимодействия сокеты TCP/IP. Класс `UDPServer` использует датаграммные сокеты. Классы `UnixStreamServer` и `UnixDatagramServer` используют сокеты доменов Unix и доступны лишь на платформах Unix.

11.5.2. Объекты сервера

Чтобы создать сервер, следует передать конструктору адрес, по которому будут прослушиваться запросы, и класс (а не экземпляр) обработчика. Формат адреса зависит от типа сервера и используемого семейства сокетов. Для получения более подробных сведений по этому вопросу обратитесь к разделу 11.2, посвященному описанию модуля `socket`.

Для обработки запросов с помощью вновь созданного объекта сервера следует использовать один из методов `handle_request()` или `serve_forever()`. Метод `serve_forever()` вызывает метод `handle_request()` в бесконечном цикле. Однако, если приложению необходимо интегрировать сервер с другим циклом событий или использовать функцию `select()` для мониторинга нескольких сокетов для различных серверов, оно может вызывать метод `handle_request()` непосредственно.

11.5.3. Реализация сервера

Обычно, когда создается сервер, достаточно использовать один из существующих классов и предоставить пользовательский класс обработчика запросов. Для всех остальных случаев класс `BaseServer` включает несколько методов, которые могут быть переопределены в подклассе.

- `verify_request(request,client_address)`. Возвращает значение `True`, если запрос необходимо обработать, или `False`, если его необходимо игнорировать. Например, перегруженный сервер может отказаться обслуживать запросы, поступающие с определенного диапазона адресов.
- `process_request(request,client_address)`. Вызывает метод `finish_request()` для фактического выполнения работы по обработке запроса. Данный метод также может создавать отдельные потоки или процессы, как это делают примесные классы.
- `finish_request(request,client_address)`. Создает экземпляр обработчика запросов, используя класс, переданный конструктору объекта сервера. Вызывает метод `handle()` обработчика для обработки запросов.

11.5.4. Обработчики запросов

Обработчики запросов выполняют бо́льшую часть работы, связанной с получением входящих запросов и принятия решений относительно того, какие действия должны предприниматься. Обработчик отвечает за реализацию протокола поверх слоя сокетов (т.е. HTTP, XML-RPC или AMQP). Обработчик читает запрос из входного канала данных, обрабатывает его и записывает отправляемый ответ. Для переопределения доступны три метода.

- `setup()`. Подготавливает обработчик к обработке запросов. В случае класса `StreamRequestHandler` создает файловые объекты для чтения и записи данных через сокет.
- `handle()`. Выполняет фактическую работу по обработке запроса. Анализирует входящий запрос, обрабатывает данные и отправляет ответ.
- `finish()`. Выполняет завершающие операции по освобождению всех ресурсов, созданных в процессе выполнения метода `setup()`.

Многие обработчики могут быть реализованы только с методом `handle()`.

11.5.5. Пример с эхо-сервером и эхо-клиентами

Этот пример реализует простую пару "сервер – обработчик запросов", которая принимает TCP-соединения и отправляет обратно клиенту посланные им данные. Ниже приведен код обработчика запросов.

Листинг 11.41. socketserver_echo.py

```
import logging
import sys
import socketserver

logging.basicConfig(level=logging.DEBUG,
                    format='%(name)s: %(message)s',
                    )


class EchoRequestHandler(socketserver.BaseRequestHandler):

    def __init__(self, request, client_address, server):
        self.logger = logging.getLogger('EchoRequestHandler')
        self.logger.debug('__init__')
        socketserver.BaseRequestHandler.__init__(self, request,
                                                 client_address,
                                                 server)
        return

    def setup(self):
        self.logger.debug('setup')
        return socketserver.BaseRequestHandler.setup(self)

    def handle(self):
        self.logger.debug('handle')
```

```
        # Эхо-сообщение клиенту
        data = self.request.recv(1024)
        self.logger.debug('recv()->"%s"', data)
        self.request.send(data)
        return

    def finish(self):
        self.logger.debug('finish')
        return socketserver.BaseRequestHandler.finish(self)
```

Единственным методом, который в действительности необходимо реализовать, является `EchoRequestHandler.handle()`, но, чтобы проиллюстрировать последовательность вызовов, в приведенный ниже код включены версии всех ранее описанных методов. Класс `EchoServer` не делает ничего сверх того, что делает класс `TCPServer`, за исключением протоколирования вызываемых методов.

```
class EchoServer(socketserver.TCPServer):

    def __init__(self, server_address,
                 handler_class=EchoRequestHandler,
                 ):
        self.logger = logging.getLogger('EchoServer')
        self.logger.debug('__init__')
        socketserver.TCPServer.__init__(self, server_address,
                                        handler_class)
        return

    def server_activate(self):
        self.logger.debug('server_activate')
        socketserver.TCPServer.server_activate(self)
        return

    def serve_forever(self, poll_interval=0.5):
        self.logger.debug('waiting for request')
        self.logger.info(
            'Handling requests, press <Ctrl-C> to quit'
        )
        socketserver.TCPServer.serve_forever(self, poll_interval)
        return

    def handle_request(self):
        self.logger.debug('handle_request')
        return socketserver.TCPServer.handle_request(self)

    def verify_request(self, request, client_address):
        self.logger.debug('verify_request(%s, %s)',
                          request, client_address)
        return socketserver.TCPServer.verify_request(
            self, request, client_address,
        )

    def process_request(self, request, client_address):
        self.logger.debug('process_request(%s, %s)',
                          request, client_address)
```

```
        return socketserver.TCPServer.process_request(
            self, request, client_address,
        )

    def server_close(self):
        self.logger.debug('server_close')
        return socketserver.TCPServer.server_close(self)

    def finish_request(self, request, client_address):
        self.logger.debug('finish_request(%s, %s)',
                          request, client_address)
        return socketserver.TCPServer.finish_request(
            self, request, client_address,
        )

    def close_request(self, request_address):
        self.logger.debug('close_request(%s)', request_address)
        return socketserver.TCPServer.close_request(
            self, request_address,
        )

    def shutdown(self):
        self.logger.debug('shutdown()')
        return socketserver.TCPServer.shutdown(self)
```

Все, что теперь остается сделать, — это добавить основную программу, которая настраивает сервер для работы в потоке и отправляет ему данные, чтобы проиллюстрировать, какие методы вызываются при обратной отправке данных клиенту.

```
if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0)  # позволить ядру назначить порт
    server = EchoServer(address, EchoRequestHandler)
    ip, port = server.server_address  # Какой порт был назначен?

    # Запустить сервер в потоке
    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # работать в фоновом режиме
    t.start()

    logger = logging.getLogger('client')
    logger.info('Server on %s:%s', ip, port)

    # Подключиться к серверу
    logger.debug('creating socket')
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logger.debug('connecting to server')
    s.connect((ip, port))

    # Отправить данные
    message = 'Hello, world'.encode()
```

```
    logger.debug('sending data: %r', message)
    len_sent = s.send(message)

    # Получить ответ
    logger.debug('waiting for response')
    response = s.recv(len_sent)
    logger.debug('response from server: %r', response)

    # Очистить ресурсы
    server.shutdown()
    logger.debug('closing socket')
    s.close()
    logger.debug('done')
    server.socket.close()
```

Выполнение программы приводит к следующим результатам.

```
$ python3 socketserver_echo.py

EchoServer: __init__
EchoServer: server_activate
EchoServer: waiting for request
EchoServer: Handling requests, press <Ctrl-C> to quit
client: Server on 127.0.0.1:55484
client: creating socket
client: connecting to server
client: sending data: b'Hello, world'
EchoServer: verify_request(<socket.socket fd=7,
family=AddressFamily
.AF_INET, type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1',
 55484), raddr=('127.0.0.1', 55485)>, ('127.0.0.1', 55485))
EchoServer: process_request(<socket.socket fd=7,
family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1',
55484), raddr=('127.0.0.1', 55485)>, ('127.0.0.1', 55485))
EchoServer: finish_request(<socket.socket fd=7,
family=AddressFamily
.AF_INET, type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1',
55484), raddr=('127.0.0.1', 55485)>, ('127.0.0.1', 55485))
EchoRequestHandler: __init__
EchoRequestHandler: setup
EchoRequestHandler: handle
client: waiting for response
EchoRequestHandler: recv()->"b'Hello, world'"
EchoRequestHandler: finish
client: response from server: b'Hello, world'
EchoServer: shutdown()
EchoServer: close_request(<socket.socket fd=7, family=AddressFamily.
AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 5
5484), raddr=('127.0.0.1', 55485)>)
```

```
client: closing socket
client: done
```

Примечание

Используемый номер порта будет меняться от одного запуска программы к другому, поскольку ядро автоматически выделяет доступный порт. Чтобы сервер слушал каждый раз один и тот же порт, укажите его номер в кортеже адреса вместо 0.

Ниже представлена компактная версия того же сервера без отладочных вызовов. В классе обработчика запросов необходимо предоставить лишь метод handle().

Листинг 11.42. socketserver_echo_simple.py

```
import socketserver


class EchoRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        # Эхо-сообщение клиенту
        data = self.request.recv(1024)
        self.request.send(data)
        return


if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0)  # позволить ядру назначить порт
    server = socketserver.TCPServer(address, EchoRequestHandler)
    ip, port = server.server_address  # Какой порт назначен?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # работать в фоновом режиме
    t.start()

    # Подключиться к серверу
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Отправить данные
    message = 'Hello, world'.encode()
    print('Sending : {!r}'.format(message))
    len_sent = s.send(message)

    # Получить ответ
    response = s.recv(len_sent)
    print('Received: {!r}'.format(response))

    # Очистить ресурсы
```

```
    server.shutdown()
    s.close()
    server.socket.close()
```

В данном случае никакой специальный класс сервера не требуется, поскольку TCPServer удовлетворяет всем требованиям, предъявляемым к серверу.

```
$ python3 socketserver_echo_simple.py

Sending : b'Hello, world'
Received: b'Hello, world'
```

11.5.6. Создание потоков и порождение процессов

Чтобы добавить в сервер поддержку потоков и процессов, следует включить в иерархию классов для сервера примесный класс. Примесные классы переопределяют метод process_request() для запуска нового потока или процесса и выполнения в нем всей работы, когда запрос готов к обработке.

Для потоков следует использовать класс ThreadingMixIn.

Листинг 11.43. socketserver_threaded.py

```
import threading
import socketserver


class ThreadedEchoRequestHandler(
        socketserver.BaseRequestHandler,
):

    def handle(self):
        # Эхо-сообщение клиенту
        data = self.request.recv(1024)
        cur_thread = threading.currentThread()
        response = b'%s: %s' % (cur_thread.getName().encode(),
                                data)
        self.request.send(response)
        return


class ThreadedEchoServer(socketserver.ThreadingMixIn,
                         socketserver.TCPServer,
                         ):
    pass


if __name__ == '__main__':
    import socket

    address = ('localhost', 0)  # позволить ядру назначить порт
    server = ThreadedEchoServer(address,
                                ThreadedEchoRequestHandler)
    ip, port = server.server_address  # Какой порт назначен?
```

```
    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # работать в фоновом режиме
    t.start()
    print('Server loop running in thread:', t.getName())

    # Подключиться к серверу
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Отправить данные
    message = b'Hello, world'
    print('Sending : {!r}'.format(message))
    len_sent = s.send(message)

    # Получить ответ
    response = s.recv(1024)
    print('Received: {!r}'.format(response))

    # Очистить ресурсы
    server.shutdown()
    s.close()
    server.socket.close()
```

Ответ этого потокового сервера включает идентификатор потока, в котором был обработан запрос.

```
$ python3 socketserver_threaded.py

Server loop running in thread: Thread-1
Sending : b'Hello, world'
Received: b'Thread-2: Hello, world'
```

Для отдельных процессов следует использовать класс `ForkingMixIn`.

Листинг 11.44. `socketserver_forking.py`

```
import os
import socketserver


class ForkingEchoRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        # Эхо-сообщение клиенту
        data = self.request.recv(1024)
        cur_pid = os.getpid()
        response = b'%d: %s' % (cur_pid, data)
        self.request.send(response)
        return


class ForkingEchoServer(socketserver.ForkingMixIn,
                        socketserver.TCPServer,
                        ):
```

```
    pass


if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0)  # позволить ядру назначить порт
    server = ForkingEchoServer(address,
                               ForkingEchoRequestHandler)
    ip, port = server.server_address  # Какой порт назначен?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # работать в фоновом режиме
    t.start()
    print('Server loop running in process:', os.getpid())

    # Подключиться к серверу
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Отправить данные
    message = 'Hello, world'.encode()
    print('Sending : {!r}'.format(message))
    len_sent = s.send(message)

    # Получить ответ
    response = s.recv(1024)
    print('Received: {!r}'.format(response))

    # Очистить ресурсы
    server.shutdown()
    s.close()
    server.socket.close()
```

В этом примере в ответ сервера включается идентификатор дочернего процесса.

```
$ python3 socketserver_forking.py

Server loop running in process: 22599
Sending : b'Hello, world'
Received: b'22600: Hello, world'
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `socketserver`[15].
- `socket` (раздел 11.2). Низкоуровневое сетевое взаимодействие.
- `select` (раздел 11.4). Средства низкоуровневого асинхронного ввода-вывода.

[15] https://docs.python.org/3.5/library/socketserver.html

- `asyncio` (раздел 10.5). Асинхронные операции ввода-вывода, цикл событий и средства параллельных вычислений.
- `SimpleXMLRPCServer`. Сервер XML-RPC, созданный с использованием модуля `socketserver`.
- *Unix Network Programming, Volume 1: The Sockets Networking API, Third Edition*, by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Addison-Wesley, 2004. ISBN-10: 0131411551.
- *Foundations of Python Network Programming, Third Edition*, by Brandon Rhodes and John Goerzen. Apress, 2014. ISBN-10: 1430258543.

Глава 12

Интернет

Интернет – непременный атрибут современных компьютерных технологий. Даже небольшие одноразовые сценарии часто взаимодействуют с удаленными службами для передачи или получения данных. Благодаря богатому набору инструментов Python, предназначенных для работы с веб-протоколами, этот язык хорошо приспособлен для программирования веб-приложений, выступающих либо в качестве клиентов, либо в качестве серверов.

Модуль `urllib.parse` (раздел 12.1) позволяет манипулировать строками URL-адресов, выделяя и объединяя их компоненты, и может быть полезным как в клиентах, так и в серверах.

Модуль `urllib.request` (раздел 12.2) реализует API удаленного поиска и извлечения содержимого.

Для предоставления серверу заполненной веб-формы, подготовленной с помощью модуля `urllib`, обычно используют HTTP-метод POST. Чтобы обеспечить соответствие стандарту форматирования сообщений, двоичные данные, передаваемые методом POST, должны предварительно кодироваться с помощью модуля `base64` (раздел 12.4).

Добросовестные клиенты, используемые в качестве поисковых веб-роботов для исследования содержимого множества сайтов, должны убеждаться в наличии соответствующих прав доступа с помощью модуля `useurllib.robotparser` (раздел 12.3), прежде чем подвергать интенсивной нагрузке удаленный сервер.

Чтобы создать собственный веб-сервер с помощью Python без привлечения внешних фреймворков, следует использовать модуль `http.server` (раздел 12.5) в качестве отправной точки. Этот модуль обеспечивает обработку протокола HTTP, поэтому единственное, что требуется от приложения, – это предоставить код, отвечающий на поступающие запросы.

Состоянием сеансов (сессий) на сервере можно управлять с помощью cookie-файлов, для создания и синтаксического анализа которых предназначен модуль `http.cookies` (раздел 12.6). Предоставляемая им полная поддержка контроля времени жизни, пути, домена и других параметров cookie-файлов упрощает конфигурирование сеансов.

Модуль `uuid` (раздел 12.8) используется для генерации уникальных значений идентификаторов ресурсов. Универсальные уникальные идентификаторы (UUID) – отличный кандидат для автоматической генерации унифицированных имен ресурсов (URN), где имя ресурса должно быть уникальным, но не обязано нести в себе какой-либо смысл.

Стандартная библиотека Python включает поддержку двух механизмов удаленного вызова веб-процедур. Текстовый формат обмена данными JSON (JavaScript Object Notation), используемый в технологии AJAX, и программные интерфейсы REST реализованы в модуле `json` (раздел 12.9). Этот модуль в равной мере пригоден для работы с сервером и клиентом. Полные библиотеки клиента и серве-

ра XML-RPC включены в модули `xmlrpc.client` (раздел 12.10) и `xmlrpc.server` (раздел 12.11) соответственно.

12.1. `urllib.parse`: разбиение URL-адресов на отдельные элементы

Модуль `urllib.parse` предоставляет функции, позволяющие манипулировать URL-адресами и их компонентами посредством разбиения или сборки.

12.1.1. Анализ URL-адресов

Функция `urlparse()` возвращает объект `ParseResult`, действующий подобно кортежу, включающему шесть элементов.

Листинг 12.1. `urllib_parse_urlparse.py`

```
from urllib.parse import urlparse

url = 'http://netloc/path;param?query=arg#frag'
parsed = urlparse(url)
print(parsed)
```

Элементами URL-адреса, доступными через интерфейс кортежа, являются схема (протокол), местоположение в сети, путь, параметры сегмента пути (отделяемые от пути точкой с запятой), строка запроса и идентификатор фрагмента.

```
$ python3 urllib_parse_urlparse.py

ParseResult(scheme='http', netloc='netloc', path='/path',
params='param', query='query=arg', fragment='frag')
```

Несмотря на то что возвращаемый объект напоминает кортеж, на самом деле он основан на именованном кортеже — подклассе кортежа, который поддерживает доступ к элементам URL-адреса посредством как именованных атрибутов, так и индексов. API атрибутов не только упрощает программистам работу, но и предлагает доступ к некоторым значениям, недоступным в API кортежей.

Листинг 12.2. `urllib_parse_urlparseattrs.py`

```
from urllib.parse import urlparse

url = 'http://user:pwd@NetLoc:80/path;param?query=arg#frag'
parsed = urlparse(url)
print('scheme  :', parsed.scheme)
print('netloc  :', parsed.netloc)
print('path    :', parsed.path)
print('params  :', parsed.params)
print('query   :', parsed.query)
print('fragment:', parsed.fragment)
print('username:', parsed.username)
print('password:', parsed.password)
print('hostname:', parsed.hostname)
print('port    :', parsed.port)
```

Значения username и password доступны тогда, когда они есть во входном URL-адресе, и устанавливаются в значение None в противном случае. Значение hostname совпадает со значением спецификатора местоположения в сети netloc с переводом всех букв в нижний регистр и без номера порта. Номер порта port преобразуется в целое число, если он указан, или принимает значение None в противном случае.

```
$ python3 urllib_parse_urlparseattrs.py

scheme  : http
netloc  : user:pwd@NetLoc:80
path    : /path
params  : param
query   : query=arg
fragment: frag
username: user
password: pwd
hostname: netloc
port    : 80
```

Функция urlsplit() аналогична функции urlparse(), но ведет себя несколько иначе, поскольку не выбирает параметры из URL. Это полезно в случае URL-адресов, соответствующих требованиям документа **RFC 2396**[1], который поддерживает параметры для каждого сегмента пути.

Листинг 12.3. urllib_parse_urlsplit.py

```
from urllib.parse import urlsplit

url = 'http://user:pwd@NetLoc:80/p1;para/p2;para?query=arg#frag'
parsed = urlsplit(url)
print(parsed)
print('scheme  :', parsed.scheme)
print('netloc  :', parsed.netloc)
print('path    :', parsed.path)
print('query   :', parsed.query)
print('fragment:', parsed.fragment)
print('username:', parsed.username)
print('password:', parsed.password)
print('hostname:', parsed.hostname)
print('port    :', parsed.port)
```

Поскольку параметры не выбираются, то атрибут params в этом случае отсутствует, и API кортежа отображает пять элементов вместо шести.

```
$ python3 urllib_parse_urlsplit.py

SplitResult(scheme='http', netloc='user:pwd@NetLoc:80',
path='/p1;para/p2;para', query='query=arg', fragment='frag')
scheme  : http
netloc  : user:pwd@NetLoc:80
```

[1] https://tools.ietf.org/html/rfc2396.html

```
path     : /p1;para/p2;para
query    : query=arg
fragment: frag
username: user
password: pwd
hostname: netloc
port     : 80
```

Чтобы упростить вычленение идентификатора фрагмента из URL-адреса, например для получения базового имени страницы, можно использовать функцию `urldefrag()`.

Листинг 12.4. `urllib_parse_urldefrag.py`

```
from urllib.parse import urldefrag

original = 'http://netloc/path;param?query=arg#frag'
print('original:', original)
d = urldefrag(original)
print('url      :', d.url)
print('fragment:', d.fragment)
```

Возвращаемым значением является объект `DefragResult` на основе именованного кортежа, содержащий базовый URL-адрес и фрагмент.

```
$ python3 urllib_parse_urldefrag.py

original: http://netloc/path;param?query=arg#frag
url      : http://netloc/path;param?query=arg
fragment: frag
```

12.1.2. Конструирование строки URL-адреса из элементов

Сборку отдельных компоенентов UR-адреса в единую строку можно осуществить несколькими способами. Объект `ParseResult` имеет метод `geturl()`.

Листинг 12.5. `urllib_parse_geturl.py`

```
from urllib.parse import urlparse

original = 'http://netloc/path;param?query=arg#frag'
print('ORIG  :', original)
parsed = urlparse(original)
print('PARSED:', parsed.geturl())
```

Метод `geturl()` работает лишь для объекта, возвращаемого функцией `urlparse()` или `urlsplit()`.

```
$ python3 urllib_parse_geturl.py

ORIG  : http://netloc/path;param?query=arg#frag
PARSED: http://netloc/path;param?query=arg#frag
```

Для конструирования URL-адреса на основе обычного кортежа, содержащего строки, следует использовать функцию `urlunparse()`.

Листинг 12.6. `urllib_parse_urlunparse.py`

```
from urllib.parse import urlparse, urlunparse

original = 'http://netloc/path;param?query=arg#frag'
print('ORIG  :', original)
parsed = urlparse(original)
print('PARSED:', type(parsed), parsed)
t = parsed[:]
print('TUPLE :', type(t), t)
print('NEW   :', urlunparse(t))
```

В то время как в качестве кортежа, представляющего элементы URL-адреса, может быть использован объект `ParseResult`, возвращаемый функцией `urlparse()`, в данном примере, для того чтобы продемонстрировать, что функция `urlunparse()` работает также с обычными кортежами, новый кортеж создается явным образом.

```
$ python3 urllib_parse_urlunparse.py

ORIG  : http://netloc/path;param?query=arg#frag
PARSED: <class 'urllib.parse.ParseResult'>
ParseResult(scheme='http', netloc='netloc', path='/path',
params='param', query='query=arg', fragment='frag')
TUPLE : <class 'tuple'> ('http', 'netloc', '/path', 'param',
'query=arg', 'frag')
NEW   : http://netloc/path;param?query=arg#frag
```

Если входной URL-адрес включает избыточные компоненты, то они могут быть исключены из реконструированной строки URL.

Листинг 12.7. `urllib_parse_urlunparseextra.py`

```
from urllib.parse import urlparse, urlunparse

original = 'http://netloc/path;?#'
print('ORIG  :', original)
parsed = urlparse(original)
print('PARSED:', type(parsed), parsed)
t = parsed[:]
print('TUPLE :', type(t), t)
print('NEW   :', urlunparse(t))
```

В данном случае параметры, строка запроса и фрагмент отсутствуют во входном URL-адресе. Новая строка URL выглядит иначе, чем исходная, однако эквивалентна ей согласно стандарту.

```
$ python3 urllib_parse_urlunparseextra.py

ORIG  : http://netloc/path;?#
PARSED: <class 'urllib.parse.ParseResult'>
ParseResult(scheme='http', netloc='netloc', path='/path',
```

```
params='', query='', fragment='')
TUPLE : <class 'tuple'> ('http', 'netloc', '/path', '', '', '')
NEW   : http://netloc/path
```

12.1.3. Объединение элементов

В дополнение к разбору URL-адресов на элементы модуль `urlib.parse` позволяет конструировать абсолютные URL-адреса путем объединения базового и относительного адресов с помощью функции `urljoin()`.

Листинг 12.8. `urllib_parse_urljoin.py`

```
from urllib.parse import urljoin

print(urljoin('http://www.example.com/path/file.html',
              'anotherfile.html'))
print(urljoin('http://www.example.com/path/file.html',
              '../anotherfile.html'))
```

В этом примере при вычислении второго URL-адреса учитывается относительная часть пути (`"../"`).

```
$ python3 urllib_parse_urljoin.py

http://www.example.com/path/anotherfile.html
http://www.example.com/anotherfile.html
```

Части пути, не являющиеся относительными, обрабатываются так же, как и с помощью функции `os.path.join()`.

Листинг 12.9. `urllib_parse_urljoin_with_path.py`

```
from urllib.parse import urljoin

print(urljoin('http://www.example.com/path/',
              '/subpath/file.html'))
print(urljoin('http://www.example.com/path/',
              'subpath/file.html'))
```

Если присоединяемый к URL-адресу путь является каталогом (начинается с символа косой черты "/"), то функция `urljoin()` удаляет часть пути URL до верхнего уровня. В противном случае новая часть пути присоединяется к концу существующей.

```
$ python3 urllib_parse_urljoin_with_path.py

http://www.example.com/subpath/file.html
http://www.example.com/path/subpath/file.html
```

12.1.4. Кодирование параметров запроса

Прежде чем параметры запроса можно будет добавить в строку URL, их необходимо преобразовать соответствующим образом.

Листинг 12.10. urllib_parse_urlencode.py

```
from urllib.parse import urlencode

query_args = {
    'q': 'query string',
    'foo': 'bar',
}
encoded_args = urlencode(query_args)
print('Encoded:', encoded_args)
```

Процесс преобразования включает замену таких специальных символов, как пробелы, что гарантирует их корректную передачу на сервер с использованием формата, удовлетворяющего требованиям стандарта.

```
$ python3 urllib_parse_urlencode.py

Encoded: q=query+string&foo=bar
```

Если какое-либо значение в строке запроса является последовательностью, то функции `urlencode()` необходимо дополнительно передать флаг `doseq`, имеющий значение `True`.

Листинг 12.11. urllib_parse_urlencode_doseq.py

```
from urllib.parse import urlencode

query_args = {
    'foo': ['foo1', 'foo2'],
}
print('Single  :', urlencode(query_args))
print('Sequence:', urlencode(query_args, doseq=True))
```

Результат представляет собой строку запроса, в которой с одним именем ассоциировано несколько значений.

```
$ python3 urllib_parse_urlencode_doseq.py

Single  : foo=%5B%27foo1%27%2C+%27foo2%27%5D
Sequence: foo=foo1&foo=foo2
```

Для декодирования строки запроса следует использовать функцию `parse_qs()` или `parse_qsl()`.

Листинг 12.12. urllib_parse_parse_qs.py

```
from urllib.parse import parse_qs, parse_qsl

encoded = 'foo=foo1&foo=foo2'

print('parse_qs :', parse_qs(encoded))
print('parse_qsl:', parse_qsl(encoded))
```

Функция parse_qs() возвращает словарь, сопоставляющий имена со значениями, тогда как функция parse_qsl() возвращает список кортежей, каждый из которых содержит имя и значение.

```
$ python3 urllib_parse_parse_qs.py

parse_qs : {'foo': ['foo1', 'foo2']}
parse_qsl: [('foo', 'foo1'), ('foo', 'foo2')]
```

Входящие в строку запроса специальные символы, при анализе которых на стороне сервера могут возникнуть проблемы, должны экранироваться при передаче их функции urlencode(). Безопасные локальные версии строк можно получить непосредственно, используя функции quote() или quote_plus().

Листинг 12.13. urllib_parse_quote.py

```
from urllib.parse import quote, quote_plus, urlencode

url = 'http://localhost:8080/~hellmann/'
print('urlencode() :', urlencode({'url': url}))
print('quote()     :', quote(url))
print('quote_plus():', quote_plus(url))
```

Реализация экранирования в функции quote_plus() отличается большей строгостью в отношении замены символов.

```
$ python3 urllib_parse_quote.py

urlencode() : url=http%3A%2F%2Flocalhost%3A8080%2F%7Ehellmann%2F
quote()     : http%3A//localhost%3A8080/%7Ehellmann/
quote_plus(): http%3A%2F%2Flocalhost%3A8080%2F%7Ehellmann%2F
```

Чтобы обратить операцию экранирования символов, следует использовать функцию unquote() или unquote_plus() соответственно.

Листинг 12.14. urllib_parse_unquote.py

```
from urllib.parse import unquote, unquote_plus

print(unquote('http%3A//localhost%3A8080/%7Ehellmann/'))
print(unquote_plus(
    'http%3A%2F%2Flocalhost%3A8080%2F%7Ehellmann%2F'
))
```

Кодированное значение преобразуется обратно в обычную строку URL.

```
$ python3 urllib_parse_unquote.py

http://localhost:8080/~hellmann/
http://localhost:8080/~hellmann/
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `urllib.parse`[2].
- `urllib.request` (раздел 12.2). Получение содержимого ресурса, идентифицированного с помощью адреса URL.
- **RFC 1738**[3]. *Uniform Resource Locator (URL) syntax.*
- **RFC 1808**[4]. *Relative URLs.*
- **RFC 2396**[5]. *Uniform Resource Identifier (URI) generic syntax.*
- **RFC 3986**[6]. *Uniform Resource Identifier (URI) syntax.*

12.2. `urllib.request`: доступ к сетевым ресурсам

Модуль `urllib.request` предоставляет API для использования интернет-ресурсов, идентифицируемых своими URL-адресами. Он спроектирован таким образом, чтобы отдельные приложения могли расширять его для поддержки новых протоколов или внесения изменений в существующие протоколы (например, для поддержки базовой HTTP-аутентификации).

12.2.1. Метод HTTP GET

Примечание

Код тестового сервера для последующих примеров содержится в файле `http_server_GET.py`, находящемся в папке примеров для модуля `http.server` (раздел 12.5). Сервер следует запустить в одном окне, а выполнять примеры — в другом.

Использование модуля `urllib.request` проще всего продемонстрировать на примере операции HTTP GET. Чтобы получить файловый дескриптор удаленных данных, следует передать URL-адрес методу `urlopen()`.

Листинг 12.15. `urllib_request_urlopen.py`

```
from urllib import request

response = request.urlopen('http://localhost:8080/')
print('RESPONSE:', response)
print('URL     :', response.geturl())

headers = response.info()
print('DATE    :', headers['date'])
print('HEADERS :')
print('---------')
print(headers)

data = response.read().decode('utf-8')
```

[2] https://docs.python.org/3.5/library/urllib.parse.html
[3] https://tools.ietf.org/html/rfc1738.html
[4] https://tools.ietf.org/html/rfc1808.html
[5] https://tools.ietf.org/html/rfc2396.html
[6] https://tools.ietf.org/html/rfc3986.html

```
print('LENGTH  :', len(data))
print('DATA    :')
print('---------')
print(data)
```

Сервер получает входящие значения и форматирует ответ в виде простого текста, отправляемого обратно клиенту. Значение, возвращаемое функцией `urlopen()`, предоставляет доступ к заголовкам ответа HTTP-сервера через метод `info()`, а доступ к данным для удаленного ресурса обеспечивают, например, такие методы, как `read()` и `readlines()`.

```
$ python3 urllib_request_urlopen.py

RESPONSE: <http.client.HTTPResponse object at 0x101744d68>
URL     : http://localhost:8080/
DATE    : Sat, 08 Oct 2016 18:08:54 GMT
HEADERS :
---------
Server: BaseHTTP/0.6 Python/3.5.2
Date: Sat, 08 Oct 2016 18:08:54 GMT
Content-Type: text/plain; charset=utf-8


LENGTH  : 349
DATA    :
---------
CLIENT VALUES:
client_address=('127.0.0.1', 58420) (127.0.0.1)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=Python-urllib/3.5
```

Функция `urlopen()` возвращает итерируемый объект, подобный файлу.

Листинг 12.16. `urllib_request_urlopen_iterator.py`

```
from urllib import request

response = request.urlopen('http://localhost:8080/')
for line in response:
    print(line.decode('utf-8').rstrip())
```

В этом примере перед выводом информации на печать из нее удаляются символы новой строки и перевода каретки.

```
$ python3 urllib_request_urlopen_iterator.py

CLIENT VALUES:
client_address=('127.0.0.1', 58444) (127.0.0.1)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=Python-urllib/3.5
```

12.2.2. Кодирование параметров запроса

Параметры запроса могут передаваться серверу посредством кодирования с помощью функции `urllib.parse.urlencode()` и присоединения к URL-адресу.

Листинг 12.17. `urllib_request_http_get_args.py`

```
from urllib import parse
from urllib import request

query_args = {'q': 'query string', 'foo': 'bar'}
encoded_args = parse.urlencode(query_args)
print('Encoded:', encoded_args)

url = 'http://localhost:8080/?' + encoded_args
print(request.urlopen(url).read().decode('utf-8'))
```

Список значений, отображаемых в выводе этого примера, содержит закодированные параметры запроса.

```
$ python urllib_request_http_get_args.py

Encoded: q=query+string&foo=bar
CLIENT VALUES:
client_address=('127.0.0.1', 58455) (127.0.0.1)
command=GET
path=/?q=query+string&foo=bar
real path=/
query=q=query+string&foo=bar
request_version=HTTP/1.1
```

```
SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=Python-urllib/3.5
```

12.2.3. Метод HTTP POST

Примечание

Код тестового сервера для последующих примеров содержится в файле `http_server_GET.py`, находящемся в папке примеров для модуля `http.server` (раздел 12.5). Сервер следует запустить в одном окне, а выполнять примеры — в другом.

Чтобы отправить удаленному серверу данные, закодированные в виде формы, используя метод POST вместо метода GET, следует передать закодированные параметры запроса в виде данных функции `urlopen()`.

Листинг 12.18. `urllib_request_urlopen_post.py`

```
from urllib import parse
from urllib import request

query_args = {'q': 'query string', 'foo': 'bar'}
encoded_args = parse.urlencode(query_args).encode('utf-8')
url = 'http://localhost:8080/'
print(request.urlopen(url, encoded_args).read().decode('utf-8'))
```

Сервер может декодировать данные формы и получить доступ к индивидуальным значениям по имени.

```
$ python3 urllib_request_urlopen_post.py
Client: ('127.0.0.1', 58568)
User-agent: Python-urllib/3.5
Path: /
Form data:
      q=query string
      foo=bar
```

12.2.4. Добавление исходящих заголовков

Функция `urlopen()` скрывает некоторые детали создания и обработки запроса. Возможности более точного контроля обеспечивает непосредственное использование экземпляра `Request`. Например, в исходящий запрос могут быть добавлены пользовательские заголовки, позволяющие управлять форматом возвращаемых данных, указывать локальную кешированную версию документа и сообщать удаленному серверу имя взаимодействующей с ним клиентской программы.

Как следует из результатов предыдущего примера, значение по умолчанию заголовка User-agent состоит из константы Python-urllib и номера версии интерпретатора Python. При создании приложения, которое будет получать доступ к веб-ресурсам, принадлежащим другим владельцам, считается правилом хорошего тона включать в запрос информацию о реальном владельце агента, что упрощает ведение аналитики посещаемости сайтов. Кроме того, использование пользовательского заголовка агента позволяет владельцам сайтов контролировать действия веб-роботов с помощью файла *robots.txt* (см. описание модуля http.robotparser).

Листинг 12.19. urllib_request_request_header.py

```
from urllib import request

r = request.Request('http://localhost:8080/')
r.add_header(
    'User-agent',
    'PyMOTW (https://pymotw.com/)',
)

response = request.urlopen(r)
data = response.read().decode('utf-8')
print(data)
```

После создания объекта Request, но до открытия запроса следует установить значение пользовательского заголовка с помощью метода add_header(). Пользовательское значение отображается в последней строке вывода.

```
$ python3 urllib_request_request_header.py

CLIENT VALUES:
client_address=('127.0.0.1', 58585) (127.0.0.1)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0

HEADERS RECEIVED:
Accept-Encoding=identity
Connection=close
Host=localhost:8080
User-Agent=PyMOTW (https://pymotw.com/)
```

12.2.5. Отправка формы данных на сервер

Создавая объект Request, можно указать в аргументе data исходящие данные, подлежащие выгрузке на сервер.

Листинг 12.20. urllib_request_request_post.py

```
from urllib import parse
from urllib import request

query_args = {'q': 'query string', 'foo': 'bar'}

r = request.Request(
    url='http://localhost:8080/',
    data=parse.urlencode(query_args).encode('utf-8'),
)
print('Request method :', r.get_method())
r.add_header(
    'User-agent',
    'PyMOTW (https://pymotw.com/)',
)

print()
print('OUTGOING DATA:')
print(r.data)

print()
print('SERVER RESPONSE:')
print(request.urlopen(r).read().decode('utf-8'))
```

При наличии этого аргумента HTTP-метод GET, используемый объектом Request, автоматически заменяется методом POST.

```
$ python3 urllib_request_request_post.py

Request method : POST
OUTGOING DATA:
b'q=query+string&foo=bar'

SERVER RESPONSE:
Client: ('127.0.0.1', 58613)
User-agent: PyMOTW (https://pymotw.com/)
Path: /
Form data:
      foo=bar
      q=query string
```

12.2.6. Выгрузка файлов

Кодирование файлов для выгрузки на сервер требует выполнения чуть большего объема работы, чем при использовании простых форм. Для этого в теле запроса необходимо сконструировать полное MIME-сообщение, чтобы сервер мог отличать поля входящей формы от выгружаемых файлов.

Листинг 12.21. urllib_request_upload_files.py

```
import io
import mimetypes
```

```
from urllib import request
import uuid


class MultiPartForm:
    """Накапливает данные, используемые при публикации формы."""

    def __init__(self):
        self.form_fields = []
        self.files = []
        # Использовать случайную байтовую строку большой длины
        # для разделения отдельных частей MIME-данных
        self.boundary = uuid.uuid4().hex.encode('utf-8')
        return

    def get_content_type(self):
        return 'multipart/form-data; boundary={}'.format(
            self.boundary.decode('utf-8'))

    def add_field(self, name, value):
        """Добавить простое поле в форму данных."""
        self.form_fields.append((name, value))

    def add_file(self, fieldname, filename, fileHandle,
                 mimetype=None):
        """Добавить выгружаемый файл."""
        body = fileHandle.read()
        if mimetype is None:
            mimetype = (
                mimetypes.guess_type(filename)[0] or
                'application/octet-stream'
            )
        self.files.append((fieldname, filename, mimetype, body))
        return

    @staticmethod
    def _form_data(name):
        return ('Content-Disposition: form-data; '
                'name="{}"\r\n').format(name).encode('utf-8')

    @staticmethod
    def _attached_file(name, filename):
        return ('Content-Disposition: file; '
                'name="{}"; filename="{}"\r\n').format(
                    name, filename).encode('utf-8')

    @staticmethod
    def _content_type(ct):
        return 'Content-Type: {}\r\n'.format(ct).encode('utf-8')

    def __bytes__(self):
        """Вернуть байтовую строку, представляющую данные формы,
        включая присоединенные файлы.
        """
```

```
        buffer = io.BytesIO()
        boundary = b'--' + self.boundary + b'\r\n'

        # Добавить поля формы
        for name, value in self.form_fields:
            buffer.write(boundary)
            buffer.write(self._form_data(name))
            buffer.write(b'\r\n')
            buffer.write(value.encode('utf-8'))
            buffer.write(b'\r\n')

        # Добавить выгружаемые файлы
        for f_name, filename, f_content_type, body in self.files:
            buffer.write(boundary)
            buffer.write(self._attached_file(f_name, filename))
            buffer.write(self._content_type(f_content_type))
            buffer.write(b'\r\n')
            buffer.write(body)
            buffer.write(b'\r\n')

        buffer.write(b'--' + self.boundary + b'--\r\n')
        return buffer.getvalue()


if __name__ == '__main__':
    # Создать форму с простыми полями
    form = MultiPartForm()
    form.add_field('firstname', 'Doug')
    form.add_field('lastname', 'Hellmann')

    # Добавить фиктивный файл
    form.add_file(
        'biography', 'bio.txt',
        fileHandle=io.BytesIO(b'Python developer and blogger.'))

    # Создать запрос, включающий байтовую строку
    # для выгружаемых данных
    data = bytes(form)
    r = request.Request('http://localhost:8080/', data=data)
    r.add_header(
        'User-agent',
        'PyMOTW (https://pymotw.com/)',
    )
    r.add_header('Content-type', form.get_content_type())
    r.add_header('Content-length', len(data))

    print()
    print('OUTGOING DATA:')
    for name, value in r.header_items():
        print('{}: {}'.format(name, value))
    print()
    print(r.data.decode('utf-8'))
```

```
    print()
    print('SERVER RESPONSE:')
    print(request.urlopen(r).read().decode('utf-8'))
```

```
$ python3 urllib_request_upload_files.py

OUTGOING DATA:
User-agent: PyMOTW (https://pymotw.com/)
Content-type: multipart/form-data;
    boundary=d99b5dc60871491b9d63352eb24972b4
Content-length: 389

--d99b5dc60871491b9d63352eb24972b4
Content-Disposition: form-data; name="firstname"

Doug
--d99b5dc60871491b9d63352eb24972b4
Content-Disposition: form-data; name="lastname"

Hellmann
--d99b5dc60871491b9d63352eb24972b4
Content-Disposition: file; name="biography";
    filename="bio.txt"
Content-Type: text/plain

Python developer and blogger.
--d99b5dc60871491b9d63352eb24972b4--


SERVER RESPONSE:
Client: ('127.0.0.1', 59310)
User-agent: PyMOTW (https://pymotw.com/)
Path: /
Form data:
    Uploaded biography as 'bio.txt' (29 bytes)
    firstname=Doug
    lastname=Hellmann
```

12.2.7. Создание пользовательских обработчиков протоколов

Модуль `urllib.request` имеет встроенную поддержку протоколов HTTP(S), FTP и локального доступа к файлам. Для добавления поддержки URL-адресов других типов необходимо зарегистрировать обработчики соответствующих протоколов. Например, чтобы организовать поддержку URL-адресов, представляющих произвольные файлы на удаленных серверах NFS, и при этом не требовать от пользователя предварительного монтирования пути к файлу для получения доступа к нему, следует создать подкласс `BaseHandler` с методом `nfs_open()`.

Специфический для протокола метод `open()` получает единственный аргумент — экземпляр `Request` — и возвращает объект, который имеет метод `read()`

для чтения данных, метод `info()`, возвращающий заголовок ответов, и метод `geturl()`, возвращающий фактический URL-адрес файла, который необходимо прочитать. Самый простой способ удовлетворить этим требованиям — создать экземпляр `urllib.response.addinfourl`, передав конструктору заголовки, URL-адрес и дескриптор файла.

Листинг 12.22. `urllib_request_nfs_handler.py`

```
import io
import mimetypes
import os
import tempfile
from urllib import request
from urllib import response


class NFSFile:

    def __init__(self, tempdir, filename):
        self.tempdir = tempdir
        self.filename = filename
        with open(os.path.join(tempdir, filename), 'rb') as f:
            self.buffer = io.BytesIO(f.read())

    def read(self, *args):
        return self.buffer.read(*args)

    def readline(self, *args):
        return self.buffer.readline(*args)

    def close(self):
        print('\nNFSFile:')
        print('  unmounting {}'.format(
            os.path.basename(self.tempdir)))
        print('  when {} is closed'.format(
            os.path.basename(self.filename)))


class FauxNFSHandler(request.BaseHandler):

    def __init__(self, tempdir):
        self.tempdir = tempdir
        super().__init__()

    def nfs_open(self, req):
        url = req.full_url
        directory_name, file_name = os.path.split(url)
        server_name = req.host
        print('FauxNFSHandler simulating mount:')
        print('  Remote path: {}'.format(directory_name))
        print('  Server     : {}'.format(server_name))
        print('  Local path : {}'.format(
            os.path.basename(tempdir)))
```

```
        print('  Filename   : {}'.format(file_name))
        local_file = os.path.join(tempdir, file_name)
        fp = NFSFile(tempdir, file_name)
        content_type = (
            mimetypes.guess_type(file_name)[0] or
            'application/octet-stream'
        )
        stats = os.stat(local_file)
        size = stats.st_size
        headers = {
            'Content-type': content_type,
            'Content-length': size,
        }
        return response.addinfourl(fp, headers,
                                   req.get_full_url())


if __name__ == '__main__':
    with tempfile.TemporaryDirectory() as tempdir:
        # Создать содержимое временного файла
        filename = os.path.join(tempdir, 'file.txt')
        with open(filename, 'w', encoding='utf-8') as f:
            f.write('Contents of file.txt')

        # Создать объект доступа к ресурсам с помощью нашего
        # NFS-обработчика и зарегистрировать его для
        # использования по умолчанию
        opener = request.build_opener(FauxNFSHandler(tempdir))
        request.install_opener(opener)

        # Открыть файл, используя URL-адрес
        resp = request.urlopen(
            'nfs://remote_server/path/to/the/file.txt'
        )
        print()
        print('READ CONTENTS:', resp.read())
        print('URL          :', resp.geturl())
        print('HEADERS:')
        for name, value in sorted(resp.info().items()):
            print('  {:<15} = {}'.format(name, value))
        resp.close()
```

Классы `FauxNFSHandler` и `NFSFile` выводят сообщения для иллюстрации того, куда именно действительная реализация поместила бы вызовы монтирования и размонтирования тома. Поскольку данный пример — всего лишь имитация, экземпляр `FauxNFSHandler` инициализируется временным каталогом, в котором он будет искать все свои файлы.

```
$ python3 urllib_request_nfs_handler.py
FauxNFSHandler simulating mount:
Remote path: nfs://remote_server/path/to/the
Server : remote_server
Local path : tmprucom5sb
```

```
Filename : file.txt
READ CONTENTS: b'Contents of file.txt'
URL : nfs://remote_server/path/to/the/file.txt
HEADERS:
Content-length = 20
Content-type = text/plain
NFSFile:
unmounting tmprucom5sb
when file.txt is closed
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `urllib.request`[7].
- `urllib.parse` (раздел 12.1). Работа непосредственно со строкой URL.
- *Form content types*[8]. Спецификация W3C, стандартизирующая выгрузку файлов или больших объемов данных посредством HTTP-форм.
- `mimetypes`. Сопоставляет имена файлов с MIME-типами, соответствующими различным типам передаваемых данных.
- `Requests`[9]. Сторонняя библиотека для работы с протоколом HTTP, предлагающая улучшенную поддержку безопасных соединений и простой в использовании API. Основная команда разработчиков Python рекомендует этот модуль к использованию отчасти потому, что он получает обновления безопасности гораздо чаще, чем стандартная библиотека.

12.3. `urllib.robotparser`: управление действиями веб-роботов

Модуль `robotparser` реализует синтаксический анализатор файлов *robots.txt*, в том числе функцию, проверяющую, может ли пользовательский агент получить доступ к ресурсу. Этот модуль предназначен для использования в добросовестных поисковых агентах и других веб-роботах, действия которых должны контролироваться или ограничиваться тем или иным образом.

12.3.1. Файл *robots.txt*

Файл *robots.txt* — это простой текстовый файл, позволяющий контролировать попытки доступа к ресурсам посредством компьютерных программ, осуществляющих автоматический перебор веб-страниц с целью их индексации и сбора информации (так называемые "веб-пауки", "краулеры" и другие веб-роботы). Этот файл содержит записи, каждая из которых включает идентификатор пользовательского агента и список URL-адресов (или их префиксов), доступ к которым не разрешен для данного агента.

В следующем листинге представлено содержимое файла *robots.txt* для адреса `https://pymotw.com/`.

[7] `https://docs.python.org/3.5/library/urllib.request.html`

[8] `www.w3.org/TR/REC-html40/interact/forms.html#h-17.13.4`

[9] `https://pypi.python.org/pypi/requests`

Листинг 12.23. `robots.txt`

```
Sitemap: https://pymotw.com/sitemap.xml
User-agent: *
Disallow: /admin/
Disallow: /downloads/
Disallow: /media/
Disallow: /static/
Disallow: /codehosting/
```

Этот файл предотвращает доступ пользовательских агентов к некоторым ресурсоемким разделам сайта во избежание перегрузки сервера, которая могла бы возникнуть при попытках индексирования ресурсов поисковыми механизмами. Более полные примеры использования файла `robots.txt` можно найти на сайте Web Robots[10].

12.3.2. Тестирование прав доступа

Ниже на примере представленных ранее данных показано, каким образом простой робот может проверить, разрешена ли ему загрузка страницы, используя для этого метод `RobotFileParser.can_fetch()`.

Листинг 12.24. `urllib_robotparser_simple.py`

```
from urllib import parse
from urllib import robotparser

AGENT_NAME = 'PyMOTW'
URL_BASE = 'https://pymotw.com/'
parser = robotparser.RobotFileParser()
parser.set_url(parse.urljoin(URL_BASE, 'robots.txt'))
parser.read()

PATHS = [
    '/',
    '/PyMOTW/',
    '/admin/',
    '/downloads/PyMOTW-1.92.tar.gz',
]

for path in PATHS:
    print('{!r:>6} : {}'.format(
        parser.can_fetch(AGENT_NAME, path), path))
    url = parse.urljoin(URL_BASE, path)
    print(,{!r:>6} : {}'.format(
        parser.can_fetch(AGENT_NAME, url), url))
    print()
```

В качестве аргумента `URL` метода `can_fetch()` может выступать путь, заданный относительно корневой папки сайта, или полный URL-адрес.

[10] www.robotstxt.org/orig.html

```
$ python3 urllib_robotparser_simple.py

  True : /
  True : https://pymotw.com/

  True : /PyMOTW/
  True : https://pymotw.com/PyMOTW/

 False : /admin/
 False : https://pymotw.com/admin/

 False : /downloads/PyMOTW-1.92.tar.gz
 False : https://pymotw.com/downloads/PyMOTW-1.92.tar.gz
```

12.3.3. Длительно выполняющиеся веб-роботы

Приложения, которые тратят много времени на обработку загружаемых ресурсов или простаивают в промежутках между загрузками, должны периодически проверять наличие новых файлов *robots.txt* на основании срока жизни уже загруженного содержимого. Сроком жизни нельзя управлять автоматически, но имеются вспомогательные методы, позволяющие отслеживать значение этого параметра.

Листинг 12.25. urllib_robotparser_longlived.py

```
from urllib import robotparser
import time

AGENT_NAME = 'PyMOTW'
parser = robotparser.RobotFileParser()
# Использование локальной копии
parser.set_url('file:robots.txt')
parser.read()
parser.modified()

PATHS = [
    '/',
    '/PyMOTW/',
    '/admin/',
    '/downloads/PyMOTW-1.92.tar.gz',
]

for path in PATHS:
    age = int(time.time() - parser.mtime())
    print('age:', age, end=' ')
    if age > 1:
        print('rereading robots.txt')
        parser.read()
        parser.modified()
    else:
        print()
    print('{!r:>6} : {}'.format(
```

```
        parser.can_fetch(AGENT_NAME, path), path))
    # Имтация задержки в процессе обработки
    time.sleep(1)
    print()
```

В этом экстремальном примере новый файл *robots.txt* загружается в том случае, если время жизни существующего файла превышает 1 секунду.

```
$ python3 urllib_robotparser_longlived.py

age: 0
  True : /

age: 1
  True : /PyMOTW/

age: 2 rereading robots.txt
 False : /admin/

age: 1
 False : /downloads/PyMOTW-1.92.tar.gz
```

Хорошо организованная версия длительно выполняющегося приложения может запрашивать время последнего изменения файла, прежде чем загружать сам файл. Однако обычно файлы *robots.txt* имеют довольно небольшие размеры, и поэтому повторная загрузка всего документа не является слишком дорогостоящей операцией.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `urllib.robotparser`[11].
- *Web Robots Pages*[12]. Описание формата файла *robots.txt*.

12.4. base64: кодирование двоичных данных с помощью ASCII

Модуль `base64` содержит функции, позволяющие преобразовывать двоичные данные в текстовое представление с использованием подмножества символов ASCII, пригодного для передачи по сети посредством простых текстовых протоколов. Кодировки Base64, Base32, Base16 и Base85 преобразуют 8-битовые байты в значения из диапазона печатаемых символов ASCII, добавляя в них дополнительные биты для достижения совместимости с системами, которые поддерживают только ASCII-данные, такими как SMTP. Значения Base соответствуют длине алфавита, используемого в каждой кодировке. Вариации исходных кодировок, безопасные в отношении включения в URI-адреса, используют несколько иные алфавиты.

[11] `https://docs.python.org/3.5/library/urllib.robotparser.html`
[12] `www.robotstxt.org/orig.html`

12.4.1. Кодировка Base64

В следующем листинге представлен простой пример кодирования текста.

Листинг 12.26. base64_b64encode.py

```
import base64
import textwrap

# Загрузить этот исходный файл и отделить заголовок
with open(__file__, 'r', encoding='utf-8') as input:
    raw = input.read()
    initial_data = raw.split('#end_pymotw_header')[1]

byte_string = initial_data.encode('utf-8')
encoded_data = base64.b64encode(byte_string)

num_initial = len(byte_string)

# Никогда не будет больше 2 дополнительных байтов
padding = 3 - (num_initial % 3)

print('{} bytes before encoding'.format(num_initial))
print('Expect {} padding bytes'.format(padding))
print('{} bytes after encoding\n'.format(len(encoded_data)))
print(encoded_data)
```

Входные данные должны быть байтовой строкой, поэтому строка Unicode предварительно преобразуется в формат UTF-8. Как следует из результатов, исходный текст, имеющий в кодировке UTF-8 длину 185 байт, после преобразования расширился до 248 байт.

Примечание

В результирующих закодированных данных символы перевода каретки отсутствуют, и эти данные представлены ниже в виде нескольких строк лишь для того, чтобы они уместились по ширине страницы.

```
$ python3 base64_b64encode.py

185 bytes before encoding
Expect 1 padding bytes
248 bytes after encoding

b'CgppbXBvcnQgYmFzZTY0CmltcG9ydCB0ZXh0d3JhcAoKIyBMb2FkIHRoaXMgc2
91cmNlIGZpbGUgYW5kIHN0cmlwIHRoZSBoZWFkZXIuCndpdGggb3BlbihfX2ZpbG
VfXywgJ3InLCBlbmNvZGluZz0ndXRmLTgnKSBhcyBpbnB1dDoKICAgIHJhdyA9IG
lucHV0LnJlYWQoKQogICAgaW5pdGlhbF9kYXRhID0gcmF3LnNwbGl0KCc='
```

12.4.2. Декодирование формата Base64

Функция `b64decode()` обеспечивает приведение кодированной строки к первоначальному виду путем преобразования каждых четырех ее байтов в исходные три байта, используя таблицу поиска.

Листинг 12.27. base64_b64decode.py

```
import base64

encoded_data = b'VGhpcyBpcyB0aGUgZGF0YSwgaW4gdGhlIGNsZWFyLg=='
decoded_data = base64.b64decode(encoded_data)
print('Encoded :', encoded_data)
print('Decoded :', decoded_data)
```

В процессе кодирования каждая последовательность из 24 битов исходных данных (3 байта) расширяется до 4 байтов выходных данных. Знаки равенства в конце — это заполнители, которые добавляются в конце преобразованных данных, если количество битов во входных данных не кратно 24.

```
$ python3 base64_b64decode.py

Encoded : b'VGhpcyBpcyB0aGUgZGF0YSwgaW4gdGhlIGNsZWFyLg=='
Decoded : b'This is the data, in the clear.'
```

Функция `b64decode()` возвращает байтовую строку. Если заведомо известно, что содержимое является текстом, то эту байтовую строку можно преобразовать в объект Unicode. Но поскольку по самой своей сущности кодировка Base64 предназначена для преобразования двоичных данных, предположение о текстовой природе данных не всегда безопасно.

12.4.3. Вариации, безопасные для использования в URL-адресах

В связи с тем что используемый по умолчанию алфавит кодирования Base64 может включать символы + и / и эти два символа используются в URL-адресах, нередко приходится прибегать к альтернативным кодировкам, в которых эти символы заменены другими символами.

Листинг 12.28. base64_urlsafe.py

```
import base64

encodes_with_pluses = b'\xfb\xef'
encodes_with_slashes = b'\xff\xff'

for original in [encodes_with_pluses, encodes_with_slashes]:
    print('Original         :', repr(original))
    print('Standard encoding:',
          base64.standard_b64encode(original))
    print('URL-safe encoding:',
          base64.urlsafe_b64encode(original))
    print()
```

Знак + заменяется знаком -, а знак / — символом подчеркивания (_). Остальной алфавит остается прежним.

```
Original         : b'\xfb\xef'
Standard encoding: b'++8='
```

```
URL-safe encoding: b'--8='

Original         : b'\xff\xff'
Standard encoding: b'//8='
URL-safe encoding: b'__8='
```

12.4.4. Другие кодировки

Кроме кодировки Base64 данный модуль предоставляет функции, позволяющие работать с кодировками Base85, Base32 и Base16 (шестнадцатеричная).

Листинг 12.29. base64_base32.py

```
import base64

original_data = b'This is the data, in the clear.'
print('Original:', original_data)

encoded_data = base64.b32encode(original_data)
print('Encoded :', encoded_data)

decoded_data = base64.b32decode(encoded_data)
print('Decoded :', decoded_data)
```

В кодировке Base32 используется алфавит, состоящий из 26 букв в верхнем регистре из набора ASCII-символов и цифр от 2 до 7.

```
$ python3 base64_base32.py

Original: b'This is the data, in the clear.'
Encoded : b'KRUGS4ZANFZSA5DIMUQGIYLUMEWCA2LOEB2GQZJAMNWGKYLSFY==
====='
Decoded : b'This is the data, in the clear.'
```

Функции Base16 работают с алфавитом, представленным шестнадцатеричными цифрами.

Листинг 12.30. base64_base16.py

```
import base64

original_data = b'This is the data, in the clear.'
print('Original:', original_data)

encoded_data = base64.b16encode(original_data)
print('Encoded :', encoded_data)

decoded_data = base64.b16decode(encoded_data)
print('Decoded :', decoded_data)
```

Каждый раз, когда количество кодируемых битов уменьшается, выходные данные в кодированном формате расширяются, чтобы занять больше места.

```
$ python3 base64_base16.py

Original: b'This is the data, in the clear.'
Encoded : b'546869732069732074686520646174612C20696E207468652063
6C6561722E'
Decoded : b'This is the data, in the clear.'
```

Функции Base85 используют расширенный алфавит, который обеспечивает более компактные выходные данные по сравнению с кодировкой Base64.

Листинг 12.31. base64_base85.py

```
import base64

original_data = b'This is the data, in the clear.'
print('Original : {} bytes {!r}'.format(
    len(original_data), original_data))

b64_data = base64.b64encode(original_data)
print('b64 Encoded : {} bytes {!r}'.format(
    len(b64_data), b64_data))

b85_data = base64.b85encode(original_data)
print('b85 Encoded : {} bytes {!r}'.format(
    len(b85_data), b85_data))

a85_data = base64.a85encode(original_data)
print('a85 Encoded : {} bytes {!r}'.format(
    len(a85_data), a85_data))
```

Некоторые кодировки Base85 и их вариации используются в системах управления версиями Mercurial и Git, а также в файловом формате PDF. Python включает две реализации: функцию `b85encode()`, которая реализует версию, используемую в Git Mercurial, и функцию `85encode()` — вариант Ascii85, используемый в PDF-файлах.

```
$ python3 base64_base85.py

Original    : 31 bytes b'This is the data, in the clear.'
b64 Encoded : 44 bytes b'VGhpcyBpcyB0aGUgZGF0YSwgaW4gdGhlIGNsZWF
yLg=='
b85 Encoded : 39 bytes b'RA^~)AZc?TbZBKDWMOn+EFfuaAarPDAY*K0VR9}'
a85 Encoded : 39 bytes b'<+oue+DGm>FD,5.A79Rg/0JYE+EV:.+Cf5!@<*t'
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `base64`[13].
- **RFC 3548**[14]. *The Base16, Base32, and Base64 Data Encodings*.

[13] https://docs.python.org/3.5/library/base64.html

[14] https://tools.ietf.org/html/rfc3548.html

- **RFC 1924**[15]. *A Compact Representation of IPv6 Addresses* (предложение кодировки Base85 для сетевых адресов IPv6).
- Википедия: *Ascii85*[16].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `base64` (раздел А.6.6).

12.5. `http.server`: базовые классы для реализации веб-серверов

Используя классы из модуля `socketserver` (раздел 11.5), модуль `http.server` позволяет получать базовые классы, предназначенные для создания HTTP-серверов. Можно использовать непосредственно класс `HTTPServer`, но расширение класса обработчика `BaseHTTPRequestHandler` обеспечивает обработку любого метода протокола (например, GET, POST).

12.5.1. HTTP GET

Чтобы добавить поддержку HTTP-метода в класс обработчика запросов, следует реализовать метод `do_XXX()`, заменив *XXX* названием соответствующего метода HTTP (например, `do_GET()`, `do_POST()`). В целях обеспечения единообразия методы обработчика запросов не имеют аргументов. Вместо этого все параметры запроса анализируются классом `BaseHTTPRequestHandler` и сохраняются в атрибутах экземпляра запроса.

В приведенном ниже примере показано, как вернуть ответ клиенту и использовать некоторые атрибуты, которые могут быть полезными при создании ответа.

Листинг 12.32. `http_server_GET.py`

```
from http.server import BaseHTTPRequestHandler
from urllib import parse


class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        parsed_path = parse.urlparse(self.path)
        message_parts = [
            'CLIENT VALUES:',
            'client_address={} ({})'.format(
                self.client_address,
                self.address_string()),
            'command={}'.format(self.command),
            'path={}'.format(self.path),
            'real path={}'.format(parsed_path.path),
            'query={}'.format(parsed_path.query),
            'request_version={}'.format(self.request_version),
            '',
```

[15] https://tools.ietf.org/html/rfc1924.html

[16] https://ru.wikipedia.org/wiki/Ascii85

```
            'SERVER VALUES:',
            ,server_version={}'.format(self.server_version),
            ,sys_version={}'.format(self.sys_version),
            ,protocol_version={}'.format(self.protocol_version),
            '',
            'HEADERS RECEIVED:',
        ]
        for name, value in sorted(self.headers.items()):
            message_parts.append(
                '{}={}'.format(name, value.rstrip())
            )
        message_parts.append('')
        message = '\r\n'.join(message_parts)
        self.send_response(200)
        self.send_header('Content-Type',
                         'text/plain; charset=utf-8')
        self.end_headers()
        self.wfile.write(message.encode('utf-8'))


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), GetHandler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

После того как текст сообщения сформирован, он записывается в выходной сокет с помощью файлового дескриптора `wfile`. Каждый ответ должен сопровождаться кодом ответа, устанавливаемым с помощью метода `send_response()`. Если этот код является кодом ошибки (например, `404`, `501`), то в заголовок включается сообщение об ошибке, используемое по умолчанию, или же сообщение может быть передано вместе с кодом ошибки. Чтобы запустить обработчик запросов на сервере, его следует передать конструктору `HTTPServer`, как это сделано в завершающей части сценария примера. После этого остается запустить сервер.

```
$ python3 http_server_GET.py

Starting server, use <Ctrl-C> to stop
```

Для доступа к серверу следует выполнить команду `curl` в другом окне.

```
$ curl -v -i http://127.0.0.1:8080/?foo=bar
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /?foo=bar HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.43.0
> Accept: */*
>
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 20:44:11 GMT
```

```
CLIENT VALUES:
client_address=('127.0.0.1', 52934) (127.0.0.1)
command=GET
path=/?foo=bar
real path=/
query=foo=bar
request_version=HTTP/1.1
SERVER VALUES:
server_version=BaseHTTP/0.6
sys_version=Python/3.5.2
protocol_version=HTTP/1.0
HEADERS RECEIVED:
Accept=*/*
Host=127.0.0.1:8080
User-Agent=curl/7.43.0
* Connection #0 to host 127.0.0.1 left intact
```

Примечание

Результирующий вывод может отличаться для разных версий `curl`. Если выполнение примера приводит к другим результатам, сверьтесь с отображаемым номером версии `curl`.

12.5.2. HTTP POST

Поддержка POST-запросов требует выполнения чуть большего объема работы, поскольку базовый класс не анализирует данные формы автоматически. Модуль `cgi` предоставляет класс `FieldStorage`, которому известно, как анализировать форму, если входные данные подготовлены корректно.

Листинг 12.33. `http_server_POST.py`

```
import cgi
from http.server import BaseHTTPRequestHandler
import io


class PostHandler(BaseHTTPRequestHandler):

    def do_POST(self):
        # Анализ выгруженных данных
        form = cgi.FieldStorage(
            fp=self.rfile,
            headers=self.headers,
            environ={
                'REQUEST_METHOD': 'POST',
                'CONTENT_TYPE': self.headers['Content-Type'],
            }
        )

        # Начать отвечать
        self.send_response(200)
        self.send_header('Content-Type',
                         'text/plain; charset=utf-8')
```

```
        self.end_headers()

        out = io.TextIOWrapper(
            self.wfile,
            encoding='utf-8',
            line_buffering=False,
            write_through=True,
        )

        out.write('Client: {}\n'.format(self.client_address))
        out.write('User-agent: {}\n'.format(
            self.headers['user-agent']))
        out.write('Path: {}\n'.format(self.path))
        out.write('Form data:\n')

        # Отправляемая обратно клиенту информация о данных,
        # выгруженных на сервер вместе с формой
        for field in form.keys():
            field_item = form[field]
            if field_item.filename:
                # Это поле содержит выгруженный файл
                file_data = field_item.file.read()
                file_len = len(file_data)
                del file_data
                out.write(
                    '\tUploaded {} as {!r} ({} bytes)\n'.format(
                        field, field_item.filename, file_len)
                )
            else:
                # Обычное значение формы
                out.write('\t{}={}\n'.format(
                    field, form[field].value))

        # Отключить экземпляр TextIOWrapper от базового буфера,
        # чтобы его удаление не привело к закрытию сокета, который
        # по-прежнему используется сервером
        out.detach()


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), PostHandler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

Запустите сервер в окне терминала.

```
$ python3 http_server_POST.py

Starting server, use <Ctrl-C> to stop
```

Аргументы команды `curl` могут включать форму данных, отправляемую на сервер с помощью опции `-F`. Последний аргумент, `-F datafile=@http_server_`

GET.py, задает отправку содержимого файла http_server_GET.py, чтобы продемонстрировать чтение данных из формы.

```
$ curl -v http://127.0.0.1:8080/ -F name=dhellmann -F foo=bar \
-F datafile=@http_server_GET.py

* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> POST / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Length: 1974
> Expect: 100-continue
> Content-Type: multipart/form-data;
boundary=------------------------a2b3c7485cf8def2
>
* Done waiting for 100-continue
HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 20:53:48 GMT

Client: ('127.0.0.1', 53121)
User-agent: curl/7.43.0
Path: /
Form data:
    name=dhellmann
    Uploaded datafile as 'http_server_GET.py' (1612 bytes)
    foo=bar
* Connection #0 to host 127.0.0.1 left intact
```

12.5.3. Порождение потоков и процессов

HTTPServer — это простой подкласс класса socketserver.TCPServer, который не использует многопоточность или ветвление процессов для обработки запросов. Чтобы добавить в него эти возможности, создайте новый класс, используя подходящий примесный класс из модуля socketserver (раздел 11.5).

Листинг 12.34. http_server_threads.py

```
from http.server import HTTPServer, BaseHTTPRequestHandler
from socketserver import ThreadingMixIn
import threading


class Handler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-Type',
                         'text/plain; charset=utf-8')
```

```
        self.end_headers()
        message = threading.currentThread().getName()
        self.wfile.write(message.encode('utf-8'))
        self.wfile.write(b'\n')


class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    """Обрабатывать запросы в отдельном потоке."""


if __name__ == '__main__':
    server = ThreadedHTTPServer(('localhost', 8080), Handler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

Запустите сервер тем же способом, что и в других примерах.

```
$ python3 http_server_threads.py

Starting server, use <Ctrl-C> to stop
```

Каждый раз, когда сервер получает запрос, он запускает новый поток или процесс для его обработки.

```
$ curl http://127.0.0.1:8080/
Thread-1
$ curl http://127.0.0.1:8080/
Thread-2
$ curl http://127.0.0.1:8080/
Thread-3
```

При замене аргумента `ThreadingMixIn` аргументом `ForkingMixIn` будут получены аналогичные результаты, но вместо потоков будут использоваться процессы.

12.5.4. Обработка ошибок

Для обработки ошибок предназначена функция `send_error()`, которая получает код ошибки и необязательное сообщение. Полный ответ (включая заголовки, код состояния и тело сообщения) генерируется автоматически.

Листинг 12.35. `http_server_errors.py`

```
from http.server import BaseHTTPRequestHandler


class ErrorHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_error(404)


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), ErrorHandler)
```

```
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

В данном случае всегда возвращается код ошибки 404.

```
$ python3 http_server_errors.py

Starting server, use <Ctrl-C> to stop
```

Клиенту отправляется заголовок с кодом ошибки и сообщение об ошибке в виде HTML-документа.

```
$ curl -i http://127.0.0.1:8080/

HTTP/1.0 404 Not Found
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 20:58:08 GMT
Connection: close
Content-Type: text/html;charset=utf-8
Content-Length: 447

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html;charset=utf-8">
        <title>Error response</title>
    </head>
    <body>
        <h1>Error response</h1>
        <p>Error code: 404</p>
        <p>Message: Not Found.</p>
        <p>Error code explanation: 404 - Nothing matches the
        given URI.</p>
    </body>
</html>
```

12.5.5. Настройка заголовков

Метод `send_header()` позволяет добавить данные заголовка в HTTP-ответ. Он получает два аргумента: имя и значение заголовка.

Листинг 12.36. `http_server_send_header.py`

```
from http.server import BaseHTTPRequestHandler
import time


class GetHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
```

```
        self.send_header(
            'Content-Type',
            'text/plain; charset=utf-8',
        )
        self.send_header(
            'Last-Modified',
            self.date_time_string(time.time())
        )
        self.end_headers()
        self.wfile.write('Response body\n'.encode('utf-8'))


if __name__ == '__main__':
    from http.server import HTTPServer
    server = HTTPServer(('localhost', 8080), GetHandler)
    print('Starting server, use <Ctrl-C> to stop')
    server.serve_forever()
```

В этом примере для отметки текущего времени, форматированной в соответствии с документом **RFC 7231**, устанавливается заголовок `Last-Modified`.

```
$ curl -i http://127.0.0.1:8080/

HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 21:00:54 GMT
Content-Type: text/plain; charset=utf-8
Last-Modified: Thu, 06 Oct 2016 21:00:54 GMT
Response body
```

Как и в других примерах, сервер выводит запрос на терминал.

```
$ python3 http_server_send_header.py

Starting server, use <Ctrl-C> to stop
127.0.0.1 - - [06/Oct/2016 17:00:54] "GET / HTTP/1.1" 200 -
```

12.5.6. Использование командной строки

Модуль `http.server` включает встроенный сервер для обслуживания файлов локальной файловой системы. Чтобы запустить его из командной строки, следует использовать опцию `-m` для интерпретатора Python.

```
$ python3 -m http.server 8080

Serving HTTP on 0.0.0.0 port 8080 ...
127.0.0.1 - - [06/Oct/2016 17:12:48] "HEAD /index.rst HTTP/1.1" 200 -
```

Корневым каталогом сервера служит рабочий каталог, в котором он запускается.

```
$ curl -I http://127.0.0.1:8080/index.rst

HTTP/1.0 200 OK
```

```
Server: SimpleHTTP/0.6 Python/3.5.2
Date: Thu, 06 Oct 2016 21:12:48 GMT
Content-type: application/octet-stream
Content-Length: 8285
Last-Modified: Thu, 06 Oct 2016 21:12:10 GMT
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `http.server`[17].
- `socketserver` (раздел 742). Модуль `socketserver` предоставляет базовый класс для управления серверным сокетом.
- **RFC 7231**[18]. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. Данный RFC-документ включает спецификацию формата заголовков и дат для протокола HTTP.

12.6. `http.cookies`: cookie-файлы HTTP

Модуль `http.cookies` реализует синтаксический анализатор для работы с cookie-файлами, который в основном отвечает требованиям документа RFC 2109[19]. Эта реализация не такая строгая, как того требует стандарт, поскольку приложение MSIE 3.0x не обеспечивает полную поддержку этого стандарта.

12.6.1. Создание и настройка cookie-файлов

Файлы *cookie* (или просто *куки*) используются в качестве средства управления состоянием в приложениях на основе браузера. Они представляют собой небольшие фрагменты данных, создаваемые сервером и сохраняемые на стороне клиента. Простейшим примером cookie-файла может служить пара "имя–значение".

Листинг 12.37. `http_cookies_setheaders.py`

```
from http import cookies

c = cookies.SimpleCookie()
c['mycookie'] = 'cookie_value'
print(c)
```

Вывод представляет собой действительный заголовок `Set-Cookie`, который может передаваться клиенту вместе с ответом HTTP.

```
$ python3 http_cookies_setheaders.py

Set-Cookie: mycookie=cookie_value
```

12.6.2. Атрибуты cookie-файлов

Cookie-файлы имеют ряд свойств, таких как срок жизни, путь и домен, которыми можно управлять. В действительности модуль `http.ccokies` предоставляет

[17] https://docs.python.org/3.5/library/http.server.html
[18] https://tools.ietf.org/html/rfc7231.html
[19] https://tools.ietf.org/html/rfc2109.html

объект Morsel, позволяющий сохранять куки и управлять всеми его атрибутами, определенными в RFC-документе.

Листинг 12.38. http_cookies_Morsel.py

```
from http import cookies
import datetime


def show_cookie(c):
    print(c)
    for key, morsel in c.items():
        print()
        print('key =', morsel.key)
        print('  value =', morsel.value)
        print('  coded_value =', morsel.coded_value)
        for name in morsel.keys():
            if morsel[name]:
                print('  {} = {}'.format(name, morsel[name]))


c = cookies.SimpleCookie()

# Куки, значение которого должно быть закодировано для
# того, чтобы поместить его в заголовок
c['encoded_value_cookie'] = '"cookie,value;"'
c['encoded_value_cookie']['comment'] = 'Has escaped punctuation'

# Куки, который применим только к части сайта
c['restricted_cookie'] = 'cookie_value'
c['restricted_cookie']['path'] = '/sub/path'
c['restricted_cookie']['domain'] = 'PyMOTW'
c['restricted_cookie']['secure'] = True

# Куки, срок действия которого истекает через 5 минут
c['with_max_age'] = 'expires in 5 minutes'
c['with_max_age']['max-age'] = 300  # seconds

# Куки, срок действия которого истекает к указанному времени
c['expires_at_time'] = 'cookie_value'
time_to_live = datetime.timedelta(hours=1)
expires = (datetime.datetime(2009, 2, 14, 18, 30, 14) +
           time_to_live)

# Формат даты: Wdy, DD-Mon-YY HH:MM:SS GMT
expires_at_time = expires.strftime('%a, %d %b %Y %H:%M:%S')
c['expires_at_time']['expires'] = expires_at_time

show_cookie(c)
```

Этот пример включает два метода настройки куки с ограниченным сроком действия. Один из них устанавливает максимальный срок действия куки в секундах, другой определяет точные дату и время истечения срока действия куки.

```
$ python3 http_cookies_Morsel.py

Set-Cookie: encoded_value_cookie="\"cookie\054value\073\"";
Comment=Has escaped punctuation
Set-Cookie: expires_at_time=cookie_value; expires=Sat, 14 Feb
2009 19:30:14
Set-Cookie: restricted_cookie=cookie_value; Domain=PyMOTW;
Path=/sub/path; Secure
Set-Cookie: with_max_age="expires in 5 minutes"; Max-Age=300

key = with_max_age
    value = expires in 5 minutes
    coded_value = "expires in 5 minutes"
    max-age = 300

key = expires_at_time
    value = cookie_value
    coded_value = cookie_value
    expires = Sat, 14 Feb 2009 19:30:14

key = restricted_cookie
    value = cookie_value
    coded_value = cookie_value
    domain = PyMOTW
    path = /sub/path
    secure = True

key = encoded_value_cookie
    value = "cookie,value;"
    coded_value = "\"cookie\054value\073\""
    comment = Has escaped punctuation
```

Оба объекта, `Cookie` и `Morsel`, работают подобно словарям. Объект `Morsel` содержит фиксированный набор ключей:

- `expires`
- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`

Ключами экземпляра `Cookie` являются имена отдельных сохраняемых cookie-файлов. Эта информация также доступна из атрибута `key` объекта `Morsel`.

12.6.3. Кодированные значения

Чтобы значения для заголовков куки могли быть корректно проанализированы, они должны кодироваться.

Листинг 12.39. http_cookies_coded_value.py

```
from http import cookies


c = cookies.SimpleCookie()
c['integer'] = 5
c['with_quotes'] = 'He said, "Hello, World!"'

for name in ['integer', 'with_quotes']:
    print(c[name].key)
    print('  {}'.format(c[name]))
    print('  value={!r}'.format(c[name].value))
    print('  coded_value={!r}'.format(c[name].coded_value))
    print()
```

В атрибуте `Morsel.value` всегда хранится декодированное значение куки, тогда как в атрибуте `Morsel.coded_value` всегда хранится его представление, используемое для передачи клиенту. Оба значения всегда являются строками. Значения, сохраняемые в куки, которые не являются строками, автоматически преобразуются в строки.

```
$ python3 http_cookies_coded_value.py

integer
  Set-Cookie: integer=5
  value='5'
  coded_value='5'

with_quotes
  Set-Cookie: with_quotes="He said\054 \"Hello\054 World!\""
  value='He said, "Hello, World!"'
  coded_value='"He said\\054 \\"Hello\\054 World!\\""'
```

12.6.4. Получение и анализ заголовков cookie-файлов

Как только клиент получает заголовки `Set-Cookie`, он отправляет их обратно серверу вместе с последующими запросами, используя заголовок `Cookie`. Входящая строка заголовка `Cookie` может содержать несколько значений куки, разделенных точками с запятой (`;`).

```
Cookie: integer=5; with_quotes="He said, \"Hello, World!\""
```

В зависимости от веб-сервера и фреймворка объекты `cookie` можно получить непосредственно из заголовка или из переменной среды `HTTP_COOKIE`.

Листинг 12.40. http_cookies_parse.py

```
from http import cookies


HTTP_COOKIE = '; '.join([
    r'integer=5',
```

```
    r'with_quotes="He said, \"Hello, World!\""',
])

print('From constructor:')
c = cookies.SimpleCookie(HTTP_COOKIE)
print(c)

print()
print('From load():')
c = cookies.SimpleCookie()
c.load(HTTP_COOKIE)
print(c)
```

Чтобы декодировать объекты `cookie`, следует передать строку без префикса заголовка конструктору объекта `SimpleCookie` при его инстанциализации или использовать метод `load()`.

```
$ python3 http_cookies_parse.py

From constructor:
Set-Cookie: integer=5
Set-Cookie: with_quotes="He said, \"Hello, World!\""

From load():
Set-Cookie: integer=5
Set-Cookie: with_quotes="He said, \"Hello, World!\""
```

12.6.5. Альтернативные выходные форматы

Помимо использования заголовка `Set-Cookie` серверы могут доставлять cookie-файлы клиенту с помощью сценариев JavaScript. Классы `SimpleCookie` и `Morsel` обеспечивают представление выходной информации в виде JavaScript-документа с помощью метода `js_output()`.

Листинг 12.41. `http_cookies_js_output.py`

```
from http import cookies
import textwrap


c = cookies.SimpleCookie()
c['mycookie'] = 'cookie_value'
c['another_cookie'] = 'second value'
js_text = c.js_output()
print(textwrap.dedent(js_text).lstrip())
```

Результат представляет собой полноценный дескриптор `script`, содержащий инструкции для установки cookie-файлов.

```
$ python3 http_cookies_js_output.py

<script type="text/javascript">
<!-- begin hiding
```

```
document.cookie = "another_cookie=\"second value\"";
// end hiding -->
</script>

<script type="text/javascript">
<!-- begin hiding
document.cookie = "mycookie=cookie_value";
// end hiding -->
</script>
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `http.cookies`[20].
- `http.cookiejar`. Модуль, обеспечивающий работу с cookie-файлами на стороне клиента.
- RFC 2109[21]. *HTTP State Management Mechanism*.

12.7. `webbrowser`: отображение веб-страниц

Модуль `webbrowser` включает функции, позволяющие открывать веб-документы в интерактивных браузерных приложениях. Он предоставляет реестр доступных браузеров, если в системе имеется несколько вариантов. Кроме того, браузеры можно контролировать с помощью переменной среды `BROWSER`.

12.7.1. Простой пример

Чтобы открыть страницу в браузере, следует использовать функцию `open()`.

Листинг 12.42. `webbrowser_open.py`

```
import webbrowser

webbrowser.open(
    'https://docs.python.org/3/library/webbrowser.html'
)
```

Документ, расположенный по указанному URL-адресу, открывается в окне браузера, и это окно выводится на передний план. В документации говорится о том, что по возможности используется открытое окно, но фактическое поведение зависит от настроек браузера. В случае использования браузера Firefox в MacOS X всегда создается новое окно.

12.7.2. Окна и вкладки

Если необходимо, чтобы всегда открывалось новое окно, следует использовать функцию `open_new()`.

[20] https://docs.python.org/3.5/library/http.cookies.html

[21] https://tools.ietf.org/html/rfc2109.html

Листинг 12.43. webbrowser_open_new.py

```
import webbrowser

webbrowser.open_new(
    'https://docs.python.org/3/library/webbrowser.html'
)
```

Если вместо этого желательно открывать новую вкладку, то необходимо использовать функцию open_new_tab().

12.7.3. Использование конкретного браузера

Если по каким-либо причинам приложению необходимо использовать конкретный браузер, то можно получить доступ к набору зарегистрированных браузерных контроллеров, используя функцию get(). Браузерный контроллер имеет методы open(), open_new() и open_new_tab(). В следующем примере принудительно используется браузер Lynx.

Листинг 12.44. webbrowser_get.py

```
import webbrowser

b = webbrowser.get('lynx')
b.open('https://docs.python.org/3/library/webbrowser.html')
```

Список доступных типов браузеров приведен в документации модуля.

12.7.4. Переменная BROWSER

Пользователи могут управлять модулем webbrowser извне приложения посредством указания в переменной среды BROWSER имен браузеров или команд, которые можно пытаться использовать. Значение этой переменной должно включать последовательность имен браузеров, разделенных символом os.pathsep. Если имя включает спецификатор %s, то оно интерпретируется как литеральная команда, в которой данный спецификатор заменяется URL-адресом. В противном случае имя передается функции get() для получения объекта контроллера из реестра.

Например, следующая команда открывает веб-страницу в браузере lynx, при условии, что она доступна, независимо от того, какие браузеры зарегистрированы.

```
$ BROWSER=lynx python3 webbrowser_open.py
```

Если ни одно из имен, указанных в переменной BROWSER, не работает, то восстанавливается поведение браузера, используемое по умолчанию.

12.7.5. Интерфейс командной строки

Все средства модуля webbrowser доступны как из программ на языке Python, так и из командной строки.

```
$ python3 -m webbrowser

Usage: .../lib/python3.5/webbrowser.py [-n | -t] url
```

```
-n: open new window
-t: open new tab
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `webbrowser`[22].
- `whatthewhat`[23]. Выполняет предоставленную программу на языке Python и в случае необходимости запускает поисковый механизм Google для поиска любых возможных сообщений об ошибках.

12.8. `uuid`: универсальные уникальные идентификаторы

Модуль `uuid` реализует стандарт идентификации UUID (Universally Unique Identifiers), определенный в документе **RFC 4122**[24]. В этом документе описана процедура создания уникальных идентификаторов ресурсов, не требующих использования централизованной системы регистрации. Значения UUID представляют собой 128-битовые номера, которые, как сказано в справочном руководстве, "могут гарантировать уникальность идентификаторов во времени и пространстве имен". UUID могут применяться для генерации идентификаторов документов, хостов и приложений клиентов, а также в других ситуациях, требующих использования уникальных значений. Упомянутый RFC-документ сфокусирован на создании пространства имен URN (Uniform Resource Name – унифицированное название ресурса) и охватывает следующие три основных алгоритма:

- использование MAC-адресов (стандарт IEEE 802) в качестве источника уникальности;
- использование псевдослучайных чисел;
- использование известных строк в сочетании с криптографическим хешированием.

В любом случае затравочное значение (seed) комбинируется с показаниями системного таймера и последовательностью таких показаний, чтобы сохранить уникальность идентификаторов даже при переводе таймера назад.

12.8.1. UUID 1: MAC-адрес (стандарт IEEE 802)

Значения UUID версии 1 вычисляются с использованием MAC-адреса хоста. Для получения значения MAC-адреса текущей системы модуль `uuid` использует функцию `getnode()`.

Листинг 12.45. `uuid_getnode.py`

```
import uuid

print(hex(uuid.getnode()))
```

[22] https://docs.python.org/3.5/library/webbrowser.html

[23] https://github.com/dhellmann/whatthewhat

[24] https://tools.ietf.org/html/rfc4122.html

Если в системе установлено несколько сетевых адаптеров и, следовательно, имеется несколько MAC-адресов, данная функция может вернуть любой из них.

```
$ python3 uuid_getnode.py

0xc82a14598875
```

Чтобы сгенерировать UUID для хоста, идентифицируемого по его MAC-адресу, следует использовать функцию `uuid1()`. Ее аргумент, определяющий идентификатор узла, — необязательный. Если он не задан, используется значение, возвращаемое функцией `getnode()`.

Листинг 12.46. `uuid_uuid1.py`

```
import uuid

u = uuid.uuid1()

print(u)
print(type(u))
print('bytes   :', repr(u.bytes))
print('hex     :', u.hex)
print('int     :', u.int)
print('urn     :', u.urn)
print('variant :', u.variant)
print('version :', u.version)
print('fields  :', u.fields)
print('  time_low            : ', u.time_low)
print('  time_mid            : ', u.time_mid)
print('  time_hi_version     : ', u.time_hi_version)
print('  clock_seq_hi_variant: ', u.clock_seq_hi_variant)
print('  clock_seq_low       : ', u.clock_seq_low)
print('  node                : ', u.node)
print('  time                : ', u.time)
print('  clock_seq           : ', u.clock_seq)
```

К компонентам возвращаемого объекта UUID можно обращаться посредством доступных только для чтения атрибутов экземпляра. Некоторые атрибуты, такие как `hex`, `int` и `urn`, являются различными представлениями значения UUID.

```
$ python3 uuid_uuid1.py

335ea282-cded-11e6-9ede-c82a14598875
<class 'uuid.UUID'>
bytes   : b'3^\xa2\x82\xcd\xed\x11\xe6\x9e\xde\xc8*\x14Y\x88u'
hex     : 335ea282cded11e69edec82a14598875
int     : 68281999803480928707202152670695098485
urn     : urn:uuid:335ea282-cded-11e6-9ede-c82a14598875
variant : specified in RFC 4122
version : 1
fields  : (861840002, 52717, 4582, 158, 222, 220083055593589)
  time_low            :  861840002
  time_mid            :  52717
```

```
  time_hi_version     :  4582
  clock_seq_hi_variant:  158
  clock_seq_low       :  222
  node                :  220083055593589
  time                :  137023257334162050
  clock_seq           :  7902
```

Из-за наличия временной составляющей каждый вызов функции `uuid1()` возвращает новое значение.

Листинг 12.47. `uuid_uuid1_repeat.py`

```
import uuid

for i in range(3):
    print(uuid.uuid1())
```

В этом выводе изменяется лишь временная составляющая (начальная часть строки).

```
$ python3 uuid_uuid1_repeat.py

3369ab5c-cded-11e6-8d5e-c82a14598875
336eea22-cded-11e6-9943-c82a14598875
336eeb5e-cded-11e6-9e22-c82a14598875
```

Поскольку MAC-адреса у всех компьютеров разные, результаты, полученные для других систем, будут совершенно другими. В следующем примере выполнение этого же кода на других хостах имитируется посредством явной передачи идентификаторов узлов в качестве аргументов.

Листинг 12.48. `uuid_uuid1_othermac.py`

```
import uuid

for node in [0x1ec200d9e0, 0x1e5274040e]:
    print(uuid.uuid1(node), hex(node))
```

Теперь возвращаемые результаты отличаются не только временной составляющей, но и идентификаторами узлов в конце UUID.

```
$ python3 uuid_uuid1_othermac.py

337969be-cded-11e6-97fa-001ec200d9e0 0x1ec200d9e0
3379b7e6-cded-11e6-9d72-001e5274040e 0x1e5274040e
```

12.8.2. UUID версий 3 и 5: значения на основе заданного имени

В некоторых контекстах желательно создавать значения UUID на основе имен, а не случайных или связанных со временем значений. В версиях 3 и 5 стандарта UUID специфические для пространства имен затравочные значения объединяются с именами посредством криптографического хеширования (MD5

и SHA1 соответственно). Для работы с именами DNS, адресами URL, объектами ISO OID и отличительными именами (distinguished names) протокола X.500 используются известные пространства имен, идентифицируемые предопределенными значениями UUID. Можно определить новые, специфические для приложения пространства имен, генерируя и сохраняя значения UUID.

Листинг 12.49. uuid_uuid3_uuid5.py

```
import uuid

hostnames = ['www.doughellmann.com', 'blog.doughellmann.com']

for name in hostnames:
    print(name)
    print('  MD5   :', uuid.uuid3(uuid.NAMESPACE_DNS, name))
    print('  SHA-1 :', uuid.uuid5(uuid.NAMESPACE_DNS, name))
    print()
```

Чтобы создать UUID на основе имени DNS, следует передать значение `uuid.NAMESPACE_DNS` в качестве аргумента `namespace` функции `uuid3()` или `uuid5()`.

```
$ python3 uuid_uuid3_uuid5.py

www.doughellmann.com
  MD5   : bcd02e22-68f0-3046-a512-327cca9def8f
  SHA-1 : e3329b12-30b7-57c4-8117-c2cd34a87ce9

blog.doughellmann.com
  MD5   : 9bdabfce-dfd6-37ab-8a3f-7f7293bcf111
  SHA-1 : fa829736-7ef8-5239-9906-b4775a5abacb
```

Значение UUID для заданного имени в пространстве имен всегда остается одним и тем же, независимо от того, когда или где оно было вычислено.

Листинг 12.50. uuid_uuid3_repeat.py

```
import uuid

namespace_types = sorted(
    n
    for n in dir(uuid)
    if n.startswith('NAMESPACE_')
)
name = 'www.doughellmann.com'

for namespace_type in namespace_types:
    print(namespace_type)
    namespace_uuid = getattr(uuid, namespace_type)
    print(' ', uuid.uuid3(namespace_uuid, name))
    print(' ', uuid.uuid3(namespace_uuid, name))
    print()
```

Для одного и того же имени значения в различных пространствах имен различаются.

```
$ python3 uuid_uuid3_repeat.py

NAMESPACE_DNS
  bcd02e22-68f0-3046-a512-327cca9def8f
  bcd02e22-68f0-3046-a512-327cca9def8f

NAMESPACE_OID
  e7043ac1-4382-3c45-8271-d5c083e41723
  e7043ac1-4382-3c45-8271-d5c083e41723

NAMESPACE_URL
  5d0fdaa9-eafd-365e-b4d7-652500dd1208
  5d0fdaa9-eafd-365e-b4d7-652500dd1208

NAMESPACE_X500
  4a54d6e7-ce68-37fb-b0ba-09acc87cabb7
  4a54d6e7-ce68-37fb-b0ba-09acc87cabb7
```

12.8.3. UUID 4: случайные значения

Иногда значения UUID на основе идентификатора хоста и пространства имен не являются "в достаточной степени различными". Например, в тех случаях, когда UUID предназначен для использования в качестве хеш-значения, желательно иметь случайную последовательность с более высокой степенью случайности, чтобы избежать возникновения конфликтов в хеш-таблице. Кроме того, если значения имеют меньшее количество общих цифр, их поиск в файлах журналов упрощается. Чтобы повысить степень случайности значений UUID, их следует генерировать с помощью функции `uuid4()` на основе случайных входных значений.

Листинг 12.51. `uuid_uuid4.py`

```
import uuid

for i in range(3):
    print(uuid.uuid4())
```

Источник фактора случайности зависит от того, какие библиотеки C доступны во время импортирования модуля `uuid`. Если может быть загружен модуль `libuuid` (или `uuid.dll`) и он содержит функцию, генерирующую случайные числа, то используется эта функция. В противном случае используется функция `os.urandom()` или модуль `random` (раздел 5.3).

```
$ python3 uuid_uuid4.py

7821863a-06f0-4109-9b88-59ba1ca5cc04
44846e16-4a59-4a21-8c8e-008f169c2dd5
1f3cef3c-e2bc-4877-96c8-eba43bf15bb6
```

12.8.4. Работа с объектами UUID

Помимо генерации новых значений UUID можно создавать объекты UUID путем анализа строк в стандартных форматах, что упрощает выполнение операций сравнения и сортировки.

Листинг 12.52. uuid_uuid_objects.py

```
import uuid


def show(msg, l):
    print(msg)
    for v in l:
        print(' ', v)
    print()


input_values = [
    'urn:uuid:f2f84497-b3bf-493a-bba9-7c68e6def80b',
    '{417a5ebb-01f7-4ed5-aeac-3d56cd5037b0}',
    '2115773a-5bf1-11dd-ab48-001ec200d9e0',
]

show('input_values', input_values)

uuids = [uuid.UUID(s) for s in input_values]
show('converted to uuids', uuids)

uuids.sort()
show('sorted', uuids)
```

Фигурные скобки и символы дефиса (-) удаляются из входных данных. Если строка имеет префикс urn: и/или uuid:, он тоже удаляется. Оставшийся текст должен представлять собой строку, состоящую из 16 шестнадцатеричных цифр, которые затем интерпретируются как значение UUID.

```
$ python3 uuid_uuid_objects.py

input_values
  urn:uuid:f2f84497-b3bf-493a-bba9-7c68e6def80b
  {417a5ebb-01f7-4ed5-aeac-3d56cd5037b0}
  2115773a-5bf1-11dd-ab48-001ec200d9e0

converted to uuids
  f2f84497-b3bf-493a-bba9-7c68e6def80b
  417a5ebb-01f7-4ed5-aeac-3d56cd5037b0
  2115773a-5bf1-11dd-ab48-001ec200d9e0

sorted
  2115773a-5bf1-11dd-ab48-001ec200d9e0
  417a5ebb-01f7-4ed5-aeac-3d56cd5037b0
  f2f84497-b3bf-493a-bba9-7c68e6def80b
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `uuid`[25].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `uuid` (раздел A.6.49).
- **RFC 4122**[26]. *A Universally Unique Identifier (UUID) URN Namespace*.

12.9. `json`: JavaScript Object Notation

Модуль `json` предоставляет API, аналогичный API модуля `pickle` (раздел 7.1) и обеспечивающий преобразование находящихся в памяти объектов Python в сериализованное представление с использованием нотации объектов JavaScript (JavaScript Object Notation – JSON). Преимуществом формата JSON по сравнению с модулем `pickle` является то, что он реализован во многих языках программирования (включая JavaScript). Чаще всего формат JSON используется для обмена данными между веб-сервером и клиентом в REST API, но также может быть полезен для этих целей и в других приложениях.

12.9.1. Кодирование и декодирование простых типов данных

Кодировщик распознает встроенные типы Python (т.е., `str`, `int`, `float`, `list`, `tuple` и `dict`) по умолчанию.

Листинг 12.53. `json_simple_types.py`

```
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA:', repr(data))

data_string = json.dumps(data)
print('JSON:', data_string)
```

Способ преобразования значений в процессе кодирования внешне напоминает тот, который используется при выводе данных с помощью функции `repr()` в Python.

```
$ python3 json_simple_types.py

DATA: [{'c': 3.0, 'a': 'A', 'b': (2, 4)}]
JSON: [{"c": 3.0, "a": "A", "b": [2, 4]}]
```

Типы объектов, восстановленных в результате декодирования, могут не совпадать с исходными.

[25] https://docs.python.org/3.5/library/uuid.html
[26] https://tools.ietf.org/html/rfc4122.html

Листинг 12.54. json_simple_types_decode.py

```
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA   :', data)

data_string = json.dumps(data)
print('ENCODED:', data_string)

decoded = json.loads(data_string)
print('DECODED:', decoded)

print('ORIGINAL:', type(data[0]['b']))
print('DECODED :', type(decoded[0]['b']))
```

В частности, кортежи преобразуются в списки.

```
$ python3 json_simple_types_decode.py

DATA   : [{'b': (2, 4), 'c': 3.0, 'a': 'A'}]
ENCODED: [{"b": [2, 4], "c": 3.0, "a": "A"}]
DECODED: [{'b': [2, 4], 'c': 3.0, 'a': 'A'}]
ORIGINAL: <class 'tuple'>
DECODED : <class 'list'>
```

12.9.2. Удобочитаемость и компактность вывода

Дополнительным преимуществом JSON по сравнению с модулем `pickle` (раздел 7.1) является удобочитаемость результатов. Функция `dumps()` получает несколько аргументов, позволяющих облегчить чтение выходных результатов. Например, флаг `sort_keys` информирует кодировщик о том, что ключи словаря должны выводиться в отсортированном, а не случайном порядке.

Листинг 12.55. json_sort_keys.py

```
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA:', repr(data))

unsorted = json.dumps(data)
print('JSON:', json.dumps(data))
print('SORT:', json.dumps(data, sort_keys=True))

first = json.dumps(data, sort_keys=True)
second = json.dumps(data, sort_keys=True)

print('UNSORTED MATCH:', unsorted == first)
print('SORTED MATCH  :', first == second)
```

Сортировка упрощает визуальный анализ результатов и обеспечивает возможность сравнения результатов в формате JSON в тестах.

```
$ python3 json_sort_keys.py

DATA: [{'c': 3.0, 'b': (2, 4), 'a': 'A'}]
JSON: [{"c": 3.0, "b": [2, 4], "a": "A"}]
SORT: [{"a": "A", "b": [2, 4], "c": 3.0}]
UNSORTED MATCH: False
SORTED MATCH : True
```

Для структур данных с большим количеством уровней вложения можно задать выделение уровней с помощью отступов для приведения результатов к более удобочитаемому виду.

Листинг 12.56. `json_indent.py`

```
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
print('DATA:', repr(data))

print('NORMAL:', json.dumps(data, sort_keys=True))
print('INDENT:', json.dumps(data, sort_keys=True, indent=2))
```

При неотрицательных значениях аргумента `indent` глубже расположенные уровни выделяются бо́льшим количеством пробелов, благодаря чему внешний вид результатов приближается к тому, который обеспечивает использование модуля `pprint` (раздел 2.10).

```
$ python3 json_indent.py

DATA: [{'a': 'A', 'c': 3.0, 'b': (2, 4)}]
NORMAL: [{"a": "A", "b": [2, 4], "c": 3.0}]
INDENT: [
  {
    "a": "A",
    "b": [
      2,
      4
    ],
    "c": 3.0
  }
]
```

Подобное детализированное представление увеличивает количество байтов, необходимых для передачи одного и того же объема данных, и поэтому его использование в производственной среде нецелесообразно. В действительности настройки, определяющие разделение данных в преобразованном выводе, позволяют обеспечить более компактный вывод, чем те, которые заданы по умолчанию.

Листинг 12.57. `json_compact_encoding.py`

```
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
```

```
print('DATA:', repr(data))

print('repr(data)             :', len(repr(data)))

plain_dump = json.dumps(data)
print('dumps(data)            :', len(plain_dump))

small_indent = json.dumps(data, indent=2)
print('dumps(data, indent=2)  :', len(small_indent))

with_separators = json.dumps(data, separators=(',', ':'))
print('dumps(data, separators):', len(with_separators))
```

Аргумент `separators` функции `dumps()` должен быть кортежем строк, определяющих разделители между элементами в списках и между ключами и значениями в словарях. По умолчанию этот аргумент принимает значение `(',',': ')`. Удаление пробела из кортежа позволяет получить более компактный вывод.

```
$ python3 json_compact_encoding.py

DATA: [{'c': 3.0, 'a': 'A', 'b': (2, 4)}]
repr(data)             : 35
dumps(data)            : 35
dumps(data, indent=2)  : 73
dumps(data, separators): 29
```

12.9.3. Кодирование словарей

Формат JSON ожидает, что в качестве ключей в словарях используются строки. При попытке преобразования в формат JSON словаря с ключами другого типа возбуждается исключение `TypeError`. Одним из способов, обеспечивающих пропуск нестроковых ключей, является использование аргумента `skipkeys`.

Листинг 12.58. `json_skipkeys.py`

```
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0, ('d',): 'D tuple'}]

print('First attempt')
try:
    print(json.dumps(data))
except TypeError as err:
    print('ERROR:', err)

print()
print('Second attempt')
print(json.dumps(data, skipkeys=True))
```

В этом случае ключи, не являющиеся строками, игнорируются и исключения не возбуждаются.

```
$ python3 json_skipkeys.py

First attempt
ERROR: keys must be a string

Second attempt
["c": 3.0}, "b": [2, 4],{"a": "A"]
```

12.9.4. Работа с пользовательскими типами

В приведенных до сих пор примерах использовались встроенные типы Python, поскольку в модуле `json` для них предусмотрена внутренняя поддержка. Однако нередко возникает потребность в преобразовании пользовательских классов в формат JSON. Для этого существуют два способа.

Предположим, необходимо преобразовать в формат JSON следующий класс.

Листинг 12.59. `json_myobj.py`

```
class MyObj:

    def __init__(self, s):
        self.s = s

    def __repr__(self):
        return '<MyObj({})>'.format(self.s)
```

Простой способ преобразования экземпляра `MyObj` заключается в определении функции, преобразующей неизвестный тип данных в известный. Эта функция не должна сама выполнять кодирование, она должна лишь преобразовывать один тип объекта в другой.

Листинг 12.60. `json_dump_default.py`

```
import json
import json_myobj

obj = json_myobj.MyObj('instance value goes here')

print('First attempt')
try:
    print(json.dumps(obj))
except TypeError as err:
    print('ERROR:', err)


def convert_to_builtin_type(obj):
    print('default(', repr(obj), ')')
    # Преобразование объекта в словарь его представления
    d = {
        '__class__': obj.__class__.__name__,
        '__module__': obj.__module__,
    }
```

```
    d.update(obj.__dict__)
    return d


print()
print('With default')
print(json.dumps(obj, default=convert_to_builtin_type))
```

В функции `convert_to_builtin_type()` экземпляры классов, не распознаваемых `json`, преобразуются в словари. Содержащейся в этих словарях информации достаточно для воссоздания объекта, если программа имеет доступ к модулям Python, необходимым для этого процесса.

```
$ python3 json_dump_default.py

First attempt
ERROR: <MyObj(instance value goes here)> is not JSON serializable

With default
default( <MyObj(instance value goes here)> )
{"s": "instance value goes here", "__module__": "json_myobj",
"__class__": "MyObj"}
```

Для декодирования результатов и воссоздания экземпляра `MyObj` необходимо передать функции `loads()` аргумент `object_hook`, определяющий функцию, которая будет применяться к результату преобразования каждого объекта JSON, чтобы можно было импортировать из модуля класс и использовать его для создания экземпляра. Функция `object_hook` вызывается для каждого словаря, декодированного из входящего потока данных, предоставляя возможность преобразования словарей в объекты другого типа. Эта функция должна возвращать соответствующий объект, который вызывающее приложение будет получать вместо словаря.

Листинг 12.61. `json_load_object_hook.py`

```
import json


def dict_to_object(d):
    if '__class__' in d:
        class_name = d.pop('__class__')
        module_name = d.pop('__module__')
        module = __import__(module_name)
        print('MODULE:', module.__name__)
        class_ = getattr(module, class_name)
        print('CLASS:', class_)
        args = {
            key: value
            for key, value in d.items()
        }
        print('INSTANCE ARGS:', args)
        inst = class_(**args)
    else:
```

```
        inst = d
    return inst


encoded_object = '''
    [{"s": "instance value goes here",
      "__module__": "json_myobj", "__class__": "MyObj"}]
    '''

myobj_instance = json.loads(
    encoded_object,
    object_hook=dict_to_object,
)
print(myobj_instance)
```

Поскольку модуль `json` преобразует строковые значения в объекты Unicode, эти объекты должны быть преобразованы в строки ASCII, прежде чем их можно будет использовать в качестве именованных аргументов в конструкторе класса.

```
$ python3 json_load_object_hook.py

MODULE: json_myobj
CLASS: <class 'json_myobj.MyObj'>
INSTANCE ARGS: {'s': 'instance value goes here'}
[<MyObj(instance value goes here)>]
```

Аналогичные функции доступны для всех встроенных типов: целых чисел (`parse_int`), чисел с плавающей точкой (`parse_float`) и констант (`parse_constant`).

12.9.5. Классы кодировщиков и декодировщиков

Кроме уже рассмотренных готовых функций модуль `json` предоставляет классы, предназначенные для выполнения операций кодирования и декодирования. Непосредственное использование классов открывает доступ к дополнительным программным интерфейсам, обеспечивающим возможность настройки поведения объектов.

Класс `JSONEncoder` использует итерируемый интерфейс для получения порций преобразованных данных, упрощая их запись в файлы или сетевые сокеты без представления в памяти всей структуры данных.

Листинг 12.62. `json_encoder_iterable.py`

```
import json

encoder = json.JSONEncoder()
data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]

for part in encoder.iterencode(data):
    print('PART:', part)
```

Выходные данные генерируются логическими блоками, а не на основе размера значений.

```
$ python3 json_encoder_iterable.py

PART: [
PART: {
PART: "c"
PART: :
PART: 3.0
PART: ,
PART: "b"
PART: :
PART: [2
PART: , 4
PART: ]
PART: ,
PART: "a"
PART: :
PART: "A"
PART: }
PART: ]
```

Метод `encode()` в основном эквивалентен вызову `''.join(encoder.iterencode())`, но предварительно выполняет дополнительные проверочные операции.

Для преобразования произвольных объектов следует переопределить метод `default()` реализацией, аналогичной той, которая использовалась в функции `convert_to_builtin_type()`.

Листинг 12.63. `json_encoder_default.py`

```
import json
import json_myobj


class MyEncoder(json.JSONEncoder):

    def default(self, obj):
        print('default(', repr(obj), ')')
        # Преобразование объекта в словарь его представления
        d = {
            '__class__': obj.__class__.__name__,
            '__module__': obj.__module__,
        }
        d.update(obj.__dict__)
        return d


obj = json_myobj.MyObj('internal data')
print(obj)
print(MyEncoder().encode(obj))
```

Результаты совпадают с теми, которые были получены с использованием предыдущей реализации.

```
$ python3 json_encoder_default.py

<MyObj(internal data)>
default( <MyObj(internal data)> )
{"s": "internal data", "__module__": "json_myobj", "__class__":
"MyObj"}
```

Декодирование текста и последующее преобразование словаря в объект требуют выполнения чуть большего объема работы по сравнению с предыдущей реализацией.

Листинг 12.64. `json_decoder_object_hook.py`

```
import json


class MyDecoder(json.JSONDecoder):

    def __init__(self):
        json.JSONDecoder.__init__(
            self,
            object_hook=self.dict_to_object,
        )

    def dict_to_object(self, d):
        if '__class__' in d:
            class_name = d.pop('__class__')
            module_name = d.pop('__module__')
            module = __import__(module_name)
            print('MODULE:', module.__name__)
            class_ = getattr(module, class_name)
            print('CLASS:', class_)
            args = {
                key: value
                for key, value in d.items()
            }
            print('INSTANCE ARGS:', args)
            inst = class_(**args)
        else:
            inst = d
        return inst


encoded_object = '''
[{"s": "instance value goes here",
  "__module__": "json_myobj", "__class__": "MyObj"}]
'''

myobj_instance = MyDecoder().decode(encoded_object)
print(myobj_instance)
```

Вывод остается тем же, что и в предыдущем примере.

```
$ python3 json_decoder_object_hook.py

MODULE: json_myobj
CLASS: <class 'json_myobj.MyObj'>
INSTANCE ARGS: {'s': 'instance value goes here'}
[<MyObj(instance value goes here)>]
```

12.9.6. Работа с потоками и файлами

До сих пор предполагалось, что преобразованная структура данных могла целиком храниться в памяти. В случае крупных структур более предпочтительной может оказаться запись результатов преобразования непосредственно в файловый объект. Функции `load()` и `dump()` получают ссылки на подобные файловые объекты, используемые как для чтения, так и для записи данных.

Листинг 12.65. `json_dump_file.py`

```
import io
import json

data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]

f = io.StringIO()
json.dump(data, f)

print(f.getvalue())
```

Дескриптор сокета или обычного файла работал бы точно так же, как и буфер `StringIO` в этом примере.

```
$ python3 json_dump_file.py

[{"c": 3.0, "b": [2, 4], "a": "A"}]
```

Несмотря на то что функция `load()` не оптимизирована для чтения данных по частям, она позволяет инкапсулировать логику генерирования объектов из входного потока.

Листинг 12.66. `json_load_file.py`

```
import io
import json

f = io.StringIO('[{"a": "A", "c": 3.0, "b": [2, 4]}]')
print(json.load(f))
```

Как и функция `dump()`, функция `load()` получает в качестве аргумента любой файловый объект.

```
$ python3 json_load_file.py

[{'c': 3.0, 'b': [2, 4], 'a': 'A'}]
```

12.9.7. Смешанные потоки данных

Класс `JSONDecoder` включает метод `raw_decode()`, позволяющий декодировать структуру данных, за которой следуют другие данные, как в случае данных JSON с замыкающим текстом. Данная функция возвращает объект, созданный в результате декодирования входных данных, и индекс позиции, в которой заканчивается декодированный объект.

Листинг 12.67. `json_mixed_data.py`

```
import json

decoder = json.JSONDecoder()


def get_decoded_and_remainder(input_data):
    obj, end = decoder.raw_decode(input_data)
    remaining = input_data[end:]
    return (obj, end, remaining)


encoded_object = '[{"a": "A", "c": 3.0, "b": [2, 4]}]'
extra_text = 'This text is not JSON.'

print('JSON first:')
data = ' '.join([encoded_object, extra_text])
obj, end, remaining = get_decoded_and_remainder(data)

print('Object              :', obj)
print('End of parsed input :', end)
print('Remaining text      :', repr(remaining))

print()
print('JSON embedded:')
try:
    data = ' '.join([extra_text, encoded_object, extra_text])
    obj, end, remaining = get_decoded_and_remainder(data)
except ValueError as err:
    print('ERROR:', err)
```

К сожалению, этот подход работает лишь в тех случаях, когда объект встречается в начале входных данных.

```
$ python3 json_mixed_data.py

JSON first:
Object              : [{'c': 3.0, 'b': [2, 4], 'a': 'A'}]
End of parsed input : 35
Remaining text      : ' This text is not JSON.'
JSON embedded  :
ERROR: Expecting value: line 1 column 1 (char 0)
```

12.9.8. JSON и командная строка

Модуль `json.tool` реализует утилиту командной строки, позволяющую переформатировать данные JSON в более удобочитаемый формат.

```
[{"a": "A", "c": 3.0, "b": [2, 4]}]
```

Входной файл *example.json* содержит словарь, порядок следования ключей которого отличается от алфавитного. В первом примере представлены данные, переформатированные в том же порядке, тогда как во втором примере ключи отображения сортируются с помощью параметра `--sort-keys`, прежде чем выводиться на консоль.

```
$ python3 -m json.tool example.json

[
    {
        "a": "A",
        "c": 3.0,
        "b": [
            2,
            4
        ]
    }
]

$ python3 -m json.tool --sort-keys example.json
[
    {
        "a": "A",
        "b": [
            2,
            4
        ],
        "c": 3.0
    }
]
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `json`[27].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `json` (раздел A.6.23).
- *Introducing JSON*[28]. Домашняя страница сайта JSON, содержащая документацию JSON и ссылки на реализации в других языках программирования.
- `jsonpickle`[29]. Модуль, обеспечивающий сериализацию любого объекта Python в формат JSON.

[27] `https://docs.python.org/3.5/library/json.html`

[28] `http://json.org/`

[29] `https://jsonpickle.github.io`

12.10. xmlrpc.client: клиент XML-RPC

XML-RPC (Extensible Markup Language Remote Procedure Call — XML-вызов удаленных процедур) — это упрощенный протокол вызова удаленных процедур, построенный поверх HTTP и XML. Модуль xmlrpclib позволяет программам на языке Python взаимодействовать с сервером XML-RPC, написанным на любом языке программирования.

Во всех примерах, приведенных в этом разделе, используется сервер, который определен в файле xmlrpc_server.py, доступном в исходном дистрибутиве Python и представленном ниже для справки.

Листинг 12.68. xmlrpc_server.py

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.client import Binary
import datetime


class ExampleService:

    def ping(self):
        """Простая функция, срабатывающая при вызове
        для демонстрации подключения.
        """
        return True

    def now(self):
        """возвращает текущие дату и время сервера."""
        return datetime.datetime.now()

    def show_type(self, arg):
        """Иллюстрирует передачу типов в серверные методы.

        Получает один аргумент любого типа.

        Возвращает кортеж со строковым представлением значения,
        имя типа и само значение.

        """
        return (str(arg), str(type(arg)), arg)

    def raises_exception(self, msg):
        "Всегда возбуждает исключение RuntimeError с переданным
        сообщением"
        raise RuntimeError(msg)

    def send_back_binary(self, bin):
        """Получает один двоичный аргумент, который распаковывается
        и перепаковывается для возврата."""
        data = bin.data
        print('send_back_binary({!r})'.format(data))
        response = Binary(data)
        return response
```

```
if __name__ == '__main__':
    server = SimpleXMLRPCServer(('localhost', 9000),
                                logRequests=True,
                                allow_none=True)
    server.register_introspection_functions()
    server.register_multicall_functions()

    server.register_instance(ExampleService())

    try:
        print('Use Control-C to exit')
        server.serve_forever()
    except KeyboardInterrupt:
        print('Exiting')
```

12.10.1. Подключение к серверу

Простейшим способом подключения клиента к серверу является создание объекта `ServerProxy` с передачей его конструктору URI сервера в качестве аргумента. Так, в приведенном ниже примере демонстрационный сервер выполняется с использованием имени `localhost` и порта `9000`.

Листинг 12.69. `xmlrpc_ServerProxy.py`

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')
print('Ping:', server.ping())
```

В данном случае метод `ping()` службы не имеет аргументов и возвращает одиночное булево значение.

```
$ python3 xmlrpc_ServerProxy.py

Ping: True
```

Также доступны другие опции, поддерживающие альтернативные транспортные протоколы для взаимодействия с серверами. Готовая поддержка протоколов HTTP и HTTPS предоставляется прямо "из коробки" вместе со средствами базовой аутентификации. Чтобы реализовать новый коммуникационный канал, требуется всего лишь создать новый транспортный класс. Например, в качестве интересного упражнения можно предложить реализацию XML-RPC поверх SMTP.

Листинг 12.70. `xmlrpc_ServerProxy_verbose.py`

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   verbose=True)
print('Ping:', server.ping())
```

Опция `verbose` задает генерирование отладочной информации, которая может быть полезной для поиска причин ошибок, возникающих при взаимодействии с сервером.

```
$ python3 xmlrpc_ServerProxy_verbose.py

send: b'POST /RPC2 HTTP/1.1\r\nHost: localhost:9000\r\n
Accept-Encoding: gzip\r\nContent-Type: text/xml\r\n
User-Agent: Python-xmlrpc/3.5\r\nContent-Length: 98\r\n\r\n'
send: b"<?xml version='1.0'?>\n<methodCall>\n<methodName>
ping</methodName>\n<params>\n</params>\n</methodCall>\n"
reply: 'HTTP/1.0 200 OK\r\n'
header: Server header: Date header: Content-type header:
Content-length body: b"<?xml version='1.0'?>\n<methodResponse>\n
<params>\n<param>\n<value><boolean>1</boolean></value>\n</param>
\n</params>\n</methodResponse>\n"
Ping: True
```

Вместо используемой по умолчанию кодировки символов UTF-8 можно задать любую другую в случае необходимости.

Листинг 12.71. xmlrpc_ServerProxy_encoding.py

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   encoding='ISO-8859-1')
print('Ping:', server.ping())
```

Сервер автоматически определяет корректную кодировку.

```
$ python3 xmlrpc_ServerProxy_encoding.py

Ping: True
```

Опция `allow_none` позволяет управлять тем, будет ли значение `None` в Python автоматически транслироваться в нулевое значение или вызывать ошибку.

Листинг 12.72. xmlrpc_ServerProxy_allow_none.py

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   allow_none=False)
try:
    server.show_type(None)
except TypeError as err:
    print('ERROR:', err)

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   allow_none=True)
print('Allowed:', server.show_type(None))
```

Исключение может возбуждаться как локально, если клиент не разрешает использовать значение `None`, так и на сервере, если он не сконфигурирован таким образом, чтобы значение `None` считалось допустимым.

```
$ python3 xmlrpc_ServerProxy_allow_none.py

ERROR: cannot marshal None unless allow_none is enabled
Allowed: ['None', "<class 'NoneType'>", None]
```

12.10.2. Типы данных

Протокол XML-RPC распознает ограниченный набор распространенных типов данных. Эти аргументы могут передаваться функциям и возвращаться ими, а также комбинироваться для создания сложных структур данных.

Листинг 12.73. xmlrpc_types.py

```
import xmlrpc.client
import datetime

server = xmlrpc.client.ServerProxy('http://localhost:9000')

data = [
    ('boolean', True),
    ('integer', 1),
    ('float', 2.5),
    ('string', 'some text'),
    ('datetime', datetime.datetime.now()),
    ('array', ['a', 'list']),
    ('array', ('a', 'tuple')),
    ('structure', {'a': 'dictionary'}),
]

for t, v in data:
    as_string, type_name, value = server.show_type(v)
    print('{:<12}: {}'.format(t, as_string))
    print('{:12}  {}'.format('', type_name))
    print('{:12}  {}'.format('', value))
```

В этом примере отображаются простые типы.

```
$ python3 xmlrpc_types.py

boolean     : True
              <class 'bool'>
              True
integer     : 1
              <class 'int'>
              1
float       : 2.5
              <class 'float'>
              2.5
string      : some text
              <class 'str'>
              some text
datetime    : 20160618T19:31:47
              <class 'xmlrpc.client.DateTime'>
```

```
                20160618T19:31:47
array         : ['a', 'list']
                <class 'list'>
                ['a', 'list']
array         : ['a', 'tuple']
                <class 'list'>
                ['a', 'tuple']
structure     : {'a': 'dictionary'}
                <class 'dict'>
                {'a': 'dictionary'}
```

Допускается вложение поддерживаемых типов для создания значений любой степени сложности.

Листинг 12.74. xmlrpc_types_nested.py

```
import xmlrpc.client
import datetime
import pprint

server = xmlrpc.client.ServerProxy('http://localhost:9000')

data = {
    'boolean': True,
    'integer': 1,
    'floating-point number': 2.5,
    'string': 'some text',
    'datetime': datetime.datetime.now(),
    'array1': ['a', 'list'],
    'array2': ('a', 'tuple'),
    'structure': {'a': 'dictionary'},
}
arg = []
for i in range(3):
    d = {}
    d.update(data)
    d['integer'] = i
    arg.append(d)

print('Before:')
pprint.pprint(arg, width=40)

print('\nAfter:')
pprint.pprint(server.show_type(arg)[-1], width=40)
```

В этой программе список словарей, содержащих все поддерживаемые типы данных, передается серверу, который возвращает данные. Кортежи транслируются в списки, а экземпляры `datetime` преобразуются в объекты `DateTime`, но во всем остальном данные не изменились.

```
$ python3 xmlrpc_types_nested.py

Before:
[{'array1': ['a', 'list'],
```

```
  'array2': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2016, 6, 18, 19, 27, 30, 45333),
  'floating-point number': 2.5,
  'integer': 0,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array1': ['a', 'list'],
  'array2': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2016, 6, 18, 19, 27, 30, 45333),
  'floating-point number': 2.5,
  'integer': 1,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array1': ['a', 'list'],
  'array2': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2016, 6, 18, 19, 27, 30, 45333),
  'floating-point number': 2.5,
  'integer': 2,
  'string': 'some text',
  'structure': {'a': 'dictionary'}}]

After:
[{'array1': ['a', 'list'],
  'array2': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20160618T19:27:30' at 0x101ecfac8>,
  'floating-point number': 2.5,
  'integer': 0,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array1': ['a', 'list'],
  'array2': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20160618T19:27:30' at 0x101ecfcc0>,
  'floating-point number': 2.5,
  'integer': 1,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array1': ['a', 'list'],
  'array2': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20160618T19:27:30' at 0x101ecfe10>,
  'floating-point number': 2.5,
  'integer': 2,
  'string': 'some text',
  'structure': {'a': 'dictionary'}}]
```

Протокол XML-RPC поддерживает даты как собственный тип. Модуль `xmlrpclib` может использовать один из двух классов для представления значений дат либо в исходящем прокси-объекте, либо по факту получения их с сервера.

Листинг 12.75. xmlrpc_ServerProxy_use_datetime.py

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   use_datetime=True)
now = server.now()
print('With:', now, type(now), now.__class__.__name__)

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   use_datetime=False)
now = server.now()
print('Without:', now, type(now), now.__class__.__name__)
```

По умолчанию используется внутренняя версия DateTime, но опция use_datetime включает поддержку использования классов из модуля datetime (раздел 4.2).

```
$ python3 source/xmlrpc.client/xmlrpc_ServerProxy_use_datetime.py

With: 2016-06-18 19:18:31 <class 'datetime.datetime'> datetime
Without: 20160618T19:18:31 <class 'xmlrpc.client.DateTime'> DateTime
```

12.10.3. Передача объектов

Экземпляры классов Python обрабатываются как структуры данных и передаются как словарь, значениями которого являются атрибуты объекта.

Листинг 12.76. xmlrpc_types_object.py

```
import xmlrpc.client
import pprint


class MyObj:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return 'MyObj({!r}, {!r})'.format(self.a, self.b)


server = xmlrpc.client.ServerProxy('http://localhost:9000')

o = MyObj(1, 'b goes here')
print('o  :', o)
pprint.pprint(server.show_type(o))

o2 = MyObj(2, o)
print('\no2 :', o2)
pprint.pprint(server.show_type(o2))
```

Когда сервер посылает значение обратно клиенту, результат представляет собой словарь. Это является отражением того факта, что в значениях не кодируется никакая информация, которая сообщала бы серверу (или клиенту) о том, что значения должны инстанциализироваться как часть класса.

```
$ python3 xmlrpc_types_object.py

o : MyObj(1, 'b goes here')
["{'b': 'b goes here', 'a': 1}", "<class 'dict'>",
{'a': 1, 'b': 'b goes here'}]
o2 : MyObj(2, MyObj(1, 'b goes here'))
["{'b': {'b': 'b goes here', 'a': 1}, 'a': 2}",
 "<class 'dict'>",
 {'a': 2, 'b': {'a': 1, 'b': 'b goes here'}}]
```

12.10.4. Двоичные данные

Все значения, передаваемые серверу, автоматически кодируются с использованием Escape-последовательностей в необходимых случаях. Однако некоторые типы данных могут содержать символы, не являющиеся действительными символами XML. Например, двоичные данные изображений могут включать байтовые значения из диапазона 0–31 таблицы ASCII, который соответствует управляющим символам. Для передачи двоичных данных лучше всего использовать класс `Binary`, обеспечивающий кодирование данных в процессе их передачи.

Листинг 12.77. `xmlrpc_Binary.py`

```
import xmlrpc.client
import xml.parsers.expat

server = xmlrpc.client.ServerProxy('http://localhost:9000')

s = b'This is a string with control characters\x00'
print('Local string:', s)

data = xmlrpc.client.Binary(s)
response = server.send_back_binary(data)
print('As binary:', response.data)

try:
    print('As string:', server.show_type(s))
except xml.parsers.expat.ExpatError as err:
    print('\nERROR:', err)
```

Если функции `show_type()` передается строка, содержащая байт `NULL`, то в процесс обработки ответа XML-анализатором возбуждается исключение.

```
$ python3 xmlrpc_Binary.py

Local string: b'This is a string with control characters\x00'
As binary: b'This is a string with control characters\x00'

ERROR: not well-formed (invalid token): line 6, column 55
```

Объекты Binary могут также использоваться для передачи объектов средствами модуля pickle (раздел 7.1). В подобных ситуациях не следует забывать о возможных угрозах безопасности, связанных с передачей по сети исполняемого кода (т.е. не следует использовать эту возможность, если нет уверенности в безопасности канала передачи данных).

```
import xmlrpc.client
import pickle
import pprint


class MyObj:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return 'MyObj({!r}, {!r})'.format(self.a, self.b)


server = xmlrpc.client.ServerProxy('http://localhost:9000')

o = MyObj(1, 'b goes here')
print('Local:', id(o))
print(o)

print('\nAs object:')
pprint.pprint(server.show_type(o))

p = pickle.dumps(o)
b = xmlrpc.client.Binary(p)
r = server.send_back_binary(b)

o2 = pickle.loads(r.data)
print('\nFrom pickle:', id(o2))
pprint.pprint(o2)
```

Атрибут data экземпляра Binary содержит версию объекта, сериализованную с помощью модуля pickle, которая должна быть десериализована перед ее использованием. Результатом этого является получение нового объекта (с другим значением ID).

```
$ python3 xmlrpc_Binary_pickle.py

Local: 4327262304
MyObj(1, 'b goes here')

As object:
["{'a': 1, 'b': 'b goes here'}", "<class 'dict'>",
{'a': 1, 'b': 'b goes here'}]

From pickle: 4327262472
MyObj(1, 'b goes here')
```

12.10.5. Обработка исключений

Если учесть тот факт, что сервер XML-RPC может быть написан на любом языке программирования, то становится понятным, что классы исключений нельзя передавать непосредственно. Вместо этого исключения, возбужденные на сервере, преобразуются в объекты `Fault` и возбуждаются как исключения локально на стороне клиента.

Листинг 12.78. `xmlrpc_exception.py`

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')
try:
    server.raises_exception('A message')
except Exception as err:
    print('Fault code:', err.faultCode)
    print('Message :', err.faultString)
```

Исходное сообщение об ошибке сохраняется в атрибуте `faultString`, тогда как в атрибуте `faultCode` сохраняется строка с номером ошибки XML-RPC.

```
$ python3 xmlrpc_exception.py

Fault code: 1
Message   : <class 'RuntimeError'>:A message
```

12.10.6. Комбинирование вызовов в одном сообщении

Multicall — это расширение протокола XML-RPC, позволяющее объединить и отправить в виде одного запроса сразу несколько вызовов, ответы на которые собираются и возвращаются клиенту.

Листинг 12.79. `xmlrpc_MultiCall.py`

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')

multicall = xmlrpc.client.MultiCall(server)
multicall.ping()
multicall.show_type(1)
multicall.show_type('string')

for i, r in enumerate(multicall()):
    print(i, r)
```

Чтобы использовать экземпляр `MultiCall`, следует вызвать те же методы, что и для экземпляра `ServerProxy`, а затем вызвать объект, не имеющий аргументов, для фактического выполнения удаленных функций. Возвращаемым значением является итератор, который возвращает результаты всех вызовов.

```
$ python3 xmlrpc_MultiCall.py

0 True
1 ['1', "<class 'int'>", 1]
2 ['string', "<class 'str'>", 'string']
```

Если один из вызовов приводит к ошибке, то при получении результата из итератора возбуждается исключение, и больше никакие другие результаты не становятся доступными.

Листинг 12.80. xmlrpc_MultiCall_exception.py

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')

multicall = xmlrpc.client.MultiCall(server)
multicall.ping()
multicall.show_type(1)
multicall.raises_exception('Next-to-last call stops execution')
multicall.show_type('string')

try:
    for i, r in enumerate(multicall()):
        print(i, r)
except xmlrpc.client.Fault as err:
    print('ERROR:', err)
```

Поскольку третий ответ, получаемый от функции `raises_exception()`, генерирует исключение, ответ функции `show_type()` недоступен.

```
$ python3 xmlrpc_MultiCall_exception.py

0 True
1 ['1', "<class 'int'>", 1]
ERROR: <Fault 1: "<class 'RuntimeError'>:Next-to-last call stops
execution">
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `xmlrpc.client`[30].
- `xmlrpc.server` (раздел 12.11). Реализация сервера XML-RPC.
- `http.server` (раздел 12.5). Реализация HTTP-сервера.
- XML-RPC HOWTO[31]. Описано использование протокола XML-RPC для реализации клиентов и серверов с помощью различных языков программирования.

[30] https://docs.python.org/3.5/library/xmlrpc.client.html
[31] www.tldp.org/HOWTO/XML-RPC-HOWTO/index.html

12.11. xmlrpc.server: сервер XML-RPC

Модуль xmlrpc.server содержит классы, предназначенные для создания кроссплатформенных серверов XML-RPC, не зависящих от языка программирования. Клиентские библиотеки существуют для многих языков помимо Python, что делает протокол XML-RPC удобным для реализации RPC-служб.

Примечание

Все примеры, приведенные в этом разделе, включают клиентский модуль, который взаимодействует с демонстрационным сервером. Для выполнения примеров необходимо использовать два окна: одно для сервера и одно для клиента.

12.11.1. Простой сервер

Используемый ниже простой сервер предоставляет всего одну функцию, которая получает имя каталога и возвращает его содержимое. Сначала необходимо создать сервер и передать ему адрес, по которому следует прослушивать запросы (в данном случае – порт 9000 локального хоста). Далее определяется функция, являющаяся частью службы, которую необходимо зарегистрировать, чтобы сервер знал, как ее вызывать. Последний шаг заключается в организации бесконечного цикла для получения запросов и отправки ответов.

Предупреждение

Данная реализация содержит очевидные уязвимости. Не используйте ее на общедоступном сервере в Интернете или в любой другой незащищенной среде.

Листинг 12.81. xmlrpc_function.py

```
from xmlrpc.server import SimpleXMLRPCServer
import logging
import os

# Включить протоколирование операций
logging.basicConfig(level=logging.INFO)

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)


# Предоставить функцию
def list_contents(dir_name):
    logging.info('list_contents(%s)', dir_name)
    return os.listdir(dir_name)


server.register_function(list_contents)

# Запустить сервер
try:
```

```
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Доступ к серверу по URL-адресу `http://localhost:9000` можно получить с помощью модуля `xmlrpc.client` (раздел 12.10). В приведенном ниже листинге клиентского кода продемонстрировано, как вызвать службу `list_contents()` из Python.

Листинг 12.82. `xmlrpc_function_client.py`

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print(proxy.list_contents('/tmp'))
```

Сначала экземпляр `ServerProxy` соединяется с сервером, используя его базовый URL-адрес, после чего методы вызываются непосредственно для прокси-объекта. Каждый метод, вызываемый для прокси, транслируется в запрос к серверу. Аргументы форматируются с использованием XML, а затем пересылаются на сервер в составе POST-сообщения. Сервер распаковывает XML-формат и определяет, какую функцию следует вызвать, на основании имени метода, активизируемого клиентом. Аргументы передаются функции, а возвращаемое значение транслируется обратно в XML и возвращается клиенту.

При запуске сервера выводится следующая информация.

```
$ python3 xmlrpc_function.py

Use Control-C to exit
```

Выполнение клиента во втором окне приводит к отображению содержимого каталога `/tmp`.

```
$ python3 xmlrpc_function_client.py

['com.apple.launchd.aoGXonn8nV', 'com.apple.launchd.ilryIaQugf',
'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']
```

После отправки ответа в окне сервера выводится информация о запросе.

```
$ python3 xmlrpc_function.py

Use Control-C to exit
INFO:root:list_contents(/tmp)
127.0.0.1 - - [18/Jun/2016 19:54:54] "POST /RPC2 HTTP/1.1" 200 -
```

Первая строка вывода происходит от вызова функции `logging.info()`, выполняющегося в теле функции `list_contents()`. Вторая строка — это вывод протоколируемой информации о запросе, поскольку для флага `logRequests` было установлено значение `True`.

12.11.2. Альтернативные имена API

Иногда имена функций, используемые в модуле или библиотеке, не совпадают с теми, которые должны использоваться во внешнем API. Необходимость в изменении имен может возникать по разным причинам: в силу специфических для платформы особенностей загруженной реализации, в силу того, что API службы создается динамически на основании конфигурационного файла, или потому, что реальные функции могли быть заменены заглушками в целях тестирования. Чтобы зарегистрировать функцию с альтернативным именем, следует передать это имя в качестве второго аргумента функции `register_function()`.

Листинг 12.83. xmlrpc_alternate_name.py

```
from xmlrpc.server import SimpleXMLRPCServer
import os

server = SimpleXMLRPCServer(('localhost', 9000))


def list_contents(dir_name):
    "Предоставляет функцию с альтернативным именем"
    return os.listdir(dir_name)
server.register_function(list_contents, 'dir')

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Клиент должен теперь использовать имя `dir()` вместо `list_contents()`.

Листинг 12.84. xmlrpc_alternate_name_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print('dir():', proxy.dir('/tmp'))
try:
    print('\nlist_contents():', proxy.list_contents('/tmp'))
except xmlrpc.client.Fault as err:
    print('\nERROR:', err)
```

Вызов функции `list_contents()` приводит к ошибке, потому что теперь ее имя отсутствует в списке зарегистрированных функций.

```
$ python3 xmlrpc_alternate_name_client.py

dir(): ['com.apple.launchd.aoGXonn8nV',
'com.apple.launchd.ilryIaQugf', 'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']

ERROR: <Fault 1: '<class \'Exception\'>:method "list_contents"
is not supported'>
```

12.11.3. Имена API с точками

Некоторые функции могут регистрироваться с именами, которые обычно являются недопустимыми идентификаторами в Python. Например, имена могут включать символы "точка" (.), разделяющие пространства имен в службе. Следующий пример расширяет службу "directory", добавляя в нее вызовы "create" и "remove". Все функции регистрируются с префиксом `dir.`, чтобы тот же сервер мог предоставлять другие услуги, используя другой префикс. Еще одним отличием является то, что некоторые из функций возвращают значение `None`, поэтому серверу необходимо сообщить, что такое значение должно транслироваться в нулевое значение.

Листинг 12.85. xmlrpc_dotted_name.py

```
from xmlrpc.server import SimpleXMLRPCServer
import os

server = SimpleXMLRPCServer(('localhost', 9000), allow_none=True)

server.register_function(os.listdir, 'dir.list')
server.register_function(os.mkdir, 'dir.create')
server.register_function(os.rmdir, 'dir.remove')

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Чтобы вызвать функцию службы, клиенту достаточно сослаться на ее имя с использованием точечной нотации.

Листинг 12.86. xmlrpc_dotted_name_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print('BEFORE       :', 'EXAMPLE' in proxy.dir.list('/tmp'))
print('CREATE       :', proxy.dir.create('/tmp/EXAMPLE'))
print('SHOULD EXIST :', 'EXAMPLE' in proxy.dir.list('/tmp'))
print('REMOVE       :', proxy.dir.remove('/tmp/EXAMPLE'))
print('AFTER        :', 'EXAMPLE' in proxy.dir.list('/tmp'))
```

В предположении, что в текущей системе файл `/tmp/EXAMPLE` отсутствует, данный клиентский сценарий выведет следующую информацию.

```
$ python3 xmlrpc_dotted_name_client.py

BEFORE       : False
CREATE       : None
SHOULD EXIST : True
REMOVE       : None
AFTER        : False
```

12.11.4. Произвольные имена API

Еще одной интересной возможностью является регистрация функций с именами, представляющими собой допустимые имена атрибутов объектов Python. В следующем примере служба регистрирует функцию с именем `multiply args`.

Листинг 12.87. xmlrpc_arbitrary_name.py

```
from xmlrpc.server import SimpleXMLRPCServer

server = SimpleXMLRPCServer(('localhost', 9000))


def my_function(a, b):
    return a * b


server.register_function(my_function, 'multiply args')

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Поскольку зарегистрированное имя содержит пробел, точечная нотация не может быть использована для непосредственного доступа к нему через прокси-объект. В таких ситуациях выходом является использование функции `getattr()`.

Листинг 12.88. xmlrpc_arbitrary_name_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print(getattr(proxy, 'multiply args')(5, 5))
```

Однако создавать службы с подобными именами не рекомендуется. Этот пример приведен не в качестве иллюстрации удачной идеи, а для того, чтобы показать, каким образом можно вызывать уже существующие службы с подобными произвольными именами.

```
$ python3 xmlrpc_arbitrary_name_client.py

25
```

12.11.5. Предоставление методов объектов

В предыдущих разделах уже обсуждались способы задания программных интерфейсов с использованием правил именования и пространств имен. Другим способом внедрения пространств имен в API является использование экземпляров классов и предоставление их методов. Первый пример можно воссоздать с использованием экземпляра, обладающего одним методом.

Листинг 12.89. xmlrpc_instance.py

```
from xmlrpc.server import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)


class DirectoryService:
    def list(self, dir_name):
        return os.listdir(dir_name)

server.register_instance(DirectoryService())

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Клиент может вызвать метод непосредственно.

Листинг 12.90. xmlrpc_instance_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print(proxy.list('/tmp'))
```

В выводе отображается содержимое каталога.

```
$ python3 xmlrpc_instance_client.py

['com.apple.launchd.aoGXonn8nV', 'com.apple.launchd.ilryIaQugf',
'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']
```

Однако при этом префикс `dir.` был утерян. Его можно восстановить, определив класс для создания иерархического дерева службы, который можно вызывать из клиентских программ.

Листинг 12.91. xmlrpc_instance_dotted_names.py

```
from xmlrpc.server import SimpleXMLRPCServer
import os
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)
```

```
class ServiceRoot:
    pass


class DirectoryService:

    def list(self, dir_name):
        return os.listdir(dir_name)


root = ServiceRoot()
root.dir = DirectoryService()

server.register_instance(root, allow_dotted_names=True)

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Поскольку экземпляр `ServiceRoot` зарегистрирован с включенной опцией `allow_dotted_names`, серверу разрешено выполнять иерархический поиск имен методов с помощью функции `getattr()` при поступлении запроса

Листинг 12.92. xmlrpc_instance_dotted_names_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print(proxy.dir.list('/tmp'))
```

Вывод метода `dir.list()` совпадает с тем, который был получен с использованием предыдущей реализации.

```
$ python3 xmlrpc_instance_dotted_names_client.py

['com.apple.launchd.aoGXonn8nV', 'com.apple.launchd.ilryIaQugf',
'example.db.db',
'KSOutOfProcessFetcher.501.ppfIhqX0vjaTSb8AJYobDV7Cu68=',
'pymotw_import_example.shelve.db']
```

12.11.6. Диспетчеризация вызовов

По умолчанию функция `register_instance()` находит все вызываемые атрибуты экземпляра, имена которых не начинаются с символа подчеркивания (_), и регистрирует их с их именами. Можно обеспечить более тщательный отбор методов, предоставляемых для обработки запросов, используя диспетчеризацию вызовов.

Листинг 12.93. xmlrpc_instance_with_prefix.py

```
from xmlrpc.server import SimpleXMLRPCServer
import os
```

```
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)


def expose(f):
    "Декоратор для установки флага функции."
    f.exposed = True
    return f


def is_exposed(f):
    "Проверяет, должна ли другая функция предоставляться открыто."
    return getattr(f, 'exposed', False)


class MyService:
    PREFIX = 'prefix'

    def _dispatch(self, method, params):
        # Удалить префикс из имени метода
        if not method.startswith(self.PREFIX + '.'):
            raise Exception(
                'method "{}" is not supported'.format(method)
            )

        method_name = method.partition('.')[2]
        func = getattr(self, method_name)
        if not is_exposed(func):
            raise Exception(
                'method "{}" is not supported'.format(method)
            )

        return func(*params)

    @expose
    def public(self):
        return 'This is public'

    def private(self):
        return 'This is private'


server.register_instance(MyService())

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

Метод public() класса MyService, в отличие от метода private(), помечен декоратором @expose как предоставляемый службе XML-RPC. Метод _dispatch() вызывается, когда клиент пытается получить доступ к функции, являющейся частью класса MyService. Сначала он пытается использовать префикс (в данном случае это prefix., но вместо него можно использовать любую другую строку). Затем он требует, чтобы функция имела атрибут exposed со значением True. Для удобства флаг exposed устанавливается для функции с помощью декоратора. Следующий пример включает несколько клиентских вызовов.

Листинг 12.94. xmlrpc_instance_with_prefix_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
print('public():', proxy.prefix.public())
try:
    print('private():', proxy.prefix.private())
except Exception as err:
    print('\nERROR:', err)
try:
    print('public() without prefix:', proxy.public())
except Exception as err:
    print('\nERROR:', err)
```

Ниже показано, как выглядит результирующий вывод вместе с ожидаемыми сообщениями об ошибке.

```
$ python3 xmlrpc_instance_with_prefix_client.py

public(): This is public

ERROR: <Fault 1: '<class \'Exception\'>:method "prefix.private" is
not supported'>

ERROR: <Fault 1: '<class \'Exception\'>:method "public" is not
supported'>
```

Существуют и другие способы переопределения механизма диспетчеризации, включая создание подклассов SimpleXMLRPCServer. Более подробную информацию об этом можно получить, прочитав строки документации модуля.

12.11.7. API интроспекции

Как и в случае многих других сетевых служб, можно опрашивать сервер XML-RPC с целью выяснения того, какие методы он поддерживает и как их использовать. Класс SimpleXMLRPCServer включает набор публичных методов, поддерживающих интроспекцию. По умолчанию они отключены, но их можно активизировать с помощью функции register_introspection_functions(). В службу можно добавить поддержку функций system.listMethods() и system.methodHelp(), определив для класса функцию _listMethods() или _methodHelp() соответственно.

Листинг 12.95. xmlrpc_introspection.py

```
from xmlrpc.server import (SimpleXMLRPCServer,
                           list_public_methods)
import os
import inspect

server = SimpleXMLRPCServer(
    ('localhost', 9000),
    logRequests=True,
)
server.register_introspection_functions()


class DirectoryService:

    def _listMethods(self):
        return list_public_methods(self)

    def _methodHelp(self, method):
        f = getattr(self, method)
        return inspect.getdoc(f)

    def list(self, dir_name):
        """list(dir_name) => [<filenames>]

        Returns a list containing the contents of
        the named directory.

        """
        return os.listdir(dir_name)


server.register_instance(DirectoryService())

try:
    print('Use Control-C to exit')
    server.serve_forever()
except KeyboardInterrupt:
    print('Exiting')
```

В данном случае функция `list_public_methods()` выполняет поиск в пространстве имен экземпляра и возвращает вызываемые атрибуты, имена которых не начинаются с символа подчеркивания (_). Чтобы применить требуемые правила, следует переопределить метод `_listMethods()`. Аналогичным образом в этом простом примере метод `_methodHelp()` возвращает строку документации функции, но его можно переписать для создания строки справки из другого источника.

Клиент посылает запрос серверу и выводит имена всех общедоступных методов.

Листинг 12.96. xmlrpc_introspection_client.py

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
for method_name in proxy.system.listMethods():
```

```
    print('=' * 60)
    print(method_name)
    print('-' * 60)
    print(proxy.system.methodHelp(method_name))
    print()
```

Результаты включают системные методы.

```
$ python3 xmlrpc_introspection_client.py
============================================================
list
------------------------------------------------------------
list(dir_name) => [<filenames>]

Returns a list containing the contents of
the named directory.
============================================================
system.listMethods
------------------------------------------------------------
system.listMethods() => ['add', 'subtract', 'multiple']

Returns a list of the methods supported by the server.
============================================================
system.methodHelp
------------------------------------------------------------
system.methodHelp('add') => "Adds two integers together"

Returns a string containing documentation for the specified method.
============================================================
system.methodSignature
------------------------------------------------------------
system.methodSignature('add') => [double, int, int]

Returns a list describing the signature of the method. In the
above example, the add method takes two integers as arguments
and returns a double result.

This server does NOT support system.methodSignature.
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `xmlrpc.server`[32].
- `xmlrpc.client` (раздел 12.10). Клиент XML-RPC.
- *XML-RPC HOWTO*[33]. Описание способов реализации клиентов и серверов, работающих по протоколу XML-RPC, с использованием различных языков программирования.

[32] https://docs.python.org/3.5/library/xmlrpc.server.html

[33] www.tldp.org/HOWTO/XML-RPC-HOWTO/index.html

Глава 13

Электронная почта

Электронная почта — один из старейших, но по-прежнему один из наиболее популярных видов цифровой связи. Стандартная библиотека Python включает модули, предназначенные для отправки, получения и хранения сообщений электронной почты.

Модуль `smtplib` (раздел 13.1) взаимодействует с сервером, обеспечивая доставку сообщений. Модуль `smtpd` (раздел 13.2) позволяет создавать пользовательские почтовые серверы и предоставляет классы, которые могут быть полезными при отладке механизмов электронной почты из других приложений.

Модуль `imaplib` (раздел 13.4) использует протокол IMAP для манипулирования сообщениями, хранящимися на сервере. Он предоставляет низкоуровневый API для клиентов IMAP и обеспечивает возможности запроса, извлечения, перемещения и удаления сообщений.

Модуль `mailbox` (раздел 13.3) позволяет создавать и изменять локальные архивы сообщений с использованием нескольких стандартных форматов, в том числе таких популярных, как mbox и Maildir, которые используются многими клиентскими программами электронной почты.

13.1. `smtplib`: клиент SMTP

Модуль `smtplib` включает класс `SMTP`, позволяющий связываться с почтовыми серверами для отправки электронных сообщений.

Примечание

Несмотря на то что в приведенных ниже примерах используются фиктивные адреса электронной почты, имена хостов и IP-адреса, они адекватно отражают последовательность выполнения команд и получения ответов.

13.1.1. Отправка сообщений

В типичных случаях протокол SMTP (Simple Mail Transfer Protocol — простой протокол передачи почты) используется для подключения к почтовому серверу и отправки сообщений. Имя хоста и номер порта почтового сервера можно передать либо непосредственно конструктору, либо явно вызванному методу `connect()`. После установления соединения с сервером вызывается метод `sendmail()` с параметрами конверта и тела сообщения. Текст сообщения должен быть оформлен в полном соответствии с требованиями документа **RFC 5322**[1], поскольку модуль `smtplib` вообще не изменяет содержимое или заголовки. Это означает, что поля `From` и `To` заголовка должны быть заполнены вызывающим кодом.

[1] `https://tools.ietf.org/html/rfc5322`

Листинг 13.1. smtplib_sendmail.py

```
import smtplib
import email.utils
from email.mime.text import MIMEText

# Создать сообщение
msg = MIMEText('This is the body of the message.')
msg['To'] = email.utils.formataddr(('Recipient',
                                    'recipient@example.com'))
msg['From'] = email.utils.formataddr(('Author',
                                      'author@example.com'))
msg['Subject'] = 'Simple test message'

server = smtplib.SMTP('localhost', 1025)
server.set_debuglevel(True)  # отображать информацию
                             # о подключении к серверу
try:
    server.sendmail('author@example.com',
                    ['recipient@example.com'],
                    msg.as_string())
finally:
    server.quit()
```

В этом примере, для того чтобы отобразить детали процесса взаимодействия клиента с сервером, включен режим отладки, иначе вывод вообще отсутствовал бы.

```
$ python3 smtplib_sendmail.py

send: 'ehlo 1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
0.0.0.0.0.0.ip6.arpa\r\n'
reply: b'250-1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
.0.0.0.0.0.0.ip6.arpa\r\n'
reply: b'250-SIZE 33554432\r\n'
reply: b'250 HELP\r\n'
reply: retcode (250); Msg: b'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa\nSIZE 33554432\nHELP'
send: 'mail FROM:<author@example.com> size=236\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'Content-Type: text/plain; charset="us-ascii"\r\nMIME-Ver
sion: 1.0\r\nContent-Transfer-Encoding: 7bit\r\nTo: Recipient <r
ecipient@example.com>\r\nFrom: Author <author@example.com>\r\nSu
bject: Simple test message\r\n\r\nThis is the body of the messag
e.\r\n.\r\n'
reply: b'250 OK\r\n'
```

```
reply: retcode (250); Msg: b'OK'
data: (250, b'OK')
send: 'quit\r\n'
reply: b'221 Bye\r\n'
reply: retcode (221); Msg: b'Bye'
```

Второй аргумент метода `sendmail()`, определяющий получателей, передается в виде списка. Сообщение будет поочередно доставлено каждому из получателей, адреса которых включены в этот список. Поскольку информация о конверте отделена от заголовков сообщения, возможна отправка скрытых копий некоторым адресатам посредством включения их адресов в аргумент метода, а не в заголовок сообщения.

13.1.2. Аутентификация и шифрование

Класс `SMTP` позволяет использовать средства аутентификации и шифрования протокола TLS (Transport Layer Security – протокол защиты транспортного уровня), если они поддерживаются сервером. Чтобы определить, поддерживает ли сервер работу с протоколом TLS, следует предоставить ему идентификатор клиента и запросить список доступных расширений, вызвав метод `ehlo()`. Результаты можно проверить, вызвав метод `has_extn()`. После запуска TLS следует повторно вызвать метод `ehlo()`, прежде чем будет выполнена процедура аутентификации пользователя. В настоящее время многие провайдеры почтовых услуг поддерживают *только* соединения на основе TLS. Связываясь с такими серверами, следует использовать экземпляр класса `SMTP_SSL` для установления шифрованного соединения.

Листинг 13.2. `smtplib_authenticated.py`

```
import smtplib
import email.utils
from email.mime.text import MIMEText
import getpass

# Вывести приглашение для ввода
# информации о соединении
to_email = input('Recipient: ')
servername = input('Mail server name: ')
serverport = input('Server port: ')
if serverport:
    serverport = int(serverport)
else:
    serverport = 25
use_tls = input('Use TLS? (yes/no): ').lower()
username = input('Mail username: ')
password = getpass.getpass("%s's password: " % username)

# Создать сообщение
msg = MIMEText('Test message from PyMOTW.')
msg.set_unixfrom('author')
msg['To'] = email.utils.formataddr(('Recipient', to_email))
msg['From'] = email.utils.formataddr(('Author',
```

```
                                        'author@example.com'))
msg['Subject'] = 'Test from PyMOTW'

if use_tls == 'yes':
    print('starting with a secure connection')
    server = smtplib.SMTP_SSL(servername, serverport)
else:
    print('starting with an insecure connection')
    server = smtplib.SMTP(servername, serverport)
try:
    server.set_debuglevel(True)

    # Идентифицировать себя, запрашивая информацию о
    # поддерживаемых средствах
    server.ehlo()

    # Использовать шифрование в этой сессии, если оно поддерживается
    if server.has_extn('STARTTLS'):
        print('(starting TLS)')
        server.starttls()
        server.ehlo()  # повторно идентифицировать себя
                       # посредством TLS-соединения
    else:
        print('(no STARTTLS)')

    if server.has_extn('AUTH'):
        print('(logging in)')
        server.login(username, password)
    else:
        print('(no AUTH)')

    server.sendmail('author@example.com',
                    [to_email],
                    msg.as_string())
finally:
    server.quit()
```

После активизации TLS расширение STARTTLS не появляется в ответе на команду EHLO.

```
$ python3 source/smtplib/smtplib_authenticated.py
Recipient: doug@pymotw.com
Mail server name: localhost
Server port: 1025
Use TLS? (yes/no): no
Mail username: test
test's password:
starting with an insecure connection
send: 'ehlo 1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
.0.0.0.0.ip6.arpa\r\n'
reply: b'250-1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
0.0.0.0.0.ip6.arpa\r\n'
reply: b'250-SIZE 33554432\r\n'
```

```
reply: b'250 HELP\r\n'
reply: retcode (250); Msg: b'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa\nSIZE 33554432\nHELP'
(no STARTTLS)
(no AUTH)
send: 'mail FROM:<author@example.com> size=220\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'rcpt TO:<doug@pymotw.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'Content-Type: text/plain; charset="us-ascii"\r\n
MIME-Version: 1.0\r\nContent-Transfer-Encoding: 7bit\r\nTo:
Recipient <doug@pymotw.com>\r\nFrom: Author <author@example.com>
\r\nSubject: Test from PyMOTW\r\n\r\nTest message from PyMOTW.
\r\n.\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
data: (250, b'OK')
send: 'quit\r\n'
reply: b'221 Bye\r\n'
reply: retcode (221); Msg: b'Bye'

$ python3 source/smtplib/smtplib_authenticated.py
Recipient: doug@pymotw.com
Mail server name: mail.isp.net
Server port: 465
Use TLS? (yes/no): yes
Mail username: doughellmann@isp.net
doughellmann@isp.net's password:
starting with a secure connection
send: 'ehlo 1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
.0.0.0.0.ip6.arpa\r\n'
reply: b'250-mail.isp.net\r\n'
reply: b'250-PIPELINING\r\n'
reply: b'250-SIZE 71000000\r\n'
reply: b'250-ENHANCEDSTATUSCODES\r\n'
reply: b'250-8BITMIME\r\n'
reply: b'250-AUTH PLAIN LOGIN\r\n'
reply: b'250 AUTH=PLAIN LOGIN\r\n'
reply: retcode (250); Msg: b'mail.isp.net\nPIPELINING\nSIZE
71000000\nENHANCEDSTATUSCODES\n8BITMIME\nAUTH PLAIN LOGIN\n
AUTH=PLAIN LOGIN'
(no STARTTLS)
(logging in)
send: 'AUTH PLAIN AGRvdWdoZWxsbWFubkBmYXN0bWFpbC5mbQBUTUZ3MDBmZmF
zdG1haWw=\r\n'
reply: b'235 2.0.0 OK\r\n'
reply: retcode (235); Msg: b'2.0.0 OK'
```

```
send: 'mail FROM:<author@example.com> size=220\r\n'
reply: b'250 2.1.0 Ok\r\n'
reply: retcode (250); Msg: b'2.1.0 Ok'
send: 'rcpt TO:<doug@pymotw.com>\r\n'
reply: b'250 2.1.5 Ok\r\n'
reply: retcode (250); Msg: b'2.1.5 Ok'
send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'Content-Type: text/plain; charset="us-ascii"\r\n
MIME-Version: 1.0\r\nContent-Transfer-Encoding: 7bit\r\nTo:
Recipient <doug@pymotw.com>\r\nFrom: Author <author@example.com>
\r\nSubject: Test from PyMOTW\r\n\r\nTest message from PyMOTW.
\r\n.\r\n'
reply: b'250 2.0.0 Ok: queued as A0EF7F2983\r\n'
reply: retcode (250); Msg: b'2.0.0 Ok: queued as A0EF7F2983'
data: (250, b'2.0.0 Ok: queued as A0EF7F2983')
send: 'quit\r\n'
reply: b'221 2.0.0 Bye\r\n'
reply: retcode (221); Msg: b'2.0.0 Bye'
```

13.1.3. Верификация адреса электронной почты

Протокол SMTP включает команду VRFY для проверки корректности имени адресата. Обычно ее отключают, чтобы затруднить спамерам определение истинных почтовых адресов. Но если она используется, то клиент может запросить у сервера информацию об адресе и получить код состояния, позволяющий убедиться в том, что адрес корректный. Также будет получено полное имя пользователя, если оно доступно.

Листинг 13.3. smtplib_verify.py

```
import smtplib

server = smtplib.SMTP('mail')
server.set_debuglevel(True)  # отображать информацию
                             # о соединении с сервером
try:
    dhellmann_result = server.verify('dhellmann')
    notthere_result = server.verify('notthere')
finally:
    server.quit()

print('dhellmann:', dhellmann_result)
print('notthere :', notthere_result)
```

Последние две строки вывода свидетельствуют о том, что адрес dhellmann — корректный, в то время как адрес notthere — некорректный.

```
$ python3 smtplib_verify.py

send: 'vrfy <dhellmann>\r\n'
```

```
reply: '250 2.1.5 Doug Hellmann <dhellmann@mail>\r\n'
reply: retcode (250); Msg: 2.1.5 Doug Hellmann <dhellmann@mail>
send: 'vrfy <notthere>\r\n'
reply: '550 5.1.1 <notthere>... User unknown\r\n'
reply: retcode (550); Msg: 5.1.1 <notthere>... User unknown
send: 'quit\r\n'
reply: '221 2.0.0 mail closing connection\r\n'
reply: retcode (221); Msg: 2.0.0 mail closing connection
dhellmann: (250, '2.1.5 Doug Hellmann <dhellmann@mail>')
notthere : (550, '5.1.1 <notthere>... User unknown')
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `smtplib`[2].
- RFC 821[3]. Спецификация SMTP — простого протокола передачи почты.
- RFC 1869[4]. *SMTP Service Extensions*. Расширение базового протокола.
- RFC 822[5]. *Standard for the Format of ARPA Internet Text Messages*. Первоначальная спецификация формата сообщений электронной почты.
- RFC 5322[6]. *Internet Message Format*. Обновление формата сообщений электронной почты.
- `email`. Модуль стандартной библиотеки, предназначенный для создания и анализа сообщений электронной почты.
- `smtpd` (раздел 13.2). Модуль, реализующий простой SMTP-сервер.

13.2. `smtpd`: примеры почтовых серверов

Модуль `smtpd` включает классы, предназначенные для создания SMTP-серверов. Эти классы реализуют серверную часть протокола, используемую модулем `smtplib` (раздел 13.1).

13.2.1. Базовый класс почтового сервера

В приведенных ниже примерах серверов в качестве базового класса используется класс `SMTPServer`. Он управляет взаимодействием с клиентом и получением входных данных, предоставляя удобную возможность организации пользовательской обработки сообщения, как только оно становится полностью доступным.

В качестве аргументов конструктор получает локальный адрес, по которому следует прослушивать соединения, и удаленные адреса, по которым должны быть доставлены сообщения. Производный класс может переопределить метод `process_message()`, который вызывается сразу же после получения всего сообщения и поддерживает следующие аргументы.

---

[2] `https://docs.python.org/3.5/library/smtplib.htm`
[3] `https://tools.ietf.org/html/rfc821.html`
[4] `https://tools.ietf.org/html/rfc1869.html`
[5] `https://tools.ietf.org/html/rfc822.html`
[6] `https://tools.ietf.org/html/rfc5322.html`

- peer. Адрес клиента в виде кортежа, содержащего IP-адрес и входящий порт.
- mailfrom. Информация, предоставляемая серверу клиентом с помощью команды MAIL FROM. Эта информация может не всегда совпадать с информацией, содержащейся в поле From заголовка.
- rcpttos. Список получателей из конверта сообщения. Опять-таки, этот список может не всегда совпадать с информацией, содержащейся в поле To заголовка, особенно если получателю направляется скрытая копия.
- data. Полное тело сообщения в соответствии с документом **RFC 5322**.

Предоставляемая по умолчанию реализация метода process_message() возбуждает исключение NotImplementedError. В следующем примере определяется подкласс, который переопределяет данный метод для вывода информации о получаемом сообщении.

Листинг 13.4. smtpd_custom.py

```
import smtpd
import asyncore


class CustomSMTPServer(smtpd.SMTPServer):

    def process_message(self, peer, mailfrom, rcpttos, data):
        print('Receiving message from:', peer)
        print('Message addressed from:', mailfrom)
        print('Message addressed to  :', rcpttos)
        print('Message length        :', len(data))


server = CustomSMTPServer(('127.0.0.1', 1025), None)

asyncore.loop()
```

Класс SMTPServer использует модуль asyncore, поэтому сервер запускается посредством вызова asyncore.loop().

Чтобы продемонстрировать работу сервера, требуется клиент. Для создания клиента, который будет посылать данные тестовому серверу, выполняющемуся локально и работающему через порт 1025, можно воспользоваться адаптированной версией одного из примеров, приведенных при обсуждении модуля smtplib (раздел 13.1).

Листинг 13.5. smtpd_senddata.py

```
import smtplib
import email.utils
from email.mime.text import MIMEText

# Создать сообщение
msg = MIMEText('This is the body of the message.')
msg['To'] = email.utils.formataddr(('Recipient',
                                    'recipient@example.com'))
```

```
msg['From'] = email.utils.formataddr(('Author',
                                      'author@example.com'))
msg['Subject'] = 'Simple test message'

server = smtplib.SMTP('127.0.0.1', 1025)
server.set_debuglevel(True)  # отображать информацию о
                             # взаимодействии с сервером
try:
    server.sendmail('author@example.com',
                    ['recipient@example.com'],
                    msg.as_string())
finally:
    server.quit()
```

Чтобы протестировать работу программ, следует выполнить сценарий `smtpd_custom.py` в одном окне терминала, а сценарий `smtpd_senddata.py` — в другом.

```
$ python3 smtpd_custom.py

Receiving message from: ('127.0.0.1', 58541)
Message addressed from: author@example.com
Message addressed to : ['recipient@example.com']
Message length : 229
```

Отладочный вывод отражает все этапы взаимодействия с сервером.

```
$ python3 smtpd_senddata.py

send: 'ehlo 1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
0.0.0.0.0.0.ip6.arpa\r\n'
reply: b'250-1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
.0.0.0.0.0.ip6.arpa\r\n'
reply: b'250-SIZE 33554432\r\n'
reply: b'250 HELP\r\n'
reply: retcode (250); Msg: b'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa\nSIZE 33554432\nHELP'
send: 'mail FROM:<author@example.com> size=236\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'Content-Type: text/plain; charset="us-ascii"\r\nMIME-Ver
sion: 1.0\r\nContent-Transfer-Encoding: 7bit\r\nTo: Recipient <r
ecipient@example.com>\r\nFrom: Author <author@example.com>\r\nSu
bject: Simple test message\r\n\r\nThis is the body of the messag
e.\r\n.\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
```

```
data: (250, b'OK')
send: 'quit\r\n'
reply: b'221 Bye\r\n'
reply: retcode (221); Msg: b'Bye'
```

Для остановки сервера следует нажать комбинацию клавиш <Ctrl-C>.

13.2.2. Отладочный сервер

В предыдущем примере было продемонстрировано использование аргументов метода `process_message()`, но модуль `smtpd` включает также сервер `DebuggingServer`, специально предназначенный для выполнения более полной отладки кода. Этот сервер выводит на консоль входящее сообщение целиком, после чего прекращает обработку (не пересылает сообщение реальному почтовому серверу).

Листинг 13.6. `smtpd_debug.py`

```
import smtpd
import asyncore

server = smtpd.DebuggingServer(('127.0.0.1', 1025), None)

asyncore.loop()
```

Совместное использование приведенной ранее клиентской программы `smtpd_senddata.py` с сервером `DebuggingServer` дает следующий вывод.

```
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
To: Recipient <recipient@example.com>
From: Author <author@example.com>
Subject: Simple test message
X-Peer: 127.0.0.1

This is the body of the message.
------------ END MESSAGE ------------
```

13.2.3. Прокси-сервер

Класс `PureProxy` реализует прокси-сервер. Входящие сообщения переадресовываются серверу, указанному в качестве аргумента конструктора.

Предупреждение

В разделе документации стандартной библиотеки, посвященном модулю `smtpd`, отмечается, что "запуская этот сервер, вы рискуете создать открытый шлюз, так что будьте осторожны".

Процедуры настройки прокси-сервера и отладочного сервера примерно одинаковы.

Листинг 13.7. smtpd_proxy.py

```
import smtpd
import asyncore

server = smtpd.PureProxy(('127.0.0.1', 1025), ('mail', 25))

asyncore.loop()
```

Эта программа не выводит никакой информации. Чтобы убедиться в том, что она правильно работает, следует обратиться к журналу сервера.

```
Aug 20 19:16:34 homer sendmail[6785]: m9JNGXJb006785:
from=<author@example.com>, size=248, class=0, nrcpts=1,
msgid=<200810192316.m9JNGXJb006785@homer.example.com>,
proto=ESMTP, daemon=MTA, relay=[192.168.1.17]
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `smtpd`[7].
- `smtplib` (раздел 13.1). Предоставляет интерфейс клиента.
- `email`. Выполняет синтаксический анализ сообщений электронной почты.
- `asyncore`. Базовый модуль, предназначенный для создания асинхронных серверов.
- **RFC 2822**[8]. *Internet Message Format*. Определяет формат сообщений электронной почты.
- **RFC 5322**[9]. Заменяет документ RFC 2822.

13.3. `mailbox`: манипулирование архивами электронной почты

Модуль `mailbox` определяет общий API для доступа к сообщениям электронной почты, сохраненным с использованием форматов локального диска, включая следующие:

- Maildir
- mbox
- MH
- Babyl
- MMDF

Существуют базовые классы `Mailbox` и `Message`, и каждый формат почтового ящика включает соответствующую пару подклассов для реализации деталей данного формата.

[7] https://docs.python.org/3.5/library/smtpd.html
[8] https://tools.ietf.org/html/rfc2822.html
[9] https://tools.ietf.org/html/rfc5322.html

13.3.1. mbox

Формат почтового ящика mbox – простейший для отображения в документации, поскольку он представляет собой простой текст. Каждый почтовый ящик сохраняется в отдельном файле, содержащем конкатенированные сообщения. Если встречается строка, начинающаяся с текста "From " (слово "From", за которым следует пробел), то она считается началом нового сообщения. Всякий раз, когда эти символы появляются в начале строки в теле сообщения, они экранируются префиксом >.

13.3.1.1. Создание почтового ящика mbox

Чтобы создать экземпляр класса `mbox`, следует передать конструктору имя файла. Если такого файла не существует, он будет создан при добавлении сообщений с помощью метода `add()`.

Листинг 13.8. `mailbox_mbox_create.py`

```
import mailbox
import email.utils

from_addr = email.utils.formataddr(('Author',
                                    'author@example.com'))
to_addr = email.utils.formataddr(('Recipient',
                                  'recipient@example.com'))

payload = '''This is the body.
From (will not be escaped).
There are 3 lines.
'''

mbox = mailbox.mbox('example.mbox')
mbox.lock()
try:
    msg = mailbox.mboxMessage()
    msg.set_unixfrom('author Sat Feb  7 01:05:34 2009')
    msg['From'] = from_addr
    msg['To'] = to_addr
    msg['Subject'] = 'Sample message 1'
    msg.set_payload(payload)
    mbox.add(msg)
    mbox.flush()

    msg = mailbox.mboxMessage()
    msg.set_unixfrom('author')
    msg['From'] = from_addr
    msg['To'] = to_addr
    msg['Subject'] = 'Sample message 2'
    msg.set_payload('This is the second body.\n')
    mbox.add(msg)
    mbox.flush()
```

```
finally:
    mbox.unlock()

print(open('example.mbox', 'r').read())
```

Результатом работы этого сценария является новый файл почтового ящика, содержащий два сообщения.

```
$ python3 mailbox_mbox_create.py

From MAILER-DAEMON Thu Dec 29 17:23:56 2016
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 1

This is the body.
>From (will not be escaped).
There are 3 lines.

From MAILER-DAEMON Thu Dec 29 17:23:56 2016
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 2

This is the second body.
```

13.3.1.2. Чтение сообщений из почтового ящика mbox

Чтобы прочитать сообщения из существующего почтового ящика, необходимо открыть его и обработать объект `mbox` как словарь. Ключами этого словаря являются значения, определяемые экземпляром почтового ящика, которые не обязательно должны что-либо выражать и служат всего лишь внутренними идентификаторами объектов сообщений.

Листинг 13.9. `mailbox_mbox_read.py`

```
import mailbox

mbox = mailbox.mbox('example.mbox')
for message in mbox:
    print(message['subject'])
```

Открытый почтовый ящик поддерживает протокол итератора. Однако в отличие от истинных объектов словаря итератор почтового ящика, предоставляемый по умолчанию, работает со *значениями*, а не с *ключами*.

```
$ python3 mailbox_mbox_read.py

Sample message 1
Sample message 2
```

13.3.1.3. Удаление сообщений из почтового ящика mbox

Чтобы удалить существующее сообщение из файла mbox, следует передать его ключ методу `remove()` или использовать команду `del`.

Листинг 13.10. `mailbox_mbox_remove.py`

```
import mailbox

mbox = mailbox.mbox('example.mbox')
mbox.lock()
try:
    to_remove = []
    for key, msg in mbox.iteritems():
        if '2' in msg['subject']:
            print('Removing:', key)
            to_remove.append(key)
    for key in to_remove:
        mbox.remove(key)
finally:
    mbox.flush()
    mbox.close()

print(open('example.mbox', 'r').read())
```

Методы `lock()` и `unlock()` позволяют избежать проблем, связанных с попытками одновременного доступа к файлу несколькими потоками, тогда как метод `flush()` принудительно записывает изменения на диск.

```
$ python3 mailbox_mbox_remove.py

Removing: 1
From MAILER-DAEMON Thu Dec 29 17:23:56 2016
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 1

This is the body.
>From (will not be escaped).
There are 3 lines.
```

13.3.2. Maildir

Формат Maildir был создан для устранения проблем, возникающих при попытках параллельного внесения изменений в файл mbox. Почтовые ящики этого формата не содержатся в единственном файле, а организуются в виде каталогов, в которых каждое сообщение хранится в собственном файле. Такая схема обеспечивает возможность вложения почтовых ящиков, поэтому API для почтовых ящиков формата Maildir расширен методами для работы с подпапками.

13.3.2.1. Создание почтового ящика Maildir

Единственное реальное различие между созданием почтовых ящиков в форматах Maildir и mbox заключается в том, что аргументом конструктора `Maildir` является имя каталога, а не файла. Если почтовый ящик не существует, то он создается при добавлении сообщения.

Листинг 13.11. `mailbox_maildir_create.py`

```
import mailbox
import email.utils
import os

from_addr = email.utils.formataddr(('Author',
                                    'author@example.com'))
to_addr = email.utils.formataddr(('Recipient',
                                  'recipient@example.com'))

payload = '''This is the body.
From (will not be escaped).
There are 3 lines.
'''

mbox = mailbox.Maildir('Example')
mbox.lock()
try:
    msg = mailbox.mboxMessage()
    msg.set_unixfrom('author Sat Feb  7 01:05:34 2009')
    msg['From'] = from_addr
    msg['To'] = to_addr
    msg['Subject'] = 'Sample message 1'
    msg.set_payload(payload)
    mbox.add(msg)
    mbox.flush()

    msg = mailbox.mboxMessage()
    msg.set_unixfrom('author Sat Feb  7 01:05:34 2009')
    msg['From'] = from_addr
    msg['To'] = to_addr
    msg['Subject'] = 'Sample message 2'
    msg.set_payload('This is the second body.\n')
    mbox.add(msg)
    mbox.flush()
finally:
    mbox.unlock()

for dirname, subdirs, files in os.walk('Example'):
    print(dirname)
    print('  Directories:', subdirs)
    for name in files:
        fullname = os.path.join(dirname, name)
        print('\n***', fullname)
        print(open(fullname).read())
        print('*' * 20)
```

При добавлении сообщений в почтовый ящик они попадают в новый подкаталог.

Предупреждение

Несмотря на безопасность записи сообщений из нескольких процессов в один и тот же почтовый ящик Maildir, метод `add()` не является потокобезопасным. Во избежание внесения изменений в почтовый ящик одновременно несколькими потоками одного и того же процесса следует использовать семафор или другой механизм блокировки.

```
$ python3 mailbox_maildir_create.py

Example
  Directories: ['cur', 'new', 'tmp']
Example/cur
  Directories: []
Example/new
  Directories: []

*** Example/new/1483032236.M378880P24253Q1.hubert.local
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 1

This is the body.
From (will not be escaped).
There are 3 lines.

********************

*** Example/new/1483032236.M381366P24253Q2.hubert.local

From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 2

This is the second body.

********************
Example/tmp
  Directories: []
```

После прочтения сообщений клиент может переместить их в подкаталог `cur`, используя метод `set_subdir()` класса `MaildirMessage`.

Листинг 13.12. `mailbox_maildir_set_subdir.py`

```
import mailbox
import os

print('Before:')
mbox = mailbox.Maildir('Example')
mbox.lock()
try:
```

```
    for message_id, message in mbox.iteritems():
        print('{:6} "{}"'.format(message.get_subdir(),
                                 message['subject']))
        message.set_subdir('cur')
        # Обновление сообщения
        mbox[message_id] = message
finally:
    mbox.flush()
    mbox.close()

print('\nAfter:')
mbox = mailbox.Maildir('Example')
for message in mbox:
    print('{:6} "{}"'.format(message.get_subdir(),
                             message['subject']))

print()
for dirname, subdirs, files in os.walk('Example'):
    print(dirname)
    print('  Directories:', subdirs)
    for name in files:
        fullname = os.path.join(dirname, name)
        print(fullname)
```

Несмотря на то что почтовый ящик Maildir включает каталог `tmp`, единственными допустимыми аргументами для метода `set_subdir()` являются `cur` и `new`.

```
$ python3 mailbox_maildir_set_subdir.py

Before:
new    "Sample message 2"
new    "Sample message 1"

After:
cur    "Sample message 2"
cur    "Sample message 1"

Example
  Directories: ['cur', 'new', 'tmp']
Example/cur
  Directories: []
Example/cur/1483032236.M378880P24253Q1.hubert.local
Example/cur/1483032236.M381366P24253Q2.hubert.local
Example/new
  Directories: []
Example/tmp
  Directories: []
```

13.3.2.2. Чтение сообщений из почтового ящика Maildir

Чтение сообщений из существующего почтового ящика Maildir осуществляется точно так же, как и из почтового ящика mbox.

Листинг 13.13. mailbox_maildir_read.py

```
import mailbox

mbox = mailbox.Maildir('Example')
for message in mbox:
    print(message['subject'])
```

Соблюдение определенного порядка чтения сообщений не гарантируется.

```
$ python3 mailbox_maildir_read.py

Sample message 2
Sample message 1
```

13.3.2.3. Удаление сообщений из почтового ящика Maildir

Чтобы удалить существующее сообщение из почтового ящика Maildir, следует передать его ключ методу `remove()` или использовать команду `del`.

Листинг 13.14. mailbox_maildir_remove.py

```
import mailbox
import os

mbox = mailbox.Maildir('Example')
mbox.lock()
try:
    to_remove = []
    for key, msg in mbox.iteritems():
        if '2' in msg['subject']:
            print('Removing:', key)
            to_remove.append(key)
    for key in to_remove:
        mbox.remove(key)
finally:
    mbox.flush()
    mbox.close()

for dirname, subdirs, files in os.walk('Example'):
    print(dirname)
    print('  Directories:', subdirs)
    for name in files:
        fullname = os.path.join(dirname, name)
        print('\n***', fullname)
        print(open(fullname).read())
        print('*' * 20)
```

Не существует способа определить ключ сообщения, поэтому необходимо использовать метод `items()` или `iteritems()`, чтобы извлечь одновременно ключ и объект сообщения из почтового ящика.

```
$ python3 mailbox_maildir_remove.py

Removing: 1483032236.M381366P24253Q2.hubert.local
Example
  Directories: ['cur', 'new', 'tmp']
Example/cur
  Directories: []

*** Example/cur/1483032236.M378880P24253Q1.hubert.local
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Sample message 1

This is the body.
From (will not be escaped).
There are 3 lines.

********************
Example/new
  Directories: []
Example/tmp
  Directories: []
```

13.3.2.4. Папки Maildir

Подкаталогами, или *папками*, почтового ящика Maildir можно управлять непосредственно с помощью методов класса `Maildir`. Вызывающий код может получать список подпапок заданного почтового ящика, а также извлекать, создавать и удалять их.

Листинг 13.15. `mailbox_maildir_folders.py`

```
import mailbox
import os


def show_maildir(name):
    os.system('find {} -print'.format(name))


mbox = mailbox.Maildir('Example')
print('Before:', mbox.list_folders())
show_maildir('Example')

print('\n{:#^30}\n'.format(''))

mbox.add_folder('subfolder')
print('subfolder created:', mbox.list_folders())
show_maildir('Example')

subfolder = mbox.get_folder('subfolder')
print('subfolder contents:', subfolder.list_folders())

print('\n{:#^30}\n'.format(''))
```

```
subfolder.add_folder('second_level')
print('second_level created:', subfolder.list_folders())
show_maildir('Example')

print('\n{:#^30}\n'.format(''))

subfolder.remove_folder('second_level')
print('second_level removed:', subfolder.list_folders())
show_maildir('Example')
```

Имя каталога для папки образуется путем присоединения префикса к имени папки через точку (.).

```
$ python3 mailbox_maildir_folders.py

Example
Example/cur
Example/cur/1483032236.M378880P24253Q1.hubert.local
Example/new
Example/tmp
Example
Example/.subfolder
Example/.subfolder/cur
Example/.subfolder/maildirfolder
Example/.subfolder/new
Example/.subfolder/tmp
Example/cur
Example/cur/1483032236.M378880P24253Q1.hubert.local
Example/new
Example/tmp
Example
Example/.subfolder
Example/.subfolder/.second_level
Example/.subfolder/.second_level/cur
Example/.subfolder/.second_level/maildirfolder
Example/.subfolder/.second_level/new
Example/.subfolder/.second_level/tmp
Example/.subfolder/cur
Example/.subfolder/maildirfolder
Example/.subfolder/new
Example/.subfolder/tmp
Example/cur
Example/cur/1483032236.M378880P24253Q1.hubert.local
Example/new
Example/tmp
Example
Example/.subfolder
Example/.subfolder/cur
Example/.subfolder/maildirfolder
Example/.subfolder/new
Example/.subfolder/tmp
Example/cur
Example/cur/1483032236.M378880P24253Q1.hubert.local
```

```
Example/new
Example/tmp
Before: []

##############################

subfolder created: ['subfolder']
subfolder contents: []

##############################

second_level created: ['second_level']

##############################

second_level removed: []
```

13.3.3. Флаги сообщений

Сообщения в почтовых ящиках имеют флаги, позволяющие отслеживать, было ли сообщение прочитано, отмечено пользователем как важное или помечено для последующего удаления. Флаги хранятся в виде последовательности специфических для формата буквенных кодов, и классы `Message` имеют методы, позволяющие извлекать и изменять значения флагов. В следующем примере отображено состояние флагов в почтовом ящике `Example` до и после добавления флага важности сообщения.

Листинг 13.16. `mailbox_maildir_add_flag.py`

```
import mailbox

print('Before:')
mbox = mailbox.Maildir('Example')
mbox.lock()
try:
    for message_id, message in mbox.iteritems():
        print('{:6} "{}"'.format(message.get_flags(),
                                 message['subject']))
        message.add_flag('F')
        # Обновление сообщения
        mbox[message_id] = message
finally:
    mbox.flush()
    mbox.close()

print('\nAfter:')
mbox = mailbox.Maildir('Example')
for message in mbox:
    print('{:6} "{}"'.format(message.get_flags(),
                             message['subject']))
```

По умолчанию сообщения не имеют флагов. Добавление флага изменяет сообщение в памяти, но не обновляет его на диске. Чтобы обновить сообщение на диске, следует сохранить объект сообщения в почтовом ящике, используя его существующий идентификатор.

```
$ python3 mailbox_maildir_add_flag.py

Before:
       "Sample message 1"


After:
F      "Sample message 1"
```

Добавление флагов с помощью метода `add_flag()` сохраняет существующие флаги. Можно переустановить существующий набор флагов с помощью метода `set_flags()`, передав ему новые значения.

Листинг 13.17. `mailbox_maildir_set_flags.py`

```
import mailbox

print('Before:')
mbox = mailbox.Maildir('Example')
mbox.lock()
try:
    for message_id, message in mbox.iteritems():
        print('{:6} "{}"'.format(message.get_flags(),
                                 message['subject']))
        message.set_flags('S')
        # Обновление сообщения
        mbox[message_id] = message
finally:
    mbox.flush()
    mbox.close()

print('\nAfter:')
mbox = mailbox.Maildir('Example')
for message in mbox:
    print('{:6} "{}"'.format(message.get_flags(),
                             message['subject']))
```

Добавленный в предыдущем примере флаг `F` теряется при замене флагов флагом `S` с помощью метода `set_flags()` в данном примере.

```
$ python3 mailbox_maildir_set_flags.py

Before:
F      "Sample message 1"

After:
S      "Sample message 1"
```

13.3.4. Другие форматы

Модуль `mailbox` поддерживает ряд других форматов, но ни один из них не пользуется такой популярностью, как форматы mbox и Maildir. Формат MH – это еще один многофайловый формат, используемый некоторыми обработчиками электронной почты. Форматы Babyl и MMDF – однофайловые и используют разделители сообщений, отличные от тех, которые применяются в формате mbox. Однофайловые форматы поддерживают тот же API, что и формат mbox, а формат MH включает методы, связанные с папками, которые содержатся в классе `Maildir`.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `mailbox`[10].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `mailbox` (раздел A.6.26).
- Раздел mbox справочного руководства почты qmail[11]. Содержит документацию по формату mbox.
- Раздел Maildir справочного руководства почты qmail[12]. Содержит документацию по формату Maildir.
- `email`. Модуль для работы с электронной почтой.
- `imaplib` (раздел 13.4). Модуль `imaplib` позволяет работать с сообщениями электронной почты, сохраненными на сервере IMAP.

13.4. `imaplib`: клиентская библиотека IMAP4

Модуль `imaplib` реализует клиентскую часть для взаимодействия с серверами по протоколу Internet Message Access Protocol (IMAP) v4. Протокол IMAP определяет набор команд, которые посылаются серверу, и ответов, которые доставляются обратно клиенту. Большинство команд доступно в виде методов объекта `IMAP4`, используемого для взаимодействия с сервером. В представленных ниже примерах обсуждается лишь часть протокола IMAP, и они ни в коей мере не иллюстрируют его полностью. Для получения более подробной информации следует обратиться к документу **RFC 3501**[13].

13.4.1. Разновидности класса `IMAP4`

Данный модуль предоставляет три класса, использующих три различных механизма для взаимодействия с серверами. Первый из них, `IMAP4`, применяет простые текстовые сокеты; класс `IMAP4_SSL` использует шифрование для взаимодействия через SSL-сокеты, а класс `IMAP4_stream` задействует стандартные потоки ввода-вывода внешней команды. Во всех примерах этого раздела используется класс `IMAP4_SSL`, но остальные классы имеют аналогичные программные интерфейсы.

[10] `https://docs.python.org/3.5/library/mailbox.html`

[11] `www.qmail.org/man/man5/mbox.html`

[12] `www.qmail.org/man/man5/maildir.html`

[13] `https://tools.ietf.org/html/rfc3501`

13.4.2. Подключение к серверу

Установление соединения с сервером осуществляется в два этапа: сначала создается собственно сокетное соединение, а затем выполняется аутентификация пользователя, имеющего учетную запись на сервере. В следующем примере информация о сервере и пользователе читается из конфигурационного файла.

Листинг 13.18. imaplib_connect.py

```
import imaplib
import configparser
import os


def open_connection(verbose=False):
    # Прочитать конфигурационный файл
    config = configparser.ConfigParser()
    config.read([os.path.expanduser('~/.pymotw')])

    # Подключиться к серверу
    hostname = config.get('server', 'hostname')
    if verbose:
        print('Connecting to', hostname)
    connection = imaplib.IMAP4_SSL(hostname)

    # Войти в учетную запись
    username = config.get('account', 'username')
    password = config.get('account', 'password')
    if verbose:
        print('Logging in as', username)
    connection.login(username, password)
    return connection


if __name__ == '__main__':
    with open_connection(verbose=True) as c:
        print(c)
```

В процессе выполнения сценария метод `open_connection()` читает конфигурационную информацию из файла, находящегося в домашнем каталоге пользователя, открывает соединение `IMAP4_SSL`, а затем аутентифицирует пользователя.

```
$ python3 imaplib_connect.py

Connecting to pymotw.hellfly.net
Logging in as example
<imaplib.IMAP4_SSL object at 0x10421e320>
```

Во избежание дублирования кода данный модуль повторно используется в других примерах, приведенных в этом разделе.

13.4.2.1. Ошибки аутентификации

Если соединение установлено, но аутентифицировать пользователя не удается, возбуждается исключение.

Листинг 13.19. imaplib_connect_fail.py

```
import imaplib
import configparser
import os

# Прочитать конфигурационный файл
config = configparser.ConfigParser()
config.read([os.path.expanduser('~/.pymotw')])

# Подключиться к серверу
hostname = config.get('server', 'hostname')
print('Connecting to', hostname)
connection = imaplib.IMAP4_SSL(hostname)

# Войти в учетную запись
username = config.get('account', 'username')
password = 'this_is_the_wrong_password'
print('Logging in as', username)
try:
    connection.login(username, password)
except Exception as err:
    print('ERROR:', err)
```

В этом примере намеренно используется неверный пароль, чтобы сработал механизм исключений.

```
$ python3 imaplib_connect_fail.py

Connecting to pymotw.hellfly.net
Logging in as example
ERROR: b'[AUTHENTICATIONFAILED] Authentication failed.'
```

13.4.3. Конфигурационные данные для примера

Используемая в примере учетная запись имеет следующую иерархию почтовых папок:

- INBOX
- Deleted Messages
- Archive
- Example
  - 2016

В данном случае имеется одно непрочитанное сообщение в папке `INBOX` и одно ранее прочитанное сообщение в папке `Example/2016`.

13.4.4. Получение списка почтовых ящиков

Для получения списка почтовых ящиков, доступных для учетной записи, следует использовать метод `list()`.

Листинг 13.20. `imaplib_list.py`

```
import imaplib
from pprint import pprint
from imaplib_connect import open_connection

with open_connection() as c:
    typ, data = c.list()
    print('Response code:', typ)
    print('Response:')
    pprint(data)
```

Метод `list()` возвращает кортеж, содержащий код ответа и данные, возвращенные сервером. В отсутствие ошибки возвращается код `OK`. Данными является последовательность строк, содержащих *флаги*, *разделитель иерархических уровней* и *имя почтового ящика* для каждого ящика.

```
$ python3 imaplib_list.py

Response code: OK
Response:
[b'(\\HasChildren) "." Example',
 b'(\\HasNoChildren) "." Example.2016',
 b'(\\HasNoChildren) "." Archive',
 b'(\\HasNoChildren) "." "Deleted Messages"',
 b'(\\HasNoChildren) "." INBOX']
```

Каждую строку ответа можно разбить на три части с помощью модуля `re` (раздел 1.3) или `csv` (раздел 7.6). (См. ссылку *IMAP Backup Script* в конце главы, если хотите увидеть пример использования модуля `csv`.)

Листинг 13.21. `imaplib_list_parse.py`

```
import imaplib
import re

from imaplib_connect import open_connection

list_response_pattern = re.compile(
    r'\((?P<flags>.*?)\) "(?P<delimiter>.*)" (?P<name>.*)'
)


def parse_list_response(line):
    match = list_response_pattern.match(line.decode('utf-8'))
    flags, delimiter, mailbox_name = match.groups()
    mailbox_name = mailbox_name.strip('"')
    return (flags, delimiter, mailbox_name)
```

```
with open_connection() as c:
    typ, data = c.list()
print('Response code:', typ)

for line in data:
    print('Server response:', line)
    flags, delimiter, mailbox_name = parse_list_response(line)
    print('Parsed response:', (flags, delimiter, mailbox_name))
```

Если имя почтового ящика содержит пробелы, сервер заключает его в кавычки, но для последующего использования имени в других вызовах сервера кавычки должны быть удалены.

```
$ python3 imaplib_list_parse.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Parsed response: ('\\HasChildren', '.', 'Example')
Server response: b'(\\HasNoChildren) "." Example.2016'
Parsed response: ('\\HasNoChildren', '.', 'Example.2016')
Server response: b'(\\HasNoChildren) "." Archive'
Parsed response: ('\\HasNoChildren', '.', 'Archive')
Server response: b'(\\HasNoChildren) "." "Deleted Messages"'
Parsed response: ('\\HasNoChildren', '.', 'Deleted Messages')
Server response: b'(\\HasNoChildren) "." INBOX'
Parsed response: ('\\HasNoChildren', '.', 'INBOX')
```

Метод `list()` получает аргументы, задающие имена почтовых ящиков в части иерархии. Например, чтобы вывести список всех подпапок папки `Example`, следует указать в качестве аргумента `directory` строку `'Example'`.

Листинг 13.22. `imaplib_list_subfolders.py`

```
import imaplib

from imaplib_connect import open_connection

with open_connection() as c:
    typ, data = c.list(directory='Example')

print('Response code:', typ)

for line in data:
    print('Server response:', line)
```

Возвращаемый список включает имена родительской папки и всех ее подпапок.

```
$ python3 imaplib_list_subfolders.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Server response: b'(\\HasNoChildren) "." Example.2016'
```

Также можно вывести только имена папок, соответствующие шаблону, задав аргумент pattern.

Листинг 13.23. imaplib_list_pattern.py

```
import imaplib

from imaplib_connect import open_connection

with open_connection() as c:
    typ, data = c.list(pattern='*Example*')

print('Response code:', typ)

for line in data:
    print('Server response:', line)
```

В данном случае в ответ включены папки Example и Example.2016.

```
$ python3 imaplib_list_pattern.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Server response: b'(\\HasNoChildren) "." Example.2016'
```

13.4.5. Состояние почтового ящика

Для получения агрегированной информации о содержимом почтового ящика следует вызвать метод status(). В табл. 13.1 приведены условия состояния, определенные стандартом.

Условия состояния должны форматироваться в виде строки, заключенной в скобки, в которой разделителем служит символ пробела, т.е. с использованием кодирования списков согласно спецификации IMAP4. Если имена включают пробелы или иные символы, которые могут сбивать с толку синтаксический анализатор, то имя почтового ящика должно заключаться в кавычки.

Таблица 13.1. Условия состояния почтового ящика IMAP4

Условие	Описание
MESSAGES	Количество сообщений в почтовом ящике
RECENT	Количество сообщений с установленным флагом \Recent
UIDNEXT	Значение следующего уникального идентификатора сообщений в почтовом ящике
UIDVALIDITY	Корректность значения уникального идентификатора почтового ящика
UNSEEN	Количество сообщений, для которых не установлен флаг \Seen

Листинг 13.24. imaplib_status.py

```
import imaplib
import re

from imaplib_connect import open_connection
```

```
from imaplib_list_parse import parse_list_response

with open_connection() as c:
    typ, data = c.list()
    for line in data:
        flags, delimiter, mailbox = parse_list_response(line)
        print('Mailbox:', mailbox)
        status = c.status(
            '"{}"'.format(mailbox),
            '(MESSAGES RECENT UIDNEXT UIDVALIDITY UNSEEN)',
        )
        print(status)
```

Метод `status()` возвращает обычный кортеж, содержащий код ответа и список данных, полученных от сервера. В данном случае список содержит одну строку, форматированную с именем почтового ящика, заключенным в кавычки, за которым следуют условия состояния и значения, заключенные в круглые скобки.

```
$ python3 imaplib_status.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Parsed response: ('\\HasChildren', '.', 'Example')
Server response: b'(\\HasNoChildren) "." Example.2016'
Parsed response: ('\\HasNoChildren', '.', 'Example.2016')
Server response: b'(\\HasNoChildren) "." Archive'
Parsed response: ('\\HasNoChildren', '.', 'Archive')
Server response: b'(\\HasNoChildren) "." "Deleted Messages"'
Parsed response: ('\\HasNoChildren', '.', 'Deleted Messages')
Server response: b'(\\HasNoChildren) "." INBOX'
Parsed response: ('\\HasNoChildren', '.', 'INBOX')
Mailbox: Example
('OK', [b'Example (MESSAGES 0 RECENT 0 UIDNEXT 2 UIDVALIDITY 145
7297771 UNSEEN 0)'])
Mailbox: Example.2016
('OK', [b'Example.2016 (MESSAGES 1 RECENT 0 UIDNEXT 3 UIDVALIDIT
Y 1457297772 UNSEEN 0)'])
Mailbox: Archive
('OK', [b'Archive (MESSAGES 0 RECENT 0 UIDNEXT 1 UIDVALIDITY 145
7297770 UNSEEN 0)'])
Mailbox: Deleted Messages
('OK', [b'"Deleted Messages" (MESSAGES 3 RECENT 0 UIDNEXT 4 UIDV
ALIDITY 1457297773 UNSEEN 0)'])
Mailbox: INBOX
('OK', [b'INBOX (MESSAGES 2 RECENT 0 UIDNEXT 6 UIDVALIDITY 14572
97769 UNSEEN 1)'])
```

13.4.6. Выбор почтового ящика

После аутентификации клиента базовой операцией является выбор почтового ящика с последующим запросом к серверу относительно имеющихся сообщений. Соединение запоминает состояние, поэтому после выбора почтового ящика все

команды воздействуют на сообщения, находящиеся в этом ящике до тех пор, пока не будет выбран другой.

Листинг 13.25. imaplib_select.py

```
import imaplib
import imaplib_connect

with imaplib_connect.open_connection() as c:
    typ, data = c.select('INBOX')
    print(typ, data)
    num_msgs = int(data[0])
    print('There are {} messages in INBOX'.format(num_msgs))
```

Данные ответа содержат общее количество сообщений в почтовом ящике.

```
$ python3 imaplib_select.py

OK [b'1']
There are 1 messages in INBOX
```

Если указано недействительное имя почтового ящика, то кодом ответа будет NO.

Листинг 13.26. imaplib_select_invalid.py

```
import imaplib
import imaplib_connect

with imaplib_connect.open_connection() as c:
    typ, data = c.select('Does-Not-Exist')
    print(typ, data)
```

В этом примере данные ответа содержат сообщение об ошибке, описывающее суть проблемы.

```
$ python3 imaplib_select_invalid.py

NO [b"Mailbox doesn't exist: Does-Not-Exist"]
```

13.4.7. Поиск сообщений

Выбрав почтовый ящик, можно использовать метод search() для получения идентификаторов сообщений, хранящихся в данном ящике.

Листинг 13.27. imaplib_search_all.py

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response

with imaplib_connect.open_connection() as c:
    typ, mbox_data = c.list()
    for line in mbox_data:
        flags, delimiter, mbox_name = parse_list_response(line)
```

```
        c.select('"{}"'.format(mbox_name), readonly=True)
        typ, msg_ids = c.search(None, 'ALL')
        print(mbox_name, typ, msg_ids)
```

Идентификаторы сообщений назначаются сервером и зависят от реализации. Протокол IMAP4 различает последовательные идентификаторы сообщений в заданный момент времени на протяжении транзакции и уникальные идентификаторы сообщений, но не все серверы реализуют обе эти разновидности идентификаторов.

```
$ python3 imaplib_search_all.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Parsed response: ('\\HasChildren', '.', 'Example')
Server response: b'(\\HasNoChildren) "." Example.2016'
Parsed response: ('\\HasNoChildren', '.', 'Example.2016')
Server response: b'(\\HasNoChildren) "." Archive'
Parsed response: ('\\HasNoChildren', '.', 'Archive')
Server response: b'(\\HasNoChildren) "." "Deleted Messages"'
Parsed response: ('\\HasNoChildren', '.', 'Deleted Messages')
Server response: b'(\\HasNoChildren) "." INBOX'
Parsed response: ('\\HasNoChildren', '.', 'INBOX')
Example OK [b'']
Example.2016 OK [b'1']
Archive OK [b'']
Deleted Messages OK [b'']
INBOX OK [b'1']
```

В данном случае в папках `INBOX` и `Example.2016` имеются разные сообщения, каждое из которых имеет идентификатор `1`. Другие почтовые ящики пусты.

13.4.8. Критерии поиска

Поиск сообщений может проводиться с использованием различных критериев, включая поиск по датам, флагам и другим заголовкам. Более подробную информацию по этому вопросу можно получить в разделе 6.4.4 документа **RFC 3501**[14].

Для поиска сообщений с текстом `'Example message 2'` в поле темы необходимо использовать такой критерий:

```
(SUBJECT "Example message 2")
```

В следующем примере ищутся все сообщения с темой “Example message 2” во всех почтовых ящиках.

Листинг 13.28. `imaplib_search_subject.py`

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response
```

[14] https://tools.ietf.org/html/rfc3501

```
with imaplib_connect.open_connection() as c:
    typ, mbox_data = c.list()
    for line in mbox_data:
        flags, delimiter, mbox_name = parse_list_response(line)
        c.select('"{}"'.format(mbox_name), readonly=True)
        typ, msg_ids = c.search(
            None,
            '(SUBJECT "Example message 2")',
        )
        print(mbox_name, typ, msg_ids)
```

Есть только одно такое сообщение для данной учетной записи, и оно находится в почтовом ящике INBOX.

```
$ python3 imaplib_search_subject.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Parsed response: ('\\HasChildren', '.', 'Example')
Server response: b'(\\HasNoChildren) "." Example.2016'
Parsed response: ('\\HasNoChildren', '.', 'Example.2016')
Server response: b'(\\HasNoChildren) "." Archive'
Parsed response: ('\\HasNoChildren', '.', 'Archive')
Server response: b'(\\HasNoChildren) "." "Deleted Messages"'
Parsed response: ('\\HasNoChildren', '.', 'Deleted Messages')
Server response: b'(\\HasNoChildren) "." INBOX'
Parsed response: ('\\HasNoChildren', '.', 'INBOX')
Example OK [b'']
Example.2016 OK [b'']
Archive OK [b'']
Deleted Messages OK [b'']
INBOX OK [b'1']
```

Допускается комбинирование критериев поиска.

Листинг 13.29. imaplib_search_from.py

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response

with imaplib_connect.open_connection() as c:
    typ, mbox_data = c.list()
    for line in mbox_data:
        flags, delimiter, mbox_name = parse_list_response(line)
        c.select('"{}"'.format(mbox_name), readonly=True)
        typ, msg_ids = c.search(
            None,
            '(FROM "Doug" SUBJECT "Example message 2")',
        )
        print(mbox_name, typ, msg_ids)
```

Критерии комбинируются с помощью логической операции И.

```
$ python3 imaplib_search_from.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Parsed response: ('\\HasChildren', '.', 'Example')
Server response: b'(\\HasNoChildren) "." Example.2016'
Parsed response: ('\\HasNoChildren', '.', 'Example.2016')
Server response: b'(\\HasNoChildren) "." Archive'
Parsed response: ('\\HasNoChildren', '.', 'Archive')
Server response: b'(\\HasNoChildren) "." "Deleted Messages"'
Parsed response: ('\\HasNoChildren', '.', 'Deleted Messages')
Server response: b'(\\HasNoChildren) "." INBOX'
Parsed response: ('\\HasNoChildren', '.', 'INBOX')
Example OK [b'']
Example.2016 OK [b'']
Archive OK [b'']
Deleted Messages OK [b'']
INBOX OK [b'1']
```

13.4.9. Извлечение сообщений

Идентификаторы, возвращаемые методом `search()`, используются для извлечения содержимого (или части содержимого), подлежащего дальнейшей обработке, с помощью метода `fetch()`. Этому методу передаются два аргумента, определяющие идентификатор извлекаемого сообщения и ту его часть, которая должна быть извлечена.

Аргумент `message_ids` задается в виде списка идентификаторов, разделенных запятыми (например, `"1"`, `"1,2"`), или диапазона идентификаторов (например, `1:2`). Аргумент `message_parts` задается в виде IMAP-списка имен сегментов сообщений. Как и в случае критериев поиска для метода `search()`, протокол IMAP определяет имена сегментов сообщений таким образом, чтобы клиенты могли эффективно извлекать лишь те части, которые фактически нужны. Например, чтобы извлечь заголовки сообщений, находящихся в почтовом ящике, следует использовать метод `fetch()` с аргументами `BODY.PEEK[HEADER]`.

Примечание

Для извлечения заголовков сообщений можно также использовать синтаксис `BODY[HEADERS]`, но при этом возникает побочный эффект, заключающийся в том, что сообщение неявно помечается как прочитанное, что во многих случаях нежелательно.

Листинг 13.30. `imaplib_fetch_raw.py`

```
import imaplib
import pprint
import imaplib_connect

imaplib.Debug = 4
with imaplib_connect.open_connection() as c:
    c.select('INBOX', readonly=True)
```

```
    typ, msg_data = c.fetch('1', '(BODY.PEEK[HEADER] FLAGS)')
    pprint.pprint(msg_data)
```

В данном примере значение, возвращаемое методом `fetch()`, подвергается частичному разбору на элементы, и потому работать с ним несколько труднее, чем с возвращаемым значением метода `list()`. Включение отладочного режима позволяет отобразить подробную информацию о взаимодействии клиента с сервером, которая помогает понять, почему это так.

```
$ python3 imaplib_fetch_raw.py

  19:40.68 imaplib version 2.58
  19:40.68 new IMAP4 connection, tag=b'IIEN'
  19:40.70 < b'* OK [CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN
-REFERRALS ID ENABLE IDLE AUTH=PLAIN] Dovecot (Ubuntu) ready.'
  19:40.70 > b'IIEN0 CAPABILITY'
  19:40.73 < b'* CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REF
ERRALS ID ENABLE IDLE AUTH=PLAIN'
  19:40.73 < b'IIEN0 OK Pre-login capabilities listed, post-logi
n capabilities have more.'
  19:40.73 CAPABILITIES: ('IMAP4REV1', 'LITERAL+', 'SASL-IR', 'L
OGIN-REFERRALS', 'ID', 'ENABLE', 'IDLE', 'AUTH=PLAIN')
  19:40.73 > b'IIEN1 LOGIN example "TMFw00fpymotw"'
  19:40.79 < b'* CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REF
ERRALS ID ENABLE IDLE SORT SORT=DISPLAY THREAD=REFERENCES THREAD
=REFS THREAD=ORDEREDSUBJECT MULTIAPPEND URL-PARTIAL CATENATE UNS
ELECT CHILDREN NAMESPACE UIDPLUS LIST-EXTENDED I18NLEVEL=1 CONDS
TORE QRESYNC ESEARCH ESORT SEARCHRES WITHIN CONTEXT=SEARCH LIST-
STATUS SPECIAL-USE BINARY MOVE'
  19:40.79 < b'IIEN1 OK Logged in'
  19:40.79 > b'IIEN2 EXAMINE INBOX'
  19:40.82 < b'* FLAGS (\\Answered \\Flagged \\Deleted \\Seen \\
Draft)'
  19:40.82 < b'* OK [PERMANENTFLAGS ()] Read-only mailbox.'
  19:40.82 < b'* 2 EXISTS'
  19:40.82 < b'* 0 RECENT'
  19:40.82 < b'* OK [UNSEEN 1] First unseen.'
  19:40.82 < b'* OK [UIDVALIDITY 1457297769] UIDs valid'
  19:40.82 < b'* OK [UIDNEXT 6] Predicted next UID'
  19:40.82 < b'* OK [HIGHESTMODSEQ 20] Highest'
  19:40.82 < b'IIEN2 OK [READ-ONLY] Examine completed (0.000 sec
s).'
  19:40.82 > b'IIEN3 FETCH 1 (BODY.PEEK[HEADER] FLAGS)'
  19:40.86 < b'* 1 FETCH (FLAGS () BODY[HEADER] {3108}'
  19:40.86 read literal size 3108
  19:40.86 < b')'
  19:40.89 < b'IIEN3 OK Fetch completed.'
  19:40.89 > b'IIEN4 LOGOUT'
  19:40.93 < b'* BYE Logging out'
  19:40.93 BYE response: b'Logging out'
[(b'1 (FLAGS () BODY[HEADER] {3108}',
  b'Return-Path: <doug@doughellmann.com>\r\nReceived: from compu
te4.internal ('
```

```
  b'compute4.nyi.internal [10.202.2.44])\r\n\t by sloti26t01 (Cy
rus 3.0.0-beta1'
  b'-git-fastmail-12410) with LMTPA;\r\n\t Sun, 06 Mar 2016 16:1
6:03 -0500\r'
  b'\nX-Sieve: CMU Sieve 2.4\r\nX-Spam-known-sender: yes, fadd1c
f2-dc3a-4984-a0'
  b'8b-02cef3cf1221="doug",\r\n ea349ad0-9299-47b5-b632-6ff1e39
4cc7d="both he'
  b'llfly"\r\nX-Spam-score: 0.0\r\nX-Spam-hits: ALL_TRUSTED -1,
BAYES_00 -1.'
  b'9, LANGUAGES unknown, BAYES_USED global,\r\n SA_VERSION 3.3
.2\r\nX-Spam'
  b"-source: IP='127.0.0.1', Host='unk', Country='unk', FromHead
er='com',\r\n "
  b" MailFrom='com'\r\nX-Spam-charsets: plain='us-ascii'\r\nX-Re
solved-to: d"
  b'oughellmann@fastmail.fm\r\nX-Delivered-to: doug@doughellmann
.com\r\nX-Ma'
  b'il-from: doug@doughellmann.com\r\nReceived: from mx5 ([10.20
2.2.204])\r'
  b'\n by compute4.internal (LMTPProxy); Sun, 06 Mar 2016 16:16
:03 -0500\r\nRe'
  b'ceived: from mx5.nyi.internal (localhost [127.0.0.1])\r\n\tb
y mx5.nyi.inter'
  b'nal (Postfix) with ESMTP id 47CBA280DB3\r\n\tfor <doug@dough
ellmann.com>; S'
  b'un, 6 Mar 2016 16:16:03 -0500 (EST)\r\nReceived: from mx5.n
yi.internal (l'
  b'ocalhost [127.0.0.1])\r\n by mx5.nyi.internal (Authentica
tion Milter) w'
  b'ith ESMTP\r\n id A717886846E.30BA4280D81;\r\n Sun, 6 M
ar 2016 16:1'
  b'6:03 -0500\r\nAuthentication-Results: mx5.nyi.internal;\r\n
   dkim=pass'
  b' (1024-bit rsa key) header.d=messagingengine.com header.i=@m
essagingengi'
  b'ne.com header.b=Jrsm+pCo;\r\n x-local-ip=pass\r\nReceived
: from mailo'
  b'ut.nyi.internal (gateway1.nyi.internal [10.202.2.221])\r\n\t
(using TLSv1.2 '
  b'with cipher ECDHE-RSA-AES256-GCM-SHA384 (256/256 bits))\r\n\
t(No client cer'
  b'tificate requested)\r\n\tby mx5.nyi.internal (Postfix) with
ESMTPS id 30BA4'
  b'280D81\r\n\tfor <doug@doughellmann.com>; Sun, 6 Mar 2016 16
:16:03 -0500 (E'
  b'ST)\r\nReceived: from compute2.internal (compute2.nyi.intern
al [10.202.2.4'
  b'2])\r\n\tby mailout.nyi.internal (Postfix) with ESMTP id 174
0420D0A\r\n\tf'
  b'or <doug@doughellmann.com>; Sun, 6 Mar 2016 16:16:03 -0500
(EST)\r\nRecei'
  b'ved: from frontend2 ([10.202.2.161])\r\n by compute2.intern
```

```
al (MEProxy); '
  b'Sun, 06 Mar 2016 16:16:03 -0500\r\nDKIM-Signature: v=1; a=rs
a-sha1; c=rela'
  b'xed/relaxed; d=\r\n\tmessagingengine.com; h=content-transfer
-encoding:conte'
  b'nt-type\r\n\t:date:from:message-id:mime-version:subject:to:x
-sasl-enc\r\n'
  b'\t:x-sasl-enc; s=smtpout; bh=P98NTsEo015suwJ4gk71knAWLa4=; b
=Jrsm+\r\n\t'
  b'pCovRIoQIRyp8Fl0L6JHOI8sbZy2obx7O28JF2iTlTWmX33Rhlq9403XRklw
N3JA\r\n\t7KSPq'
  b'MTp30Qdx6yIUaADwQqlO+QMuQq/QxBHdjeebmdhgVfjhqxrzTbSMww/ZNhL\
r\n\tYwv/QM/oDH'
  b'bXiLSUlB3Qrg+9wsE/0jU/EOisiU=\r\nX-Sasl-enc: 8ZJ+4ZRE8AGPzdL
RWQFivGymJb8pa'
  b'4G9JGcb7k4xKn+I 1457298962\r\nReceived: from [192.168.1.14]
(75-137-1-34.d'
  b'hcp.nwnn.ga.charter.com [75.137.1.34])\r\n\tby mail.messagin
gengine.com (Po'
  b'stfix) with ESMTPA id C0B366801CD\r\n\tfor <doug@doughellman
n.com>; Sun, 6'
  b' Mar 2016 16:16:02 -0500 (EST)\r\nFrom: Doug Hellmann <doug@
doughellmann.c'
  b'om>\r\nContent-Type: text/plain; charset=us-ascii\r\nContent
-Transfer-En'
  b'coding: 7bit\r\nSubject: PyMOTW Example message 2\r\nMessage
-Id: <00ABCD'
  b'46-DADA-4912-A451-D27165BC3A2F@doughellmann.com>\r\nDate: Su
n, 6 Mar 2016 '
  b'16:16:02 -0500\r\nTo: Doug Hellmann <doug@doughellmann.com>\
r\nMime-Vers'
  b'ion: 1.0 (Mac OS X Mail 9.2 \\(3112\\))\r\nX-Mailer: Apple M
ail (2.3112)'
  b'\r\n\r\n'),
 b')']
```

Полученный с помощью команды FETCH ответ начинается с флагов, а далее он указывает на то, что сообщение содержит 595 байт заголовочных данных. Клиент конструирует кортеж с ответом, а затем закрывает последовательность одиночной строкой, содержащей правую круглую скобку ()), которую сервер посылает в конце ответа на запрос FETCH. Возможно, при таком способе форматирования проще извлекать различные фрагменты информации по отдельности или рекомбинировать ответ и анализировать его на стороне клиента.

Листинг 13.31. imaplib_fetch_separately.py

```
import imaplib
import pprint
import imaplib_connect

with imaplib_connect.open_connection() as c:
    c.select('INBOX', readonly=True)
```

```
print('HEADER:')
typ, msg_data = c.fetch('1', '(BODY.PEEK[HEADER])')
for response_part in msg_data:
    if isinstance(response_part, tuple):
        print(response_part[1])

print('\nBODY TEXT:')
typ, msg_data = c.fetch('1', '(BODY.PEEK[TEXT])')
for response_part in msg_data:
    if isinstance(response_part, tuple):
        print(response_part[1])

print('\nFLAGS:')
typ, msg_data = c.fetch('1', '(FLAGS)')
for response_part in msg_data:
    print(response_part)
    print(imaplib.ParseFlags(response_part))
```

Дополнительным преимуществом раздельного извлечения значений является то, что такой подход упрощает использование метода `ParseFlags()` для выделения флагов из ответа.

```
$ python3 imaplib_fetch_separately.py

HEADER:
b'Return-Path: <doug@doughellmann.com>\r\nReceived: from compute
4.internal (compute4.nyi.internal [10.202.2.44])\r\n\t by sloti2
6t01 (Cyrus 3.0.0-beta1-git-fastmail-12410) with LMTPA;\r\n\t Su
n, 06 Mar 2016 16:16:03 -0500\r\nX-Sieve: CMU Sieve 2.4\r\nX-Spa
m-known-sender: yes, fadd1cf2-dc3a-4984-a08b-02cef3cf1221="doug"
,\r\n ea349ad0-9299-47b5-b632-6ff1e394cc7d="both
hellfly"\r\nXSpam-
score: 0.0\r\nX-Spam-hits: ALL_TRUSTED -1, BAYES_00 -1.9, L
ANGUAGES unknown, BAYES_USED global,\r\n SA_VERSION
3.3.2\r\nXSpam-
source: IP=\'127.0.0.1\', Host=\'unk\', Country=\'unk\', Fr
omHeader=\'com\',\r\n MailFrom=\'com\'\r\nX-Spam-charsets: plai
n=\'us-ascii\'\r\nX-Resolved-to: doughellmann@fastmail.fm\r\nX-D
elivered-to: doug@doughellmann.com\r\nX-Mail-from: doug@doughell
mann.com\r\nReceived: from mx5 ([10.202.2.204])\r\n by compute4
.internal (LMTPProxy); Sun, 06 Mar 2016 16:16:03 -0500\r\nReceiv
ed: from mx5.nyi.internal (localhost [127.0.0.1])\r\n\tby mx5.ny
i.internal (Postfix) with ESMTP id 47CBA280DB3\r\n\tfor <doug@do
ughellmann.com>; Sun, 6 Mar 2016 16:16:03 -0500 (EST)\r\nReceiv
ed: from mx5.nyi.internal (localhost [127.0.0.1])\r\n by mx5.
nyi.internal (Authentication Milter) with ESMTP\r\n id A71788
6846E.30BA4280D81;\r\n Sun, 6 Mar 2016 16:16:03 -0500\r\nAuth
entication-Results: mx5.nyi.internal;\r\n dkim=pass (1024-bit
rsa key) header.d=messagingengine.com header.i=@messagingengine
.com header.b=Jrsm+pCo;\r\n x-local-ip=pass\r\nReceived: from
mailout.nyi.internal (gateway1.nyi.internal [10.202.2.221])\r\n
\t(using TLSv1.2 with cipher ECDHE-RSA-AES256-GCM-SHA384 (256/25
6 bits))\r\n\t(No client certificate requested)\r\n\tby mx5.nyi.
```

```
internal (Postfix) with ESMTPS id 30BA4280D81\r\n\tfor <doug@dou
ghellmann.com>; Sun, 6 Mar 2016 16:16:03 -0500 (EST)\r\nReceive
d: from compute2.internal (compute2.nyi.internal [10.202.2.42])\
r\n\tby mailout.nyi.internal (Postfix) with ESMTP id 1740420D0A\
r\n\tfor <doug@doughellmann.com>; Sun, 6 Mar 2016 16:16:03 -050
0 (EST)\r\nReceived: from frontend2 ([10.202.2.161])\r\n by com
pute2.internal (MEProxy); Sun, 06 Mar 2016 16:16:03 -0500\r\nDKI
M-Signature: v=1; a=rsa-sha1; c=relaxed/relaxed; d=\r\n\tmessagi
ngengine.com; h=content-transfer-encoding:content-type\r\n\t:dat
e:from:message-id:mime-version:subject:to:x-sasl-enc\r\n\t:x-sas
l-enc; s=smtpout; bh=P98NTsEo015suwJ4gk71knAWLa4=; b=Jrsm+\r\n\t
pCovRIoQIRyp8Fl0L6JHOI8sbZy2obx7O28JF2iTlTWmX33Rhlq9403XRklwN3JA
\r\n\t7KSPqMTp30Qdx6yIUaADwQqlO+QMuQq/QxBHdjeebmdhgVfjhqxrzTbSMw
w/ZNhL\r\n\tYwv/QM/oDHbXiLSUlB3Qrg+9wsE/0jU/EOisiU=\r\nX-Sasl-en
c: 8ZJ+4ZRE8AGPzdLRWQFivGymJb8pa4G9JGcb7k4xKn+I 1457298962\r\nRe
ceived: from [192.168.1.14] (75-137-1-34.dhcp.nwnn.ga.charter.co
m [75.137.1.34])\r\n\tby mail.messagingengine.com (Postfix) with
ESMTPA id C0B366801CD\r\n\tfor <doug@doughellmann.com>; Sun, 6
Mar 2016 16:16:02 -0500 (EST)\r\nFrom: Doug Hellmann <doug@doug
hellmann.com>\r\nContent-Type: text/plain; charset=us-ascii\r\nC
ontent-Transfer-Encoding: 7bit\r\nSubject: PyMOTW Example messag
e 2\r\nMessage-Id: <00ABCD46-DADA-4912-A451-D27165BC3A2F@doughel
lmann.com>\r\nDate: Sun, 6 Mar 2016 16:16:02 -0500\r\nTo: Doug H
ellmann <doug@doughellmann.com>\r\nMime-Version: 1.0 (Mac OS X M
ail 9.2 \\(3112\\))\r\nX-Mailer: Apple Mail (2.3112)\r\n\r\n'
BODY TEXT:
b'This is the second example message.\r\n'
FLAGS:
b'1 (FLAGS ())'
()
```

13.4.10. Извлечение всего сообщения

Как было показано ранее, клиент может запрашивать у сервера конкретные части сообщений по отдельности. Также существует возможность извлечения всего почтового сообщения целиком, отформатированного в соответствии с требованиями документа **RFC 822**[15], и его анализа с помощью классов, содержащихся в модуле `email`.

Листинг 13.32. `imaplib_fetch_rfc822.py`

```
import imaplib
import email
import email.parser

import imaplib_connect


with imaplib_connect.open_connection() as c:
    c.select('INBOX', readonly=True)
```

[15] https://tools.ietf.org/html/rfc822

```
    typ, msg_data = c.fetch('1', '(RFC822)')
    for response_part in msg_data:
        if isinstance(response_part, tuple):
            email_parser = email.parser.BytesFeedParser()
            email_parser.feed(response_part[1])
            msg = email_parser.close()
            for header in ['subject', 'to', 'from']:
                print('{:^8}: {}'.format(
                    header.upper(), msg[header]))
```

Предлагаемый модулем email синтаксический анализатор значительно упрощает доступ к сообщениям и манипулирование ими. В этом примере выводятся лишь несколько заголовков для каждого сообщения.

```
$ python3 imaplib_fetch_rfc822.py

SUBJECT : PyMOTW Example message 2
TO      : Doug Hellmann <doug@doughellmann.com>
FROM    : Doug Hellmann <doug@doughellmann.com>
```

13.4.11. Выгрузка сообщений

Чтобы добавить в почтовый ящик новое сообщение, нужно создать экземпляр Message и передать его методу append() вместе с временной меткой.

Листинг 13.33. imaplib_append.py

```
import imaplib
import time
import email.message
import imaplib_connect

new_message = email.message.Message()
new_message.set_unixfrom('pymotw')
new_message['Subject'] = 'subject goes here'
new_message['From'] = 'pymotw@example.com'
new_message['To'] = 'example@example.com'
new_message.set_payload('This is the body of the message.\n')

print(new_message)

with imaplib_connect.open_connection() as c:
    c.append('INBOX', '',
             imaplib.Time2Internaldate(time.time()),
             str(new_message).encode('utf-8'))

    # Отобразить заголовки для всех сообщений в почтовом ящике
    c.select('INBOX')
    typ, [msg_ids] = c.search(None, 'ALL')
    for num in msg_ids.split():
        typ, msg_data = c.fetch(num, '(BODY.PEEK[HEADER])')
        for response_part in msg_data:
            if isinstance(response_part, tuple):
```

```
            print('\n{}:'.format(num))
            print(response_part[1])
```

В этом примере полезной нагрузкой (payload) является тело сообщения в виде простого текста. Класс `Message` поддерживает также MIME-сообщения с составным типом содержимого.

```
$ python3 imaplib_append.py

Subject: subject goes here
From: pymotw@example.com
To: example@example.com
This is the body of the message.

b'1':
b'Return-Path: <doug@doughellmann.com>\r\nReceived: from compute
4.internal (compute4.nyi.internal [10.202.2.44])\r\n\t by sloti2
6t01 (Cyrus 3.0.0-beta1-git-fastmail-12410) with LMTPA;\r\n\t Su
n, 06 Mar 2016 16:16:03 -0500\r\nX-Sieve: CMU Sieve 2.4\r\nX-Spa
m-known-sender: yes, fadd1cf2-dc3a-4984-a08b-02cef3cf1221="doug"
,\r\n ea349ad0-9299-47b5-b632-6ff1e394cc7d="both
hellfly"\r\nXSpam-
score: 0.0\r\nX-Spam-hits: ALL_TRUSTED -1, BAYES_00 -1.9, L
ANGUAGES unknown, BAYES_USED global,\r\n SA_VERSION
3.3.2\r\nXSpam-
source: IP=\'127.0.0.1\', Host=\'unk\', Country=\'unk\', Fr
omHeader=\'com\',\r\n MailFrom=\'com\'\r\nX-Spam-charsets: plai
n=\'us-ascii\'\r\nX-Resolved-to: doughellmann@fastmail.fm\r\nX-D
elivered-to: doug@doughellmann.com\r\nX-Mail-from: doug@doughell
mann.com\r\nReceived: from mx5 ([10.202.2.204])\r\n by compute4
.internal (LMTPProxy); Sun, 06 Mar 2016 16:16:03 -0500\r\nReceiv
ed: from mx5.nyi.internal (localhost [127.0.0.1])\r\n\tby mx5.ny
i.internal (Postfix) with ESMTP id 47CBA280DB3\r\n\tfor <doug@do
ughellmann.com>; Sun, 6 Mar 2016 16:16:03 -0500 (EST)\r\nReceiv
ed: from mx5.nyi.internal (localhost [127.0.0.1])\r\n by mx5.
nyi.internal (Authentication Milter) with ESMTP\r\n id A71788
6846E.30BA4280D81;\r\n Sun, 6 Mar 2016 16:16:03 -0500\r\nAuth
entication-Results: mx5.nyi.internal;\r\n dkim=pass (1024-bit
rsa key) header.d=messagingengine.com header.i=@messagingengine
.com header.b=Jrsm+pCo;\r\n x-local-ip=pass\r\nReceived: from
mailout.nyi.internal (gateway1.nyi.internal [10.202.2.221])\r\n
\t(using TLSv1.2 with cipher ECDHE-RSA-AES256-GCM-SHA384 (256/25
6 bits))\r\n\t(No client certificate requested)\r\n\tby mx5.nyi.
internal (Postfix) with ESMTPS id 30BA4280D81\r\n\tfor <doug@dou
ghellmann.com>; Sun, 6 Mar 2016 16:16:03 -0500 (EST)\r\nReceive
d: from compute2.internal (compute2.nyi.internal [10.202.2.42])\
r\n\tby mailout.nyi.internal (Postfix) with ESMTP id 1740420D0A\
r\n\tfor <doug@doughellmann.com>; Sun, 6 Mar 2016 16:16:03 -050
0 (EST)\r\nReceived: from frontend2 ([10.202.2.161])\r\n by com
pute2.internal (MEProxy); Sun, 06 Mar 2016 16:16:03 -0500\r\nDKI
M-Signature: v=1; a=rsa-sha1; c=relaxed/relaxed; d=\r\n\tmessagi
ngengine.com; h=content-transfer-encoding:content-type\r\n\t:dat
e:from:message-id:mime-version:subject:to:x-sasl-enc\r\n\t:x-sas
```

```
l-enc; s=smtpout; bh=P98NTsEo015suwJ4gk71knAWLa4=; b=Jrsm+\r\n\t
pCovRIoQIRyp8Fl0L6JHOI8sbZy2obx7O28JF2iTlTWmX33Rhlq9403XRklwN3JA
\r\n\t7KSPqMTp30Qdx6yIUaADwQqlO+QMuQq/QxBHdjeebmdhgVfjhqxrzTbSMw
w/ZNhL\r\n\tYwv/QM/oDHbXiLSUlB3Qrg+9wsE/0jU/EOisiU=\r\nX-Sasl-en
c: 8ZJ+4ZRE8AGPzdLRWQFivGymJb8pa4G9JGcb7k4xKn+I 1457298962\r\nRe
ceived: from [192.168.1.14] (75-137-1-34.dhcp.nwnn.ga.charter.co
m [75.137.1.34])\r\n\tby mail.messagingengine.com (Postfix) with
ESMTPA id C0B366801CD\r\n\tfor <doug@doughellmann.com>; Sun, 6
Mar 2016 16:16:02 -0500 (EST)\r\nFrom: Doug Hellmann <doug@doug
hellmann.com>\r\nContent-Type: text/plain; charset=us-ascii\r\nC
ontent-Transfer-Encoding: 7bit\r\nSubject: PyMOTW Example messag
e 2\r\nMessage-Id: <00ABCD46-DADA-4912-A451-D27165BC3A2F@doughel
lmann.com>\r\nDate: Sun, 6 Mar 2016 16:16:02 -0500\r\nTo: Doug H
ellmann <doug@doughellmann.com>\r\nMime-Version: 1.0 (Mac OS X M
ail 9.2 \\(3112\\))\r\nX-Mailer: Apple Mail (2.3112)\r\n\r\n'

b'2':
b'Subject: subject goes here\r\nFrom: pymotw@example.com\r\nTo:
example@example.com\r\n\r\n'
```

13.4.12. Перемещение и копирование сообщений

Как только сообщение оказалось на сервере, его можно перемещать и копировать с помощью методов `move()` и `copy()` так, как если бы оно располагалось на локальном компьютере. Как и метод `fetch()`, эти методы могут работать с диапазонами идентификаторов сообщений.

Листинг 13.34. `imaplib_archive_read.py`

```
import imaplib
import imaplib_connect

with imaplib_connect.open_connection() as c:
    # Найти сообщения с флагом "SEEN" в папке INBOX
    c.select('INBOX')
    typ, [response] = c.search(None, 'SEEN')
    if typ != 'OK':
        raise RuntimeError(response)
    msg_ids = ','.join(response.decode('utf-8').split(' '))

    # Создать новый почтовый ящик "Example.Today"
    typ, create_response = c.create('Example.Today')
    print('CREATED Example.Today:', create_response)

    # Копировать сообщения
    print('COPYING:', msg_ids)
    c.copy(msg_ids, 'Example.Today')

    # Просмотреть результаты
    c.select('Example.Today')
    typ, [response] = c.search(None, 'ALL')
    print('COPIED:', response)
```

Этот сценарий создает новый почтовый ящик в папке Example и копирует в него все прочитанные сообщения из папки INBOX.

```
$ python3 imaplib_archive_read.py

CREATED Example.Today: [b'Completed']
COPYING: 2
COPIED: b'1'
```

Повторное выполнение этого же сценария показывает важность проверки кодов завершения. Вместо возбуждения исключения при вызове метода create() для создания нового почтового ящика выводится сообщение, извещающее о том, что почтовый ящик с указанным именем уже существует.

```
$ python3 imaplib_archive_read.py

CREATED Example.Today: [b'[ALREADYEXISTS] Mailbox already exists
']
COPYING: 2
COPIED: b'1 2'
```

13.4.13. Удаление сообщений

Несмотря на то что многие современные клиенты используют модель "корзины" для работы с удаленными сообщениями, сообщения редко отправляются в фактическую папку корзины. Вместо этого набор их флагов обновляется за счет добавления флага \Deleted. Операция "опустошения" корзины выполняется посредством команды EXPUNGE. В следующем примере выполняется поиск всех архивированных сообщений со строкой "Lorem ipsum" в поле темы, и для каждого из них устанавливается флаг удаления, после чего с помощью повторного запроса к серверу отображаются все сообщения, которые все еще остаются в папке.

Листинг 13.35. imaplib_delete_messages.py

```
import imaplib
import imaplib_connect
from imaplib_list_parse import parse_list_response

with imaplib_connect.open_connection() as c:
    c.select('Example.Today')

    # Каковы идентификаторы сообщений в почтовом ящике?
    typ, [msg_ids] = c.search(None, 'ALL')
    print('Starting messages:', msg_ids)

    # Найти сообщение (сообщения)
    typ, [msg_ids] = c.search(
        None,
        '(SUBJECT "subject goes here")',
    )
    msg_ids = ','.join(msg_ids.decode('utf-8').split(' '))
    print('Matching messages:', msg_ids)
```

```
    # Каково текущее состояние флагов?
    typ, response = c.fetch(msg_ids, '(FLAGS)')
    print('Flags before:', response)

    # Изменить флаг Deleted
    typ, response = c.store(msg_ids, '+FLAGS', r'(\Deleted)')

    # Каким стало состояние флагов?
    typ, response = c.fetch(msg_ids, '(FLAGS)')
    print('Flags after:', response)

    # Удалить сообщение навсегда
    typ, response = c.expunge()
    print('Expunged:', response)

    # Каковы идентификаторы сообщений,
    # оставшихся в почтовом ящике?
    typ, [msg_ids] = c.search(None, 'ALL')
    print('Remaining messages:', msg_ids)
```

Явный вызов метода `expunge()` удаляет сообщения, но вызов метода `close()` дает тот же результат.

```
$ python3 imaplib_delete_messages.py

Response code: OK
Server response: b'(\\HasChildren) "." Example'
Parsed response: ('\\HasChildren', '.', 'Example')
Server response: b'(\\HasNoChildren) "." Example.Today'
Parsed response: ('\\HasNoChildren', '.', 'Example.Today')
Server response: b'(\\HasNoChildren) "." Example.2016'
Parsed response: ('\\HasNoChildren', '.', 'Example.2016')
Server response: b'(\\HasNoChildren) "." Archive'
Parsed response: ('\\HasNoChildren', '.', 'Archive')
Server response: b'(\\HasNoChildren) "." "Deleted Messages"'
Parsed response: ('\\HasNoChildren', '.', 'Deleted Messages')
Server response: b'(\\HasNoChildren) "." INBOX'
Parsed response: ('\\HasNoChildren', '.', 'INBOX')
Starting messages: b'1 2'
Matching messages: 1,2
Flags before: [b'1 (FLAGS (\\Seen))', b'2 (FLAGS (\\Seen))']
Flags after: [b'1 (FLAGS (\\Deleted \\Seen))', b'2 (FLAGS (\\Del
eted \\Seen))']
Expunged: [b'2', b'1']
Remaining messages: b''
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `imaplib`[16].
- `rfc822`. Модуль `rfc822` включает синтаксический анализатор RFC 822/RFC 5322.

[16] https://docs.python.org/3.5/library/imaplib.html

- `email`. Модуль `email` обеспечивает синтаксический анализ сообщений электронной почты.
- `mailbox` (раздел 13.3). Локальный синтаксический анализатор сообщений электронной почты.
- Класс `ConfigParser`. Обеспечивает чтение и запись конфигурационных файлов.
- IMAP Information Center[17]. Специализированный сайт Вашингтонского университета, содержащий обширную информацию касательно протокола IMAP, вместе с исходным кодом набора инструментов для клиентов и серверов электронной почты.
- **RFC 3501**[18]. *Internet Message Access Protocol.*
- **RFC 5322**[19]. *Internet Message Format.*
- *IMAP Backup Script*[20]. Сценарий для резервного копирования электронной почты с сервера IMAP.
- *IMAPClient* (Menno Smits)[21]. Высокоуровневый клиент для взаимодействия с серверами IMAP.
- `offlineimap`[22]. Приложение на языке Python, обеспечивающее синхронизацию локального набора почтовых ящиков с сервером IMAP.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `imaplib` (раздел A.6.20).

[17] www.washington.edu/imap/
[18] https://tools.ietf.org/html/rfc3501.html
[19] https://tools.ietf.org/html/rfc5322.html
[20] http://snipplr.com/view/7955/imap-backup-script/
[21] http://imapclient.freshfoo.com/
[22] www.offlineimap.org

Глава 14

Строительные блоки приложений

Мощные возможности стандартной библиотеки Python обусловлены ее размерами. Она включает реализации настолько большого количества всевозможных аспектов структуры программы, что разработчики могут сосредоточиться только на том, что делает их приложения уникальными, а не заниматься каждый раз заново написанием их базовых компонентов. Данная глава охватывает наиболее часто встречающиеся повторно используемые строительные блоки, которые предназначены для решения задач, общих для многих приложений.

Модуль `argparse` (раздел 14.1) предоставляет интерфейс для синтаксического анализа и проверки допустимости аргументов командной строки. Он поддерживает преобразование аргументов из строк в целые числа и другие типы, выполнение функций обратного вызова, если встречается соответствующий параметр, установку значений по умолчанию для параметров, не предоставленных пользователем, и автоматическое создание инструкций по использованию программы. Модуль `getopt` (раздел 14.2) реализует низкоуровневую модель обработки аргументов, доступную программам на языке C и сценариям командной оболочки. Он предоставляет меньше возможностей, чем другие библиотеки, обеспечивающие анализ параметров, но в силу простоты и известности предлагаемых им средств он пользуется широкой популярностью.

Для предоставления пользователю командной подсказки интерактивные программы должны использовать модуль `readline` (раздел 14.3), который включает инструменты для управления историей команд, автозавершения ввода и интерактивного изменения вводимых данных с использованием привязок комбинаций клавиш к командам редакторов `emacs` и `vi`. Безопасный ввод пользователем паролей и другой секретной информации без отображения ее на экране обеспечивает модуль `getpass` (раздел 14.4).

Модуль `cmd` (раздел 14.5) включает фреймворк, предназначенный для создания интерактивных утилит в стиле командной оболочки Unix, управляемых командами. Он предоставляет основной цикл ожидания событий и обрабатывает взаимодействие с пользователем, поэтому приложению остается лишь реализовать функции-обработчики обратного вызова для отдельных команд.

Модуль `shlex` (раздел 14.6) — это лексический анализатор для языков, следующих синтаксису командной оболочки Unix, в соответствии с которым строки состоят из лексем, разделенных пробелами. Этот модуль распознает кавычки и экранирующие последовательности, которые обеспечивают обработку текста, содержащего встроенные пробелы, как одиночной лексемы. Он хорошо справляется с разбивкой входного потока на лексемы в случае таких специализированных языков, как языки конфигурационных файлов или языки программирования.

Модуль `configparser` (раздел 14.7) упрощает управление конфигурационными файлами приложений. Он позволяет сохранять пользовательские установки

в промежутках между запусками программы и читать их при очередном запуске программы и даже использовать в качестве простого файлового формата данных.

Приложения, развертываемые в реальной среде, должны предоставлять пользователям отладочную информацию. Простые сообщения об ошибках и трассировочная информация несомненно полезны, но в тех случаях, когда воспроизведение проблемы сопряжено с трудностями, журнал, в котором подробно протоколируются все действия, может непосредственно указать на цепочку событий, которая привела к сбою в работе программы. Модуль `logging` (раздел 14.8) включает полнофункциональный API протоколирования событий посредством ведения журналов, который поддерживает многопоточность и даже интерфейсы с демонами удаленных журналов для организации централизованной регистрации событий.

Одним из наиболее распространенных программных шаблонов в среде Unix является построчный фильтр, позволяющий читать, изменять и записывать данные. Чтение данных из файлов не вызывает трудностей, но модуль `fileinput` (раздел 14.9) предлагает самый простой способ создания приложений на основе фильтров. Его API представляет собой линейный итератор, поочередно возвращающий каждую входную строку, поэтому основным телом приложения может быть простой цикл `for`. Модуль `fileinput` обеспечивает построчное чтение файлов, имена которых определяются из командной строки или читаются из стандартного потока ввода, поэтому инструменты, созданные поверх модуля `fileinput`, могут применяться непосредственно к файлу или работать в качестве составной части конвейера.

Модуль `atexit` (раздел 14.10) позволяет планировать запуск функций, которые должны выполняться при завершении работы программы в интерпретаторе. Вызов зарегистрированных функций при выходе из программы можно использовать для освобождения ресурсов при завершении работы с удаленными службами, закрытии файлов и в других случаях.

Модуль `sched` (раздел 14.11) реализует планировщик запуска событий в будущие моменты времени. Его API не навязывает определение времени запуска, так что для этих целей может быть использовано как собственно время, так и другие события, в том числе действия, предпринимаемые интерпретатором.

14.1. `argparse`: анализ параметров и аргументов командной строки

Модуль `argparse` включает инструменты, предназначенные для создания процессоров аргументов и параметров командной строки. Он был добавлен в версии Python 2.7, заменив собой модуль `optparse`. Реализация модуля `argparse` поддерживает средства, которые было нелегко добавить в модуль `optparse` и которые требовали внесения в API изменений, не обеспечивающих обратной совместимости, что и побудило разработчиков к созданию нового модуля. В настоящее время модуль `optparse` считается устаревшим и не рекомендуется к применению.

14.1.1. Настройка синтаксического анализатора

Используя модуль `argparse`, прежде всего необходимо создать объект парсера (синтаксического анализатора) и передать ему информацию об ожидаемых

аргументах. Далее этот анализатор можно использовать для обработки аргументов командной строки при запуске программы. Конструктор класса анализатора `ArgumentParser` поддерживает ряд аргументов, позволяющих задать описание, используемое в справке по программе, и другие глобальные параметры.

```
import argparse
parser = argparse.ArgumentParser(
    description='This is a PyMOTW sample program',
)
```

14.1.2. Определение аргументов

Библиотека `argparse` содержит весь необходимый набор средств для обработки аргументов командной строки. Аргументы могут инициировать различные действия, определяемые аргументом `action` метода `add_argument()`. В число поддерживаемых действий входят: сохранение аргумента (одиночного или как части списка), сохранение постоянного значения, если встречается соответствующий аргумент (включая специальную обработку истинных и ложных значений для булевых переключателей), подсчет количества вхождений аргумента и вызов функций для выполнения специальных инструкций обработки.

Действием по умолчанию является сохранение значения аргумента. Если предоставляется тип, то сохраняемое значение предварительно преобразуется в этот тип. Если предоставлен аргумент `dest`, то указанное в нем имя присваивается значению, и это имя используется при разборе аргументов командной строки.

14.1.3. Анализ командной строки

Определив все аргументы, следует проанализировать командную строку, передав последовательность строк аргументов методу `parse_args()`. По умолчанию аргументы берутся из среза `sys.argv[1:]`, но можно использовать любой список строк. Параметры обрабатываются с использованием синтаксиса GNU/POSIX, поэтому значения опций (параметров) и аргументов могут следовать в произвольном порядке.

Метод `parse_args()` возвращает объект `Namespace`, содержащий аргументы команды. Значения аргументов сохраняются в виде атрибутов этого объекта. Таким образом, если аргумент добавлялся со значением `dest`, равным `"myoption"`, то значение данного аргумента доступно в виде атрибута `args.myoption`.

14.1.4. Простые примеры

Ниже представлен простой пример, в котором используются три различных параметра: булев (`-a`), строковый (`-b`) и целочисленный (`-c`).

Листинг 14.1. `argparse_short.py`

```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
```

```
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args(['-a', '-bval', '-c', '3']))
```

Для передачи значений параметрам, заданным одиночными символами, предусмотрено несколько способов. В предыдущем примере были использованы две различные формы: `-bval` и `-c val`.

```
$ python3 argparse_short.py

Namespace(a=True, b='val', c=3)
```

Типом значения, ассоциированным с параметром `'c'` в выводе, является целое число, поскольку объект `ArgumentParser` был извещен о необходимости преобразования аргумента в этот тип, прежде чем сохранять его.

Точно так же обрабатываются длинные имена параметров, включающие более одного символа.

Листинг 14.2. `argparse_long.py`

```
import argparse

parser = argparse.ArgumentParser(
    description='Example with long option names',
)

parser.add_argument('--noarg', action="store_true",
                    default=False)
parser.add_argument('--witharg', action="store",
                    dest="witharg")
parser.add_argument('--witharg2', action="store",
                    dest="witharg2", type=int)

print(
    parser.parse_args(
        ['--noarg', '--witharg', 'val', '--witharg2=3']
    )
)
```

Результаты аналогичны предыдущим.

```
$ python3 argparse_long.py

Namespace(noarg=True, witharg='val', witharg2=3)
```

Модуль `argparse` — полноценный парсер аргументов командной строки. Он обрабатывает как необязательные, так и обязательные аргументы.

Листинг 14.3. `argparse_arguments.py`

```
import argparse

parser = argparse.ArgumentParser(
```

```
    description='Example with nonoptional arguments',
)

parser.add_argument('count', action="store", type=int)
parser.add_argument('units', action="store")

print(parser.parse_args())
```

В этом примере аргумент `count` — целое число, тогда как аргумент `units` сохраняется как строка. Если какой-либо из этих аргументов не указан в командной строке или его значение не может быть преобразовано в соответствующий тип, то выводится сообщение об ошибке.

```
$ python3 argparse_arguments.py 3 inches

Namespace(count=3, units='inches')

$ python3 argparse_arguments.py some inches

usage: argparse_arguments.py [-h] count units
argparse_arguments.py: error: argument count: invalid int value:
'some'

$ python3 argparse_arguments.py

usage: argparse_arguments.py [-h] count units
argparse_arguments.py: error: the following arguments are
required: count, units
```

14.1.4.1. Аргумент `action`

Может быть инициировано любое из перечисленных ниже шести действий, если соответствующий аргумент встречается в командной строке.

- `store`. Сохранить значение после его необязательного преобразования в другой тип. Это действие выполняется по умолчанию, если никакое другое действие не указано явно.
- `store_const`. Сохранить значение, определенное как часть спецификации аргумента, а не как значение, происходящее от анализируемого аргумента. Как правило, этой возможностью пользуются для реализации флагов командной строки, не являющихся булевыми значениями.
- `store_true/store_false`. Сохранить подходящее булево значение. Эти действия используются для реализации булевых переключателей.
- `append`. Сохранить значение в списке. Если аргумент повторяется, сохраняются несколько значений.
- `append_const`. Сохранить значение, определенное в спецификации аргумента, в списке.
- `version`. Вывести подробную информацию о версии программы и завершить выполнение.

Следующий пример демонстрирует использование этих типов действий и минимальную конфигурацию, необходимую для работы каждого из них.

Листинг 14.4. argparse_action.py

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-s', action='store',
                    dest='simple_value',
                    help='Store a simple value')

parser.add_argument('-c', action='store_const',
                    dest='constant_value',
                    const='value-to-store',
                    help='Store a constant value')

parser.add_argument('-t', action='store_true',
                    default=False,
                    dest='boolean_t',
                    help='Set a switch to true')
parser.add_argument('-f', action='store_false',
                    default=True,
                    dest='boolean_f',
                    help='Set a switch to false')

parser.add_argument('-a', action='append',
                    dest='collection',
                    default=[],
                    help='Add repeated values to a list')

parser.add_argument('-A', action='append_const',
                    dest='const_collection',
                    const='value-1-to-append',
                    default=[],
                    help='Add different values to list')
parser.add_argument('-B', action='append_const',
                    dest='const_collection',
                    const='value-2-to-append',
                    help='Add different values to list')

parser.add_argument('--version', action='version',
                    version='%(prog)s 1.0')

results = parser.parse_args()
print('simple_value     = {!r}'.format(results.simple_value))
print('constant_value   = {!r}'.format(results.constant_value))
print('boolean_t        = {!r}'.format(results.boolean_t))
print('boolean_f        = {!r}'.format(results.boolean_f))
print('collection       = {!r}'.format(results.collection))
print('const_collection = {!r}'.format(results.const_collection))
```

Параметры `-t` и `-f` сконфигурированы для изменения разных значений, причем каждый из них сохраняет значение `True` или `False`. Значения аргумента `dest` параметров для `-A` и `-B` одинаковы, поэтому их постоянные значения присоединяются к одному и тому же списку.

```
$ python3 argparse_action.py -h

usage: argparse_action.py [-h] [-s SIMPLE_VALUE] [-c] [-t] [-f]
                          [-a COLLECTION] [-A] [-B] [--version]

optional arguments:
  -h, --help       show this help message and exit
  -s SIMPLE_VALUE  Store a simple value
  -c               Store a constant value
  -t               Set a switch to true
  -f               Set a switch to false
  -a COLLECTION    Add repeated values to a list
  -A               Add different values to list
  -B               Add different values to list
  --version        show program's version number and exit

$ python3 argparse_action.py -s value

simple_value     = 'value'
constant_value   = None
boolean_t        = False
boolean_f        = True
collection       = []
const_collection = []

$ python3 argparse_action.py -c

simple_value     = None
constant_value   = 'value-to-store'
boolean_t        = False
boolean_f        = True
collection       = []
const_collection = []

$ python3 argparse_action.py -t

simple_value     = None
constant_value   = None
boolean_t        = True
boolean_f        = True
collection       = []
const_collection = []

$ python3 argparse_action.py -f

simple_value     = None
constant_value   = None
boolean_t        = False
```

```
boolean_f        = False
collection       = []
const_collection = []

$ python3 argparse_action.py -a one -a two -a three

a two -a three
simple_value     = None
constant_value   = None
boolean_t        = False
boolean_f        = True
collection       = ['one', 'two', 'three']
const_collection = []

$ python3 argparse_action.py -B -A

simple_value     = None
constant_value   = None
boolean_t        = False
boolean_f        = True
collection       = []
const_collection = ['value-2-to-append', 'value-1-to-append']

$ python3 argparse_action.py --version

argparse_action.py 1.0
```

14.1.4.2. Префиксы параметров

Синтаксис параметров, используемый по умолчанию, основан на принятом в Unix соглашении относительно обозначения переключателей командной строки префиксами в виде символа дефиса (-). Модуль `argparse` поддерживает и другие префиксы, поэтому программа может выбрать префикс, согласующийся с умолчаниями, принятыми для локальной платформы (например, использовать символ / в случае Windows), или следовать другому соглашению.

Листинг 14.5. `argparse_prefix_chars.py`

```
import argparse

parser = argparse.ArgumentParser(
    description='Change the option prefix characters',
    prefix_chars='-+/',
)

parser.add_argument('-a', action="store_false",
                    default=None,
                    help='Turn A off',
                    )
parser.add_argument('+a', action="store_true",
                    default=None,
                    help='Turn A on',
                    )
```

```
parser.add_argument('//noarg', '++noarg',
                    action="store_true",
                    default=False)

print(parser.parse_args())
```

В этом примере в качестве аргумента `prefix_chars` конструктора экземпляра `ArgumentParser` задается строка, содержащая все символы, которыми разрешается обозначать параметры командной строки. Несмотря на то что аргумент `prefix_chars` устанавливает разрешенные символы переключателей, синтаксис каждого конкретного переключателя специфицируется определениями отдельных аргументов. Эта кажущаяся избыточность обеспечивает возможность явного контроля над тем, являются ли параметры, использующие различные префиксы, просто разными псевдонимами (например, обеспечивающими независимость синтаксиса командной строки от платформы) или же альтернативными версиями параметра (например, использование символа `+` для включения переключателя и символа `-` для его выключения). В приведенном примере `+a` и `-a` — это независимые аргументы, тогда как аргумент `//noarg` может быть предоставлен и как `++noarg`, но не как `--noarg`.

```
$ python3 argparse_prefix_chars.py -h

usage: argparse_prefix_chars.py [-h] [-a] [+a] [//noarg]

Change the option prefix characters

optional arguments:
  -h, --help        show this help message and exit
  -a                Turn A off
  +a                Turn A on
  //noarg, ++noarg

$ python3 argparse_prefix_chars.py +a

Namespace(a=True, noarg=False)

$ python3 argparse_prefix_chars.py -a

Namespace(a=False, noarg=False)

$ python3 argparse_prefix_chars.py //noarg

Namespace(a=None, noarg=True)

$ python3 argparse_prefix_chars.py ++noarg

Namespace(a=None, noarg=True)

$ python3 argparse_prefix_chars.py --noarg

usage: argparse_prefix_chars.py [-h] [-a] [+a] [//noarg]
argparse_prefix_chars.py: error: unrecognized arguments: --noarg
```

14.1.4.3. Источники аргументов

В рассмотренных ранее примерах список аргументов, передаваемых объекту анализатора, либо предоставлялся в явном виде, либо извлекался из переменной `sys.argv`. Явная передача списка удобна в тех случаях, когда модуль `argparse` используется для обработки инструкций командной строки, источником которых не является сама командная строка (например, когда их источником служит конфигурационный файл).

Листинг 14.6. `argparse_with_shlex.py`

```
import argparse
from configparser import ConfigParser
import shlex

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

config = ConfigParser()
config.read('argparse_with_shlex.ini')
config_value = config.get('cli', 'options')
print('Config  :', config_value)

argument_list = shlex.split(config_value)
print('Arg List:', argument_list)

print('Results :', parser.parse_args(argument_list))
```

В этом примере для чтения конфигурационного файла используется модуль `configparser` (раздел 14.7).

```
[cli]
options = -a -b 2
```

Модуль `shlex` (раздел 14.6) упрощает разбиение строки, хранящейся в конфигурационном файле.

```
$ python3 argparse_with_shlex.py

Config : -a -b 2
Arg List: ['-a', '-b', '2']
Results : Namespace(a=True, b='2', c=None)
```

Вместо того чтобы обрабатывать конфигурационный файл в коде приложения, можно использовать аргумент `fromfile_prefix_chars` конструктора объекта `ArgumentParser` для передачи модулю `argparse` информации о том, как распознать аргумент, определяющий входной файл с набором аргументов для обработки.

Листинг 14.7. argparse_fromfile_prefix_chars.py

```
import argparse
import shlex

parser = argparse.ArgumentParser(description='Short sample app',
                                 fromfile_prefix_chars='@',
                                 )

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args(['@argparse_fromfile_prefix_chars.txt']))
```

Выполнение данного примера приостанавливается, как только встречается аргумент с префиксом @, после чего делается попытка чтения файла с указанным именем для поиска дополнительных аргументов. Этот файл должен содержать по одному аргументу в каждой строке, как показано в следующем листинге.

Листинг 14.8. argparse_fromfile_prefix_chars.txt

```
-a
-b
2
```

Обработка файла argparse_from_prefix_chars.txt дает следующий результат.

```
$ python3 argparse_fromfile_prefix_chars.py

Namespace(a=True, b='2', c=None)
```

14.1.5. Вывод справки

14.1.5.1. Автоматическая генерация справки

Модуль argparse автоматически добавляет параметры для генерации справочных сведений, если он сконфигурирован соответствующим образом. Параметрами генерации справки управляет аргумент add_help конструктора ArgumentParser.

Листинг 14.9. argparse_with_help.py

```
import argparse

parser = argparse.ArgumentParser(add_help=True)

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args())
```

Параметры вывода справки (-h и --help) добавляются по умолчанию, но их можно отключить, установив для аргумента add_help значение False.

Листинг 14.10. argparse_without_help.py

```
import argparse

parser = argparse.ArgumentParser(add_help=False)

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args())
```

Несмотря на то что параметры -h и --help де-факто являются стандартными именами параметров для вызова справки, некоторые приложения либо вообще не должны предоставлять справку, либо используют имена этих параметров для других целей.

```
$ python3 argparse_with_help.py -h

usage: argparse_with_help.py [-h] [-a] [-b B] [-c C]

optional arguments:
  -h, --help  show this help message and exit
  -a
  -b B
  -c C

$ python3 argparse_without_help.py -h

usage: argparse_without_help.py [-a] [-b B] [-c C]
argparse_without_help.py: error: unrecognized arguments: -h
```

14.1.5.2. Адаптация справки

В приложениях, непосредственно обрабатывающих вывод справки, могут применяться вспомогательные методы класса ArgumentParser, с помощью которых можно создавать пользовательские действия (раздел 14.1.7.4), выводящие дополнительные справочные данные.

Листинг 14.11. argparse_custom_help.py

```
import argparse

parser = argparse.ArgumentParser(add_help=True)

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print('print_usage output:')
```

```
parser.print_usage()
print()

print('print_help output:')
parser.print_help()
```

Метод `print_usage()` выводит короткое сообщение о порядке использования анализатора аргументов, тогда как метод `print_help()` выводит полную справку.

```
print_usage output:
usage: argparse_custom_help.py [-h] [-a] [-b B] [-c C]

print_help output:
usage: argparse_custom_help.py [-h] [-a] [-b B] [-c C]

optional arguments:
  -h, --help  show this help message and exit
  -a
  -b B
  -c C
```

Для управления внешним видом справочной информации класс `ArgumentParser` использует класс форматировщика. Чтобы изменить этот класс, следует передать его посредством аргумента `formatter_class` конструктору класса `ArgumentParser` при создании экземпляра. Например, класс `RawDescriptionHelpFormatter` отключает автоматическое заполнение строк с переходом на следующую строку, обеспечиваемое форматировщиком по умолчанию.

Листинг 14.12. `argparse_raw_description_help_formatter.py`

```
import argparse

parser = argparse.ArgumentParser(
    add_help=True,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    description="""
    description
        not
           wrapped""",
    epilog="""
    epilog
      not
         wrapped""",
)

parser.add_argument(
    '-a', action="store_true",
    help="""argument
    help is
    wrapped
    """,
)

parser.print_help()
```

Весь текст описания и эпилога команды останется неизменным.

```
$ python3 argparse_raw_description_help_formatter.py

usage: argparse_raw_description_help_formatter.py [-h] [-a]

    description
        not
           wrapped

optional arguments:
  -h, --help  show this help message and exit
  -a          argument help is wrapped

    epilog
      not
         wrapped
```

Класс `RawTextHelpFormatter` обрабатывает текст справки так, как если бы он был предварительно отформатирован, отображая его в том виде, в каком он задан.

Листинг 14.13. `argparse_raw_text_help_formatter.py`

```
import argparse

parser = argparse.ArgumentParser(
    add_help=True,
    formatter_class=argparse.RawTextHelpFormatter,
    description="""
    description
        not
           wrapped""",
    epilog="""
    epilog
      not
         wrapped""",
)

parser.add_argument(
    '-a', action="store_true",
    help="""argument
    help is not
    wrapped
    """,
)

parser.print_help()
```

Теперь аккуратное расположение текста справки для аргумента `-a` по строкам оказывается нарушенным.

```
$ python3 argparse_raw_text_help_formatter.py

usage: argparse_raw_text_help_formatter.py [-h] [-a]
```

```
    description
        not
           wrapped

optional arguments:
  -h, --help  show this help message and exit
  -a          argument
                  help is not
                  wrapped


    epilog
      not
         wrapped
```

Такие форматировщики могут пригодиться в тех случаях, когда изменение текста примера, приведенного в описании или эпилоге, сделало бы код примера недействительным.

Класс `MetavarTypeHelpFormatter` выводит для каждого параметра не целевую переменную, а имя типа, что может быть полезным в случае приложений с множеством параметров, имеющих различные типы.

Листинг 14.14. `argparse_metavar_type_help_formatter.py`

```
import argparse

parser = argparse.ArgumentParser(
    add_help=True,
    formatter_class=argparse.MetavarTypeHelpFormatter,
)

parser.add_argument('-i', type=int, dest='notshown1')
parser.add_argument('-f', type=float, dest='notshown2')

parser.print_help()
```

Здесь, вместо того чтобы отображать значение аргумента `dest` конструктора `ArgumentParser`, выводится имя типа, связанного с параметром.

```
$ python3 argparse_metavar_type_help_formatter.py

usage: argparse_metavar_type_help_formatter.py [-h] [-i int] [-f float]

optional arguments:
  -h, --help  show this help message and exit
  -i int
  -f float
```

14.1.6. Организация работы анализатора

Модуль `argparse` включает ряд средств, позволяющих организовывать работу анализаторов аргументов командной строки таким образом, чтобы упростить реализацию или повысить удобство использования выводимой справки.

14.1.6.1. Использование общих правил анализаторами

Программисты часто нуждаются в реализации пакета инструментов командной строки, каждый из которых принимает некий набор аргументов, а затем выполняет специализированные действия различного рода. Например, если для всех программ требуется аутентификация пользователей, прежде чем будут предприниматься какие-либо действия, то все они нуждаются в поддержке параметров `--user` и `--password`. Вместо того чтобы явным образом добавлять параметры в каждый объект `ArgumentParser`, можно определить родительский парсер с общими параметрами и использовать в каждой конкретной программе парсер, наследующий поведение в соответствии с этими параметрами.

Первое, что необходимо сделать, — это создать парсер с общими определениями параметров. Поскольку каждый последующий пользователь будет пытаться добавлять те же параметры справки, тем самым вызывая возбуждение исключения, автоматическая генерация справки в базовом парсере должна быть отключена.

Листинг 14.15. `argparse_parent_base.py`

```
import argparse

parser = argparse.ArgumentParser(add_help=False)

parser.add_argument('--user', action="store")
parser.add_argument('--password', action="store")
```

Создадим еще один парсер с родительским набором параметров.

Листинг 14.16. `argparse_uses_parent.py`

```
import argparse
import argparse_parent_base

parser = argparse.ArgumentParser(
    parents=[argparse_parent_base.parser],
)

parser.add_argument('--local-arg',
                    action="store_true",
                    default=False)

print(parser.parse_args())
```

Результирующая программа принимает все три параметра.

```
$ python3 argparse_uses_parent.py -h

usage: argparse_uses_parent.py [-h] [--user USER]
                               [--password PASSWORD]
                               [--local-arg]

optional arguments:
  -h, --help           show this help message and exit
  --user USER
  --password PASSWORD
  --local-arg
```

14.1.6.2. Конфликт параметров

Как было отмечено в предыдущем примере, добавление в парсер двух обработчиков, использующих одинаковые имена аргументов, приводит к возбуждению исключения. Чтобы изменить поведение парсера при разрешении конфликтов имен, следует передать его конструктору аргумент `conflict_handler`. Двумя встроенными обработчиками являются `error` (используется по умолчанию) и `resolve`, который выбирает обработчики, исходя из того, в каком порядке они добавлялись.

Листинг 14.17. `argparse_conflict_handler_resolve.py`

```
import argparse

parser = argparse.ArgumentParser(conflict_handler='resolve')

parser.add_argument('-a', action="store")
parser.add_argument('-b', action="store", help='Short alone')
parser.add_argument('--long-b', '-b',
                    action="store",
                    help='Long and short together')

print(parser.parse_args(['-h']))
```

В данном примере используется последний из обработчиков с заданным именем аргумента. Как следствие, отдельно заданный параметр `-b` маскируется псевдонимом `--long-b`.

```
$ python3 argparse_conflict_handler_resolve.py

usage: argparse_conflict_handler_resolve.py [-h] [-a A]
                                            [--long-b LONG_B]

optional arguments:
  -h, --help            show this help message and exit
  -a A
  --long-b LONG_B, -b LONG_B
                        Long and short together
```

При изменении порядка следования вызовов метода `add_argument()` независимо заданный параметр не маскируется.

Листинг 14.18. `argparse_conflict_handler_resolve2.py`

```
import argparse

parser = argparse.ArgumentParser(conflict_handler='resolve')

parser.add_argument('-a', action="store")
parser.add_argument('--long-b', '-b',
                    action="store",
                    help='Long and short together')
parser.add_argument('-b', action="store", help='Short alone')

print(parser.parse_args(['-h']))
```

Теперь возможно совместное использование обоих параметров.

```
$ python3 argparse_conflict_handler_resolve2.py

usage: argparse_conflict_handler_resolve2.py [-h] [-a A]
                                             [--long-b LONG_B]
                                             [-b B]

optional arguments:
  -h, --help       show this help message and exit
  -a A
  --long-b LONG_B  Long and short together
  -b B             Short alone
```

14.1.6.3. Группы аргументов

Модуль argparse объединяет определения аргументов в группы. По умолчанию он использует две группы: одну для параметров и одну для обязательных позиционных аргументов.

Листинг 14.19. argparse_default_grouping.py

```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('--optional', action="store_true",
                    default=False)
parser.add_argument('positional', action="store")

print(parser.parse_args())
```

Признаком группирования является наличие разделов "positional arguments" и "optional arguments" в выводе справки.

```
$ python3 argparse_default_grouping.py -h

usage: argparse_default_grouping.py [-h] [--optional] positional

Short sample app

positional arguments:
  positional

optional arguments:
  -h, --help show this help message and exit
  --optional
```

Группирование можно настраивать таким образом, чтобы обеспечить более логичный способ организации справки, при котором родственные параметры или значения выводятся вместе. Так, приведенный ранее пример с общими параметрами можно переписать с использованием пользовательского группирования таким образом, чтобы в справке отображались все параметры аутентификации.

Создадим группу "authentication" с помощью метода add_argument_group(), а затем добавим каждый из параметров, связанных с аутентификацией, не в базовый анализатор, а в эту группу.

Листинг 14.20. argparse_parent_with_group.py

```
import argparse

parser = argparse.ArgumentParser(add_help=False)

group = parser.add_argument_group('authentication')

group.add_argument('--user', action="store")
group.add_argument('--password', action="store")
```

Как и перед этим, программа, использующая родительский анализатор на основе групп, включает его в значение parents.

Листинг 14.21. argparse_uses_parent_with_group.py

```
import argparse
import argparse_parent_with_group

parser = argparse.ArgumentParser(
    parents=[argparse_parent_with_group.parser],
)

parser.add_argument('--local-arg',
                    action="store_true",
                    default=False)

print(parser.parse_args())
```

Теперь все параметры, относящиеся к аутентификации, отображаются в выводе справки в виде одной группы.

```
$ python3 argparse_uses_parent_with_group.py -h
usage: argparse_uses_parent_with_group.py [-h] [--user USER]
                                          [--password PASSWORD]
                                          [--local-arg]

optional arguments:
  -h, --help           show this help message and exit
  --local-arg

authentication:
  --user USER
  --password PASSWORD
```

14.1.6.4. Взаимоисключающие параметры

Определение взаимоисключающих параметров — это особый случай группирования параметров. В его основе лежит вызов метода add_mutually_exclusive_group() вместо метода add_argument_group().

Листинг 14.22. argparse_mutually_exclusive.py

```
import argparse

parser = argparse.ArgumentParser()

group = parser.add_mutually_exclusive_group()
group.add_argument('-a', action='store_true')
group.add_argument('-b', action='store_true')

print(parser.parse_args())
```

Модуль argparse строго контролирует использование взаимоисключающих параметров, поэтому возможно предоставление только одного из параметров, входящих в группу.

```
$ python3 argparse_mutually_exclusive.py -h

usage: argparse_mutually_exclusive.py [-h] [-a | -b]

optional arguments:
  -h, --help show this help message and exit
  -a
  -b

$ python3 argparse_mutually_exclusive.py -a

Namespace(a=True, b=False)

$ python3 argparse_mutually_exclusive.py -b

Namespace(a=False, b=True)

$ python3 argparse_mutually_exclusive.py -a -b

usage: argparse_mutually_exclusive.py [-h] [-a | -b]
argparse_mutually_exclusive.py: error: argument -b: not allowed
with argument -a
```

14.1.6.5. Вложенные парсеры

Описанный ранее подход, основанный на применении родительских парсеров, является одним из способов обеспечения совместного использования параметров родственными командами. Альтернативный подход заключается в предварительном объединении команд в одну программу и последующем использовании подчиненных парсеров для обработки каждой части командной строки. Этот подход работает аналогично программам svn, hg и другим, способным выполнять ряд действий, задаваемых в командной строке.

Рассмотрим, например, программу для работы с каталогами файловой системы, в которой определены команды для создания, удаления и вывода содержимого каталога.

Листинг 14.23. `argparse_subparsers.py`

```
import argparse

parser = argparse.ArgumentParser()

subparsers = parser.add_subparsers(help='commands')

# Команда list
list_parser = subparsers.add_parser(
    'list', help='List contents')
list_parser.add_argument(
    'dirname', action='store',
    help='Directory to list')

# Команда create
create_parser = subparsers.add_parser(
    'create', help='Create a directory')
create_parser.add_argument(
    'dirname', action='store',
    help='New directory to create')
create_parser.add_argument(
    '--read-only', default=False, action='store_true',
    help='Set permissions to prevent writing to the directory',
)

# Команда delete
delete_parser = subparsers.add_parser(
    'delete', help='Remove a directory')
delete_parser.add_argument(
    'dirname', action='store', help='The directory to remove')
delete_parser.add_argument(
    '--recursive', '-r', default=False, action='store_true',
    help='Remove the contents of the directory, too',
)

print(parser.parse_args())
```

В выводе справки именованные подчиненные парсеры отображаются как команды, которые могут быть указаны в командной строке в качестве позиционных аргументов.

```
$ python3 argparse_subparsers.py -h

usage: argparse_subparsers.py [-h] {list,create,delete} ...

positional arguments:
  {list,create,delete}  commands
    list                List contents
    create              Create a directory
    delete              Remove a directory

optional arguments:
  -h, --help            show this help message and exit
```

Каждый подчиненный парсер имеет собственную справку, которая описывает аргументы и параметры данной команды.

```
$ python3 argparse_subparsers.py create -h

usage: argparse_subparsers.py create [-h] [--read-only] dirname

positional arguments:
  dirname      New directory to create

optional arguments:
  -h, --help   show this help message and exit
  --read-only  Set permissions to prevent writing to the directory
```

В процессе анализа аргументов объект `Namespace`, возвращенный методом `parse_args()`, включает лишь значения, относящиеся к указанной команде.

```
$ python3 argparse_subparsers.py delete -r foo

Namespace(dirname='foo', recursive=True)
```

14.1.7. Дополнительная обработка аргументов

В приведенных до сих пор примерах было продемонстрировано использование простых булевых флагов, параметров со строковыми или числовыми аргументами и позиционных аргументов. Модуль `argparse` также поддерживает сложные определения аргументов в виде списков переменной длины, перечислений и постоянных значений.

14.1.7.1. Переменный список аргументов

Определение одиночного аргумента можно сконфигурировать так, чтобы учитывалось наличие нескольких аргументов в анализируемой командной строке. В зависимости от количества требуемых или ожидаемых аргументов следует задать для `nargs` одно из значений флагов, приведенных в табл. 14.1.

Таблица 14.1. Флаги, определяющие переменное количество аргументов `argparse`

Значение	Описание
`N`	Абсолютное количество аргументов (например, 3)
`?`	0 или 1 аргумент
`*`	0 или все аргументы
`+`	Все аргументы, количество которых должно быть не менее 1

Листинг 14.24. `argparse_nargs.py`

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('--three', nargs=3)
```

```
parser.add_argument('--optional', nargs='?')
parser.add_argument('--all', nargs='*', dest='all')
parser.add_argument('--one-or-more', nargs='+')

print(parser.parse_args())
```

Парсер навязывает использование инструкций подсчета аргументов и генерирует точную синтаксическую диаграмму в качестве части текста справки по команде.

```
$ python3 argparse_nargs.py -h

usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
                         [--optional [OPTIONAL]]
                         [--all [ALL [ALL ...]]]
                         [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]

optional arguments:
  -h, --help            show this help message and exit
  --three THREE THREE THREE
  --optional [OPTIONAL]
  --all [ALL [ALL ...]]
  --one-or-more ONE_OR_MORE [ONE_OR_MORE ...]

$ python3 argparse_nargs.py

Namespace(all=None, one_or_more=None, optional=None, three=None)

$ python3 argparse_nargs.py --three

usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
                         [--optional [OPTIONAL]]
                         [--all [ALL [ALL ...]]]
                         [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]
argparse_nargs.py: error: argument --three: expected 3 arguments

$ python3 argparse_nargs.py --three a b c

Namespace(all=None, one_or_more=None, optional=None,
three=['a', 'b', 'c'])

$ python3 argparse_nargs.py --optional

Namespace(all=None, one_or_more=None, optional=None, three=None)

$ python3 argparse_nargs.py --optional with_value

Namespace(all=None, one_or_more=None, optional='with_value',
three=None)

$ python3 argparse_nargs.py --all with multiple values

Namespace(all=['with', 'multiple', 'values'], one_or_more=None,
```

```
optional=None, three=None)

$ python3 argparse_nargs.py --one-or-more with_value

Namespace(all=None, one_or_more=['with_value'], optional=None,
three=None)

$ python3 argparse_nargs.py --one-or-more with multiple values

Namespace(all=None, one_or_more=['with', 'multiple', 'values'],
optional=None, three=None)

$ python3 argparse_nargs.py --one-or-more

usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
                [--optional [OPTIONAL]]
                [--all [ALL [ALL ...]]]
                [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]
argparse_nargs.py: error: argument --one-or-more: expected
at least one argument
```

14.1.7.2. Типы аргументов

Если не предписано преобразование строки в другой тип, то модуль `argparse` обрабатывает значения всех аргументов как строки. Параметр `type` метода `add_argument()` определяет функцию преобразования, которую класс `ArgumentParser` использует для преобразования значения аргумента из строки в другой тип.

Листинг 14.25. `argparse_type.py`

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-i', type=int)
parser.add_argument('-f', type=float)
parser.add_argument('--file', type=open)

try:
    print(parser.parse_args())
except IOError as msg:
    parser.error(str(msg))
```

В качестве параметра `type` может быть передан любой вызываемый объект, принимающий одиночный строковый аргумент, в том числе встроенные типы, такие как `int` и `float`, и даже функция `open()`.

```
$ python3 argparse_type.py -i 1

Namespace(f=None, file=None, i=1)

$ python3 argparse_type.py -f 3.14
```

```
Namespace(f=3.14, file=None, i=None)

$ python3 argparse_type.py --file argparse_type.py

Namespace(f=None, file=<_io.TextIOWrapper
name='argparse_type.py' mode='r' encoding='UTF-8'>, i=None)
```

Если выполнить преобразование типа не удается, модуль `argparse` возбуждает исключение. Исключения `TypeError` и `ValueError` автоматически перехватываются и преобразуются в простое сообщение об ошибке, предназначенное для пользователя. Другие исключения, такие как `IOError` в следующем примере, которое генерируется в том случае, если входной файл не существует, должны обрабатываться вызывающим кодом.

```
$ python3 argparse_type.py -i a

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: argument -i: invalid int value: 'a'

$ python3 argparse_type.py -f 3.14.15

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: argument -f: invalid float value:
'3.14.15'

$ python3 argparse_type.py --file does_not_exist.txt

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: [Errno 2] No such file or directory:
'does_not_exist.txt'
```

Чтобы ограничить круг значений, допустимых в качестве входного аргумента, предопределенным набором, следует использовать параметр `choices`.

Листинг 14.26. `argparse_choices.py`

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument(
    '--mode',
    choices=('read-only', 'read-write'),
)

print(parser.parse_args())
```

Если аргумент `--mode` не является одним из допустимых значений, генерируется ошибка, и дальнейшая обработка прекращается.

```
$ python3 argparse_choices.py -h

usage: argparse_choices.py [-h] [--mode {read-only,read-write}]

optional arguments:
```

```
  -h, --help            show this help message and exit
  --mode {read-only,read-write}

$ python3 argparse_choices.py --mode read-only

Namespace(mode='read-only')

$ python3 argparse_choices.py --mode invalid

usage: argparse_choices.py [-h] [--mode {read-only,read-write}]
argparse_choices.py: error: argument --mode: invalid choice:
'invalid' (choose from 'read-only', 'read-write')
```

14.1.7.3. Аргументы, определяющие файлы

Несмотря на то что файловые объекты могут инстанциализироваться посредством одного строкового аргумента, это не обеспечивает возможность доступа к аргументу `mode`, определяющему режим доступа к файлу. Тип `FileType` предоставляет более гибкий способ указания того, что данный аргумент является файлом, позволяя определить также режим доступа и размер буфера.

Листинг 14.27. `argparse_FileType.py`

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-i', metavar='in-file',
                    type=argparse.FileType('rt'))
parser.add_argument('-o', metavar='out-file',
                    type=argparse.FileType('wt'))

try:
    results = parser.parse_args()
    print('Input file:', results.i)
    print('Output file:', results.o)
except IOError as msg:
    parser.error(str(msg))
```

Значением, связанным с именем аргумента, является дескриптор открываемого файла. Ответственность за закрытие файла, когда он больше не используется, возлагается на приложение.

```
$ python3 argparse_FileType.py -h

usage: argparse_FileType.py [-h] [-i in-file] [-o out-file]

optional arguments:
  -h, --help   show this help message and exit
  -i in-file
  -o out-file

$ python3 argparse_FileType.py -i argparse_FileType.py -o tmp_\
```

```
file.txt

Input file: <_io.TextIOWrapper name='argparse_FileType.py'
mode='rt' encoding='UTF-8'>
Output file: <_io.TextIOWrapper name='tmp_file.txt' mode='wt'
encoding='UTF-8'>

$ python3 argparse_FileType.py -i no_such_file.txt

usage: argparse_FileType.py [-h] [-i in-file] [-o out-file]
argparse_FileType.py: error: argument -i: can't open
'no_such_file.txt': [Errno 2] No such file or directory:
'no_such_file.txt'
```

14.1.7.4. Пользовательские действия

Помимо описанных ранее встроенных действий можно определить пользовательские действия, предоставив объект, реализующий Action API. Объект, передаваемый методу `add_argument()` в качестве аргумента `action`, должен принимать параметры, описывающие определяемый аргумент (из числа тех, которые передаются вместе с ним методу `add_argument()`), и возвращать вызываемый объект, который принимает в качестве параметров объект `parser`, обрабатывающий аргументы, пространство имен `namespace`, в котором сохраняются результаты парсинга, значение `value` аргумента, в отношении которого выполняется действие, и параметр `option_string`, который запустил действие.

В качестве удобной отправной точки для определения новых действий предоставляется класс `Action`. Определения аргументов обрабатываются конструктором, поэтому в подклассе необходимо переопределить лишь метод `__call__()`.

Листинг 14.28. `argparse_custom_action.py`

```
import argparse


class CustomAction(argparse.Action):
    def __init__(self,
                 option_strings,
                 dest,
                 nargs=None,
                 const=None,
                 default=None,
                 type=None,
                 choices=None,
                 required=False,
                 help=None,
                 metavar=None):
        argparse.Action.__init__(self,
                                 option_strings=option_strings,
                                 dest=dest,
                                 nargs=nargs,
                                 const=const,
                                 default=default,
```

```
                                type=type,
                                choices=choices,
                                required=required,
                                help=help,
                                metavar=metavar,
                                )
        print('Initializing CustomAction')
        for name, value in sorted(locals().items()):
            if name == 'self' or value is None:
                continue
            print('  {} = {!r}'.format(name, value))
        print()
        return

    def __call__(self, parser, namespace, values,
                 option_string=None):
        print('Processing CustomAction for {}'.format(self.dest))
        print('  parser = {}'.format(id(parser)))
        print('  values = {!r}'.format(values))
        print('  option_string = {!r}'.format(option_string))

        # Выполнить необходимую обработку входных значений
        if isinstance(values, list):
            values = [v.upper() for v in values]
        else:
            values = values.upper()
        # Сохранить результаты в пространстве имен, используя
        # переменную destination, предоставленную конструктору
        setattr(namespace, self.dest, values)
        print()


parser = argparse.ArgumentParser()

parser.add_argument('-a', action=CustomAction)
parser.add_argument('-m', nargs='*', action=CustomAction)

results = parser.parse_args(['-a', 'value',
                             '-m', 'multivalue',
                             'second'])
print(results)
```

Тип значений зависит от значения аргумента `nargs`. Если этот аргумент разрешает использование нескольких значений, то значением `values` будет список, даже если в этом списке содержится всего один элемент. Значение `option_string` также зависит от первоначального определения аргумента. Для обязательных позиционных параметров значением аргумента `option_string` всегда является `None`.

```
$ python3 argparse_custom_action.py

Initializing CustomAction
  dest = 'a'
```

```
  option_strings = ['-a']
  required = False

Initializing CustomAction
  dest = 'm'
  nargs = '*'
  option_strings = ['-m']
  required = False

Processing CustomAction for a
  parser = 4315836992
  values = 'value'
  option_string = '-a'

Processing CustomAction for m
  parser = 4315836992
  values = ['multivalue', 'second']
  option_string = '-m'

Namespace(a='VALUE', m=['MULTIVALUE', 'SECOND'])
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `argparse`[1].
- `configparser` (раздел 14.7). Чтение и запись конфигурационных файлов.
- `shlex` (раздел 14.6). Лексический анализатор команд, использующих синтаксис командной оболочки Unix.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `argparse` (раздел A.6.3).

14.2. `getopt`: анализ параметров командной строки

Модуль `getopt` — это оригинальный лексический анализатор параметров командной строки, следующий правилам, установленным функцией `getopt()` в Unix. Он анализирует последовательность аргументов, содержащихся, например, в срезе `sys.argv[1:]`, и возвращает последовательность кортежей в виде пар (*параметр*, *аргумент*) и список аргументов программы, остающихся после исключения параметров.

Поддерживаемый синтаксис включает короткую и длинную формы параметров командной строки.

```
-a
-bval
-b val
--noarg
--witharg=val
--witharg val
```

[1] https://docs.python.org/3.5/library/argparse.html

Примечание

Модуль `getopt` не считается устаревшим, однако модуль `argparse` (см. раздел 14.1) получает более активную поддержку, и поэтому именно его рекомендуется использовать в новых разработках.

14.2.1. Аргументы функции `getopt()`

Функция `getopt()` получает три аргумента, назначение которых описано ниже.

- Первый аргумент — последовательность аргументов, подлежащих анализу. Обычно эта информация берется из среза `sys.argv[1:]` (имя программы, содержащееся в элементе `sys.arg[0]`, игнорируется).
- Второй аргумент — строка определения односимвольных параметров. Если какой-либо из параметров требует задания аргумента, вслед за его буквой необходимо указать двоеточие.
- Третий аргумент, если он используется, — последовательность параметров с длинными именами. Такие имена могут включать более одного символа, например `--noarg` или `--witharg`. Имена параметров в последовательности не должны включать префикс `--`. Если какой-либо из параметров с длинными именами требует задания аргумента, его имя должно содержать суффикс `=`.

В одном вызове могут сочетаться параметры как с короткими, так и с длинными именами.

14.2.2. Короткая форма параметров

В следующем примере программа принимает три параметра. Параметр `-a` — просто флаг, тогда как параметры `-b` и `-c` требуют задания аргумента. Строкой определения параметров является `"ab:c:"`.

Листинг 14.29. `getopt_short.py`

```
import getopt

opts, args = getopt.getopt(['-a', '-bval', '-c', 'val'], 'ab:c:')

for opt in opts:
    print(opt)
```

В данной программе функции `getopt()` передается список параметров, часть которых требует указания для них соответствующих значений.

```
$ python3 getopt_short.py

('-a', '')
('-b', 'val')
('-c', 'val')
```

14.2.3. Длинная форма параметров

Для программы, которая принимает два параметра, `--noarg` и `--witharg`, последовательность длинных аргументов должна иметь вид `['noarg','witharg=']`.

Листинг 14.30. `getopt_long.py`

```
import getopt

opts, args = getopt.getopt(
    ['--noarg',
     '--witharg', 'val',
     '--witharg2=another'],
    '',
    ['noarg', 'witharg=', 'witharg2='],
)
for opt in opts:
    print(opt)
```

Поскольку данный образец программы не принимает параметров в короткой форме, в качестве второго аргумента функции `getopt()` передается пустая строка.

```
$ python3 getopt_long.py

('--noarg', '')
('--witharg', 'val')
('--witharg2', 'another')
```

14.2.4. Более полный пример

В приведенном ниже листинге представлен более полный пример программы, которая принимает пять параметров: `-o`, `-v`, `--output`, `--verbose` и `--version`. Параметры `-o`, `--output` и `--version` требуют задания аргументов.

Листинг 14.31. `getopt_example.py`

```
import getopt
import sys

version = '1.0'
verbose = False
output_filename = 'default.out'

print('ARGV      :', sys.argv[1:])

try:
    options, remainder = getopt.getopt(
        sys.argv[1:],
        'o:v',
        ['output=',
         'verbose',
         'version=',
         ])
```

```
except getopt.GetoptError as err:
    print('ERROR:', err)
    sys.exit(1)

print('OPTIONS   :', options)

for opt, arg in options:
    if opt in ('-o', '--output'):
        output_filename = arg
    elif opt in ('-v', '--verbose'):
        verbose = True
    elif opt == '--version':
        version = arg

print('VERSION   :', version)
print('VERBOSE   :', verbose)
print('OUTPUT    :', output_filename)
print('REMAINING :', remainder)
```

Эту программу можно вызвать различными способами. Если она вызывается без аргументов, то используются значения, заданные по умолчанию.

```
$ python3 getopt_example.py

ARGV      : []
OPTIONS   : []
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : default.out
REMAINING : []
```

Параметр, заданный одной буквой, можно отделить от его аргумента пробелом.

```
$ python3 getopt_example.py -o foo

ARGV      : ['-o', 'foo']
OPTIONS   : [('-o', 'foo')]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []
```

Возможен и другой вариант, когда параметр и его значение объединяются в один аргумент.

```
$ python3 getopt_example.py -ofoo

ARGV      : ['-ofoo']
OPTIONS   : [('-o', 'foo')]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []
```

Длинная форма параметра также может быть отделена от значения.

```
$ python3 getopt_example.py --output foo

ARGV      : ['--output', 'foo']
OPTIONS   : [('--output', 'foo')]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []
```

Если длинная форма параметра сочетается со своим значением, то имя параметра и значение должны быть разделены знаком равенства =.

```
$ python3 getopt_example.py --output=foo

ARGV      : ['--output=foo']
OPTIONS   : [('--output', 'foo')]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []
```

14.2.5. Сокращение длинной формы параметров

Длинные имена параметров не обязательно должны воспроизводиться в командной строке полностью, коль скоро предоставлен уникальный префикс.

```
$ python3 getopt_example.py --o foo

ARGV      : ['--o', 'foo']
OPTIONS   : [('--output', 'foo')]
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : foo
REMAINING : []
```

Если уникальный префикс не предоставлен, возбуждается исключение.

```
$ python3 getopt_example.py --ver 2.0

ARGV      : ['--ver', '2.0']
ERROR: option --ver not a unique prefix
```

14.2.6. Анализ параметров в стиле GNU

Обычно обработка параметров прекращается сразу же, как только встречается аргумент, не являющийся параметром.

```
$ python3 getopt_example.py -v not_an_option --output foo

ARGV      : ['-v', 'not_an_option', '--output', 'foo']
```

```
OPTIONS   : [('-v', '')]
VERSION   : 1.0
VERBOSE   : True
OUTPUT    : default.out
REMAINING : ['not_an_option', '--output', 'foo']
```

Использование функции `gnu_getopt()` вместо `getopt()` позволяет смешивать в произвольном порядке аргументы командной строки, являющиеся и не являющиеся параметрами.

Листинг 14.32. `getopt_gnu.py`

```
import getopt
import sys

version = '1.0'
verbose = False
output_filename = 'default.out'

print('ARGV      :', sys.argv[1:])

try:
    options, remainder = getopt.gnu_getopt(
        sys.argv[1:],
        'o:v',
        ['output=',
         'verbose',
         'version=',
         ])
except getopt.GetoptError as err:
    print('ERROR:', err)
    sys.exit(1)

print('OPTIONS   :', options)

for opt, arg in options:
    if opt in ('-o', '--output'):
        output_filename = arg
    elif opt in ('-v', '--verbose'):
        verbose = True
    elif opt == '--version':
        version = arg

print('VERSION   :', version)
print('VERBOSE   :', verbose)
print('OUTPUT    :', output_filename)
print('REMAINING :', remainder)
```

После изменения вызова в предыдущем примере различие между двумя подходами становится очевидным.

```
$ python3 getopt_gnu.py -v not_an_option --output foo

ARGV      : ['-v', 'not_an_option', '--output', 'foo']
```

```
OPTIONS  : [('-v', ''), ('--output', 'foo')]
VERSION  : 1.0
VERBOSE  : True
OUTPUT   : foo
REMAINING : ['not_an_option']
```

14.2.7. Прекращение обработки аргументов

Если функция `getopt()` обнаруживает аргумент `--`, то обработка остальных аргументов как параметров прекращается. Эту особенность можно использовать для передачи значений аргументов, которые выглядят как параметры, таких, например, как имена файлов, начинающиеся с символа дефиса (`-`).

```
$ python3 getopt_example.py -v -- --output foo

ARGV      : ['-v', '--', '--output', 'foo']
OPTIONS   : [('-v', '')]
VERSION   : 1.0
VERBOSE   : True
OUTPUT    : default.out
REMAINING : ['--output', 'foo']
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `getopt`[2].
- `argparse` (раздел 14.1). Модуль `argparse` следует использовать вместо модуля `getopt` в новых приложениях.

14.3. `readline`: библиотека GNU Readline

Модуль `readline` предоставляет интерфейс к библиотеке GNU Readline. Его можно использовать для улучшения интерактивных утилит командной строки, упрощая их использование, например, за счет добавления средства автозавершения ввода с помощью клавиши <Tab>.

Примечание

Поскольку модуль `readline` взаимодействует с содержимым консоли, вывод отладочных сообщений затрудняет наблюдение за тем, что происходит в исследуемом коде. В следующих примерах отладочная информация записывается в отдельный файл с помощью модуля `logging` (раздел 14.8). Вывод, направляемый в журнал, приводится в каждом примере.

Примечание

Библиотеки GNU, необходимые для работы модуля `readline`, доступны по умолчанию не на всех платформах. Если ваша система их не включает, то, возможно, вам придется заново скомпилировать интерпретатор Python для активизации этого модуля после установки всех зависимостей. Также имеется автономная версия библиотеки, доступная на сайте

[2] `https://docs.python.org/3.5/library/getopt.html`

Python Package Index под именем gnureadline[3]. В приведенных в этом разделе примерах сначала делается попытка импортировать gnureadline, а в качестве резервного варианта используется readline.

14.3.1. Конфигурирование библиотеки Readline

Возможны два способа конфигурирования базовой библиотеки GNU Readline: с помощью конфигурационного файла и с помощью функции parse_and_bind(). Конфигурационные опции обеспечивают автозавершение ввода, выбор режима редактирования (vi или emacs) и многие другие возможности. Для получения более подробных сведений обратитесь к документации библиотеки GNU Readline.

Проще всего активизировать автозавершение ввода с помощью вызова функции parse_and_bind(). Одновременно можно установить другие параметры. В следующем примере вместо заданного по умолчанию режима редактирования emacs включается режим редактирования vi. Чтобы изменить текущую строку ввода, следует нажать клавишу <ESC> и использовать обычные навигационные клавиши редактора vi, такие как <j>, <k>, <l> и <h>.

Листинг 14.33. readline_parse_and_bind.py

```
try:
    import gnureadline as readline
except ImportError:
    import readline

readline.parse_and_bind('tab: complete')
readline.parse_and_bind('set editing-mode vi')

while True:
    line = input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print('ENTERED: {!r}'.format(line))
```

Ту же самую конфигурацию можно сохранить в виде инструкций в файле, которые библиотека прочитает посредством одного вызова. Пусть в файле myreadline.rc содержатся следующие инструкции.

Листинг 14.34. myreadline.rc

```
# Включить автозавершение ввода
tab: complete
# Использовать режим редактирования vi вместо режима emacs
set editing-mode vi
```

Тогда содержимое этого файла можно прочитать с помощью функции read_init_file().

Листинг 14.35. readline_read_init_file.py

```
try:
    import gnureadline as readline
```

[3] https://pypi.python.org/pypi/gnureadline

```
except ImportError:
    import readline

readline.read_init_file('myreadline.rc')

while True:
    line = input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print('ENTERED: {!r}'.format(line))
```

14.3.2. Автозавершение ввода

В следующей программе предусмотрен встроенный набор возможных команд, в процессе работы с которыми можно использовать средство автозавершения ввода.

Листинг 14.36. readline_completer.py

```
try:
    import gnureadline as readline
except ImportError:
    import readline
import logging

LOG_FILENAME = '/tmp/completer.log'
logging.basicConfig(
    format='%(message)s',
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)


class SimpleCompleter:

    def __init__(self, options):
        self.options = sorted(options)

    def complete(self, text, state):
        response = None
        if state == 0:
            # Это первое обращение к данному тексту,
            # поэтому создаем полный список вариантов
            if text:
                self.matches = [
                    s
                    for s in self.options
                    if s and s.startswith(text)
                ]
                logging.debug('%s matches: %s',
                              repr(text), self.matches)
            else:
                self.matches = self.options[:]
```

```
                logging.debug('(empty input) matches: %s',
                              self.matches)

        # Вернуть элемент с индексом state из списка,
        # если он содержит достаточное количество элементов
        try:
            response = self.matches[state]
        except IndexError:
            response = None
        logging.debug('complete(%s, %s) => %s',
                      repr(text), state, repr(response))
        return response


def input_loop():
    line = ''
    while line != 'stop':
        line = input('Prompt ("stop" to quit): ')
        print('Dispatch {}'.format(line))


# Зарегистрировать функцию завершения ввода
OPTIONS = ['start', 'stop', 'list', 'print']
readline.set_completer(SimpleCompleter(OPTIONS).complete)

# Использовать клавишу <tab> для завершения ввода
readline.parse_and_bind('tab: complete')

# Предложить пользователю ввести текст
input_loop()
```

В этой программе функция `input_loop()` читает одну строку за другой, пока пользователь не введет слово `"stop"`. Более сложная программа могла бы осуществлять фактический разбор строк и выполнять команды.

Класс `SimpleCompleter` поддерживает список вариантов, являющихся кандидатами для завершения ввода. Метод `complete()` предназначен для регистрации в качестве источника завершающего текста. Его аргументами являются строка `text` для завершения ввода и значение `state`, которое указывает, сколько раз вызывалась функция с тем же текстом. Данная функция вызывается повторно, и при каждом вызове значение переменной `state` увеличивается на 1. Метод должен вернуть строку, если имеется строка-кандидат для данного значения `state`, или значение `None` при отсутствии таких строк. Реализация метода `complete()` в предыдущем листинге просматривает набор совпадений, когда `state` равно 0, и возвращает все варианты совпадений по одному за раз при последующих вызовах.

Начальный вывод при запуске этой программы выглядит так.

```
$ python3 readline_completer.py

Prompt ("stop" to quit):
```

При двойном нажатии клавиши <Tab> дважды выводится полный список вариантов команд.

```
$ python3 readline_completer.py

Prompt ("stop" to quit):
list   print  start  stop
Prompt ("stop" to quit):
list   print  start  stop
Prompt ("stop" to quit):
```

Заглянув в журнал, можно увидеть, что метод `complete()` вызывался с двумя независимыми последовательностями значений переменной `state`.

```
$ tail -f /tmp/completer.log

(empty input) matches: ['list', 'print', 'start', 'stop']
complete('', 0) => 'list'
complete('', 1) => 'print'
complete('', 2) => 'start'
complete('', 3) => 'stop'
complete('', 4) => None
(empty input) matches: ['list', 'print', 'start', 'stop']
complete('', 0) => 'list'
complete('', 1) => 'print'
complete('', 2) => 'start'
complete('', 3) => 'stop'
complete('', 4) => None
```

Первая последовательность является результатом первого нажатия клавиши <Tab>. Алгоритм автозавершения запрашивает все варианты продолжения, но не расширяет пустую строку ввода. После второго нажатия клавиши <Tab> список возможных вариантов продолжения заново пересчитывается, чтобы его можно было вывести для пользователя.

Если далее ввести букву `l` и нажать клавишу <Tab>, будет получен следующий вывод:

```
Prompt ("stop" to quit): list
```

Теперь в журнале отображаются другие аргументы для передачи методу `complete()`.

```
'l' matches: ['list']
complete('l', 0) => 'list'
complete('l', 1) => None
```

В результате нажатия клавиши <Enter> метод `input()` вернет значение, и цикл `while` продолжит свою работу.

```
Dispatch list
Prompt ("stop" to quit):
```

Команду, начинающуюся с буквы `s`, можно завершить двумя способами. В результате ввода буквы `s` и последующего нажатия клавиши <Tab> отображаются

два варианта продолжения — start и stop, но средство автозавершения дополняет текст на экране лишь буквой t.

В файле журнала отображается следующая информация.

```
's' matches: ['start', 'stop']
complete('s', 0) => 'start'
complete('s', 1) => 'stop'
complete('s', 2) => None
```

Теперь на экране генерируется следующий вывод:

```
Prompt ("stop" to quit): st
```

Примечание

Если метод completer() возбуждает исключение, оно игнорируется, и модуль readline предполагает, что подходящие варианты для продолжения ввода отсутствуют.

14.3.3. Доступ к буферу автозавершения ввода

Алгоритм автозавершения ввода, предусмотренный в классе SimpleCompleter, просматривает лишь текстовый аргумент, переданный функции, но никак не использует любую другую информацию о внутреннем состоянии модуля readline. Функции модуля readline можно использовать также для манипулирования текстом, находящимся в буфере ввода.

Листинг 14.37. readline_buffer.py

```
try:
    import gnureadline as readline
except ImportError:
    import readline
import logging

LOG_FILENAME = '/tmp/completer.log'
logging.basicConfig(
    format='%(message)s',
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)


class BufferAwareCompleter:

    def __init__(self, options):
        self.options = options
        self.current_candidates = []

    def complete(self, text, state):
        response = None
        if state == 0:
            # Это первое обращение к данному тексту,
            # поэтому создаем полный список вариантов
```

```
        origline = readline.get_line_buffer()
        begin = readline.get_begidx()
        end = readline.get_endidx()
        being_completed = origline[begin:end]
        words = origline.split()

        logging.debug('origline=%s', repr(origline))
        logging.debug('begin=%s', begin)
        logging.debug('end=%s', end)
        logging.debug('being_completed=%s', being_completed)
        logging.debug('words=%s', words)

        if not words:
            self.current_candidates = sorted(
                self.options.keys()
            )
        else:
            try:
                if begin == 0:
                    # Первое слово
                    candidates = self.options.keys()
                else:
                    # Последнее слово
                    first = words[0]
                    candidates = self.options[first]

                if being_completed:
                    # Сопоставить варианты с завершаемой
                    # частью ввода
                    self.current_candidates = [
                        w for w in candidates
                        if w.startswith(being_completed)
                    ]
                else:
                    # При сопоставлении с пустой строкой
                    # использовать все подходящие варианты
                    self.current_candidates = candidates

                logging.debug('candidates=%s',
                              self.current_candidates)

            except (KeyError, IndexError) as err:
                logging.error('completion error: %s', err)
                self.current_candidates = []

    try:
        response = self.current_candidates[state]
    except IndexError:
        response = None
    logging.debug('complete(%s, %s) => %s',
                  repr(text), state, response)
    return response
```

```
def input_loop():
    line = ''
    while line != 'stop':
        line = input('Prompt ("stop" to quit): ')
        print('Dispatch {}'.format(line))


# Зарегистрировать функцию завершения ввода
completer = BufferAwareCompleter({
    'list': ['files', 'directories'],
    'print': ['byname', 'bysize'],
    'stop': [],
})
readline.set_completer(completer.complete)

# Использовать клавишу <tab> для завершения ввода
readline.parse_and_bind('tab: complete')

# Предложить пользователю ввести текст
input_loop()
```

В этом примере выполняется автозавершение команд с дополнительными опциями. Метод `complete()` должен просмотреть позицию завершаемого текста во входном буфере, чтобы определить, является ли он частью первого или последнего слова. Если обнаружено совпадение с первым словом, то в качестве возможных вариантов используются ключи словаря опций. Если это не первое слово, то первое слово используется для поиска подходящих вариантов из словаря опций.

В данном случае имеются три команды верхнего уровня, из которых две имеют подкоманды.

- **list**
  - **files**
  - **directories**
- **print**
  - **byname**
  - **bysize**
- **stop**

Если действовать в той же последовательности, что и ранее, то в результате двукратного нажатия клавиши <Tab> отобразятся три команды верхнего уровня.

```
$ python3 readline_buffer.py

Prompt ("stop" to quit):
list print stop
Prompt ("stop" to quit):
list print stop
Prompt ("stop" to quit):
```

Журнал включает следующую информацию.

```
origline=''
begin=0
end=0
being_completed=
words=[]
complete('', 0) => list
complete('', 1) => print
complete('', 2) => stop
complete('', 3) => None
origline=''
begin=0
end=0
being_completed=
words=[]
complete('', 0) => list
complete('', 1) => print
complete('', 2) => stop
complete('', 3) => None
```

Если первым введенным словом является `'list '` (с пробелом в конце), то варианты для завершения команды будут различными.

```
Prompt ("stop" to quit): list

directories files
```

В журнале отражается тот факт, что завершаемый текст *не* образует полную строку, а является лишь частью, следующей после слова `list`.

```
origline='list '
begin=5
end=5
being_completed=
words=['list']
candidates=['files', 'directories']
complete('', 0) => files
complete('', 1) => directories
complete('', 2) => None
origline='list '
begin=5
end=5
being_completed=
words=['list']
candidates=['files', 'directories']
complete('', 0) => files
complete('', 1) => directories
complete('', 2) => None
```

14.3.4. История ввода

Модуль `readline` автоматически отслеживает историю ввода. Для работы с историей ввода можно использовать два набора функций. Доступ к истории вво-

да текущего сеанса осуществляется с помощью функций get_current_history_length() и get_history_item(). Историю можно сохранить в файле, а впоследствии загрузить, используя функции write_history_file() и read_history_file() соответственно. По умолчанию сохраняется вся история ввода, но с помощью функции set_history_length() можно установить максимальную длину файла. Значению -1 соответствует отсутствие ограничений.

Листинг 14.38. readline_history.py

```
try:
    import gnureadline as readline
except ImportError:
    import readline
import logging
import os

LOG_FILENAME = '/tmp/completer.log'
HISTORY_FILENAME = '/tmp/completer.hist'

logging.basicConfig(
    format='%(message)s',
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)


def get_history_items():
    num_items = readline.get_current_history_length() + 1
    return [
        readline.get_history_item(i)
        for i in range(1, num_items)
    ]


class HistoryCompleter:

    def __init__(self):
        self.matches = []

    def complete(self, text, state):
        response = None
        if state == 0:
            history_values = get_history_items()
            logging.debug('history: %s', history_values)
            if text:
                self.matches = sorted(
                    h
                    for h in history_values
                    if h and h.startswith(text)
                )
            else:
                self.matches = []
            logging.debug('matches: %s', self.matches)
```

```
        try:
            response = self.matches[state]
        except IndexError:
            response = None
        logging.debug('complete(%s, %s) => %s',
                      repr(text), state, repr(response))
        return response


def input_loop():
    if os.path.exists(HISTORY_FILENAME):
        readline.read_history_file(HISTORY_FILENAME)
    print('Max history file length:',
          readline.get_history_length())
    print('Startup history:', get_history_items())
    try:
        while True:
            line = input('Prompt ("stop" to quit): ')
            if line == 'stop':
                break
            if line:
                print('Adding {!r} to the history'.format(line))
    finally:
        print('Final history:', get_history_items())
        readline.write_history_file(HISTORY_FILENAME)


# Зарегистрировать функцию завершения ввода
readline.set_completer(HistoryCompleter().complete)

# Использовать клавишу <tab> для завершения ввода
readline.parse_and_bind('tab: complete')

# Предложить пользователю ввести текст
input_loop()
```

Класс `HistoryCompleter` запоминает все, что вводит пользователь, и использует эти значения для автозавершения последующего ввода.

```
$ python3 readline_history.py

Max history file length: -1
Startup history: []
Prompt ("stop" to quit): foo
Adding 'foo' to the history
Prompt ("stop" to quit): bar
Adding 'bar' to the history
Prompt ("stop" to quit): blah
Adding 'blah' to the history
Prompt ("stop" to quit): b
bar blah
Prompt ("stop" to quit): b
Prompt ("stop" to quit): stop
Final history: ['foo', 'bar', 'blah', 'stop']
```

После ввода буквы b и двукратного нажатия клавиши <Tab> в журнале отображается следующая информация.

```
history: ['foo', 'bar', 'blah']
matches: ['bar', 'blah']
complete('b', 0) => 'bar'
complete('b', 1) => 'blah'
complete('b', 2) => None
history: ['foo', 'bar', 'blah']
matches: ['bar', 'blah']
complete('b', 0) => 'bar'
complete('b', 1) => 'blah'
complete('b', 2) => None
```

При последующем запуске этого сценария вся история ввода читается из файла.

```
$ python3 readline_history.py

Max history file length: -1
Startup history: ['foo', 'bar', 'blah', 'stop']
Prompt ("stop" to quit):
```

Также доступны функции, обеспечивающие удаление отдельных элементов истории и ее полную очистку.

14.3.5. Функции-перехватчики

Действия, являющиеся частью интерактивного взаимодействия, можно инициировать с помощью функций-перехватчиков. Перехватчик запуска вызывается непосредственно перед выводом интерактивной подсказки, перехватчик предввода — после ее вывода, но до чтения текста, вводимого пользователем.

Листинг 14.39. `readline_hooks.py`

```
try:
    import gnureadline as readline
except ImportError:
    import readline


def startup_hook():
    readline.insert_text('from startup_hook')


def pre_input_hook():
    readline.insert_text(' from pre_input_hook')
    readline.redisplay()


readline.set_startup_hook(startup_hook)
readline.set_pre_input_hook(pre_input_hook)
readline.parse_and_bind('tab: complete')
```

```
while True:
    line = input('Prompt ("stop" to quit): ')
    if line == 'stop':
        break
    print('ENTERED: {!r}'.format(line))
```

Любая из этих функций предоставляет отличную возможность использовать функцию `insert_text()` для изменения буфера ввода.

```
$ python3 readline_hooks.py

Prompt ("stop" to quit): from startup_hook from pre_input_hook
```

Если буфер изменен в теле перехватчика предввода, то необходимо обновить экран, вызвав функцию `redisplay()`.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `readline`[4].
- *GNU Readline Library*[5]. Документация библиотеки GNU Readline.
- *Readline init file format*[6]. Описание формата файла инициализации и конфигурирования библиотеки Readline.
- Сайт Effbot: *The readline module*[7]. Руководство Effbot по использованию модуля `readline`.
- `gnureadline`[8]. Версия `readline`, статически связанная с библиотекой GNU Readline, доступная для многих платформ и устанавливаемая посредством программы `pip`.
- `pyreadline`[9]. Модуль Python, предоставляющий GNU Readline API для использования на платформах Windows.
- `cmd` (раздел 14.5). Модуль `cmd` интенсивно использует модуль `readline` для реализации автозавершения ввода при работе с командным интерфейсом. Некоторые примеры, приведенные в этом разделе, были взяты в адаптированном виде из раздела, посвященного модулю `cmd`.
- `rlcompleter`. Данный модуль использует модуль `readline` для добавления возможностей автозавершения ввода в интерактивный интерпретатор Python.

14.4. `getpass`: безопасный ввод пароля

Во многих случаях пользователю, взаимодействующему с программой через терминал, приходится вводить пароль, и в подобных ситуациях никогда не помешает позволить делать это таким образом, чтобы никто из посторонних не смог подсмотреть его на экране. Именно такой безопасный способ ввода пароля обеспечивает модуль `getpass`.

[4] https://docs.python.org/3.5/library/readline.html
[5] http://tiswww.case.edu/php/chet/readline/readline.html
[6] http://tiswww.case.edu/php/chet/readline/readline.html#SEC10
[7] http://sandbox.effbot.org/librarybook/readline.htm
[8] https://pypi.python.org/pypi/gnureadline
[9] http://ipython.org/pyreadline.html

14.4.1. Пример

Функция `getpass()` выводит подсказку, а затем читает вводимый пользователем текст, пока не будет нажата клавиша <Enter>. Введенный текст возвращается вызывающему коду в виде строки.

Листинг 14.40. `getpass_defaults.py`

```
import getpass

try:
    p = getpass.getpass()
except Exception as err:
    print('ERROR:', err)
else:
    print('You entered:', p)
```

По умолчанию выводится подсказка "Password:", если в вызывающем коде не был определен другой текст.

```
$ python3 getpass_defaults.py

Password:
You entered: sekret
```

Вместо этой подсказки можно использовать любой другой подходящий текст.

Листинг 14.41. `getpass_prompt.py`

```
import getpass

p = getpass.getpass(prompt='What is your favorite color? ')
if p.lower() == 'blue':
    print('Right. Off you go.')
else:
    print('Auuuuugh!')
```

С целью повышения безопасности некоторые программы требуют ввода целой фразы, а не просто слова-пароля.

```
$ python3 getpass_prompt.py

What is your favorite color?
Right. Off you go.

$ python3 getpass_prompt.py

What is your favorite color?
Auuuuugh!
```

По умолчанию функция `getpass()` использует для вывода строки стандартный поток `sys.stdout`. В случае программ, которые выводят в поток `sys.stdout` полезную информацию, подсказку лучше направлять в другой поток, например `sys.stderr`.

Листинг 14.42. getpass_stream.py

```
import getpass
import sys

p = getpass.getpass(stream=sys.stderr)
print('You entered:', p)
```

Использование потока `sys.stderr` для вывода подсказки позволяет перенаправить стандартный поток вывода в канал или файл. В этом случае значение, введенное пользователем, не будет отображаться на экране.

```
$ python3 getpass_stream.py >/dev/null

Password:
```

14.4.2. Использование функции getpass() без терминала

В Unix, для того чтобы можно было отключить эхо-отображение ввода, функции `getpass()` требуется терминал (*tty*), которым она могла бы управлять через интерфейс, предоставляемый модулем `termios`. При таком подходе значения, поступающие из нетерминального потока, перенаправленного на стандартный ввод, не будут читаться. Вместо этого функция `getpass()` пытается получить доступ к терминалу для процесса, и если ей это удается, то никакой ошибки не возникает.

```
$ echo "not sekret" | python3 getpass_defaults.py

Password:
You entered: sekret
```

В этом случае ответственность за определение того, что входной поток не является терминальным, и использование альтернативного метода для чтения входных данных возлагается на вызывающий код.

Листинг 14.43. getpass_noterminal.py

```
import getpass
import sys

if sys.stdin.isatty():
    p = getpass.getpass('Using getpass: ')
else:
    print('Using readline')
    p = sys.stdin.readline().rstrip()

print('Read: ', p)
```

Вывод в случае чтения пароля с терминала.

```
$ python3 ./getpass_noterminal.py

Using getpass:
Read: sekret
```

Вывод в случае чтения пароля не с терминала.

```
$ echo "sekret" | python3 ./getpass_noterminal.py

Using readline
Read: sekret
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `getpass`[10].
- `readline` (раздел 14.3). Интерактивная библиотека для работы с командной строкой.

14.5. cmd: построчные командные процессоры

Модуль `cmd` содержит общедоступный класс `Cmd`, предназначенный для использования в качестве базового класса для интерактивных оболочек и других интерпретаторов команд. По умолчанию он использует модуль `readline` (раздел 14.3) для обработки интерактивного ввода, редактирования командной строки и автозавершения команд.

14.5.1. Обработка команд

Интерпретатор команд, создаваемый с помощью модуля `cmd`, использует цикл для чтения входных строк, их синтаксического анализа и последующей передачи команды соответствующему обработчику. Входные строки разбиваются на две части: команду и другой текст, находящийся в данной строке. Например, если пользователь введет `foo bar`, а класс интерпретатора включает метод с именем `do_foo()`, то этот метод будет вызван с `"bar"` в качестве единственного аргумента.

Маркер конца файла передается методу `do_EOF()`. Если обработчик команды возвращает значение, эквивалентное `True`, то выполняется корректный выход из программы. Таким образом, чтобы обеспечить корректный выход из интерпретатора, необходимо реализовать метод `do_EOF()`, который будет возвращать значение `True`.

Ниже приведен пример простой программы, которая поддерживает команду "greet".

Листинг 14.44. `cmd_simple.py`

```
import cmd


class HelloWorld(cmd.Cmd):

    def do_greet(self, line):
        print("hello")

    def do_EOF(self, line):
        return True
```

[10] https://docs.python.org/3.5/library/getpass.html

```
if __name__ == '__main__':
    HelloWorld().cmdloop()
```

Запуск программы в интерактивном режиме позволяет продемонстрировать выполнение команд и некоторые возможности, предоставляемые классом Cmd.

```
$ python3 cmd_simple.py

(Cmd)
```

Первое, на что следует обратить внимание, — это командная подсказка, (Cmd). Ее можно настраивать с помощью атрибута prompt. Значение подсказки — динамическое. Иными словами, если обработчик команды изменяет атрибут prompt, то для приглашения к вводу следующей команды используется новое значение.

```
Documented commands (type help <topic>):
========================================
help

Undocumented commands:
======================
EOF greet
```

В класс Cmd встроена команда help. Будучи вызванной без аргументов, она отображает список доступных команд. Если ввод включает имя команды, выводится более подробная информация, ограниченная описанием этой команды.

Если команда — greet, то для ее обработки вызывается метод do_greet().

```
(Cmd) greet
hello
```

Если в классе не предусмотрен обработчик для конкретной команды, то вызывается метод default(), которому в качестве аргумента передается вся входная строка. Встроенная реализация метода default() выводит сообщение об ошибке.

```
(Cmd) foo
*** Unknown syntax: foo
```

Поскольку метод do_EOF() возвращает значение True, нажатие комбинации клавиш <Ctrl+D> (для Windows — <Ctrl+Z>) приводит к выходу из интерпретатора.

```
(Cmd) ^D$
```

Символ новой строки не выводится при выходе из программы, поэтому результаты выглядят немного неопрятно.

14.5.2. Аргументы команд

Следующий пример включает усовершенствования, устраняющие некоторые недостатки предыдущей программы, и справку для команды greet.

Листинг 14.45. cmd_arguments.py

```
import cmd


class HelloWorld(cmd.Cmd):

    def do_greet(self, person):
        """greet [person]
        Greet the named person"""
        if person:
            print("hi,", person)
        else:
            print('hi')

    def do_EOF(self, line):
        return True

    def postloop(self):
        print()


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

Добавленная в метод `do_greet()` строка документации становится текстом справки для данной команды.

```
$ python3 cmd_arguments.py

(Cmd) help

Documented commands (type help <topic>):
========================================
greet help

Undocumented commands:
======================
EOF

(Cmd) help greet
greet [person]
        Greet the named person
```

В выводе справки отображается информация о том, что команда `greet` имеет один необязательный аргумент: `person`. Несмотря на то что этот аргумент необязательный, различие между командой и методом обратного вызова очевидно. Метод всегда получает аргумент, но иногда его значением является пустая строка. Обработчик команды отвечает за определение того, является ли пустой аргумент допустимым значением или необходимо продолжить анализ и обработку команды. В данном примере, если предоставляется имя человека, то выводится персонализированное приветствие.

```
(Cmd) greet Alice
hi, Alice
(Cmd) greet
hi
```

Независимо от того, предоставил ли пользователь аргумент или не предоставил, передаваемое обработчику значение не включает саму команду. Это упрощает анализ команды в обработчике, особенно если имеется несколько аргументов.

14.5.3. Активная справка

Форматирование справочного текста в предыдущем примере все еще страдает некоторыми недостатками. Поскольку источником справки является строка документирования, справочный текст наследует все отступы из исходного файла. Можно было бы удалить лишние пробелы непосредственно в источнике, но тогда неудачно отформатированным оказался бы код приложения. Лучшее решение – реализовать обработчик справки для команды `greet`, присвоив ему имя `help_greet()`. Обработчик справки обеспечивает получение текста справки для команды с указанным именем.

Листинг 14.46. `cmd_do_help.py`

```
try:
    import gnureadline
    import sys
    sys.modules['readline'] = gnureadline
except ImportError:
    pass

import cmd


class HelloWorld(cmd.Cmd):

    def do_greet(self, person):
        if person:
            print("hi,", person)
        else:
            print('hi')

    def help_greet(self):
        print('\n'.join([
            'greet [person]',
            'Greet the named person',
        ]))

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

В этом примере текст справки остается статическим, но имеет более привлекательный внешний вид. Кроме того, существует возможность использовать состояние предыдущей команды для адаптации справочного текста к текущему контексту.

```
$ python3 cmd_do_help.py

(Cmd) help greet
greet [person]
Greet the named person
```

Решение о том, следует ли вывести справочное сообщение на экран или просто вернуть его для последующей обработки другим кодом, принимается самим обработчиком справки.

14.5.4. Автозавершение ввода

Класс `Cmd` включает методы, обеспечивающие поддержку автозавершения ввода команд на основании их имен. Пользователь активизирует автозавершение нажатием клавиши <Tab> в процессе ввода команды. Если имеется несколько возможных вариантов продолжения ввода, то двойное нажатие клавиши <Tab> позволяет вывести на экран их полный список.

Примечание

Необходимые для работы модуля `readline` библиотеки GNU доступны по умолчанию не на всех платформах. В подобных случаях автозавершение ввода с помощью клавиши <Tab> может не работать. Советы относительно установки необходимых библиотек в случае их отсутствия приведены в разделе 14.3, посвященном модулю `readline`.

```
$ python3 cmd_do_help.py

(Cmd) <tab><tab>
EOF greet help
(Cmd) h<tab>
(Cmd) help
```

Как только стала известна команда, завершение аргумента обрабатывается методами с префиксом `complete_`. Это позволяет новым обработчикам автозавершения ввода сформировать список возможных вариантов завершения, используя любые критерии (например, путем запроса к базе данных или поиска файла или каталога в файловой системе). В данном случае в коде запрограммирован набор друзей (`FRIENDS`), для которых выводится менее формальное приветствие, чем то, которое используется для пользователей с другими именами или анонимных незнакомцев. Вероятно, реальная программа хранила бы его где-то в другом месте, а затем читала однократно и кешировала содержимое для анализа в случае необходимости.

Листинг 14.47. `cmd_arg_completion.py`

```
try:
    import gnureadline
```

```
    import sys
    sys.modules['readline'] = gnureadline
except ImportError:
    pass

import cmd


class HelloWorld(cmd.Cmd):

    FRIENDS = ['Alice', 'Adam', 'Barbara', 'Bob']

    def do_greet(self, person):
        "Greet the person"
        if person and person in self.FRIENDS:
            greeting = 'hi, {}!'.format(person)
        elif person:
            greeting = 'hello, {}'.format(person)
        else:
            greeting = 'hello'
        print(greeting)

    def complete_greet(self, text, line, begidx, endidx):
        if not text:
            completions = self.FRIENDS[:]
        else:
            completions = [
                f
                for f in self.FRIENDS
                if f.startswith(text)
            ]
        return completions

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

При наличии введенного текста метод `complete_greet()` возвращает список друзей, имена которых соответствуют вводу. В противном случае возвращается полный список друзей.

```
$ python3 cmd_arg_completion.py

(Cmd) greet <tab><tab>
Adam Alice Barbara Bob
(Cmd) greet A<tab><tab>
Adam Alice
(Cmd) greet Ad<tab>
(Cmd) greet Adam
hi, Adam!
```

Если предоставленное имя не встречается в списке друзей, то выводится формальное приветствие.

```
(Cmd) greet Joe
hello, Joe
```

14.5.5. Переопределение методов базового класса

Класс Cmd включает несколько методов, которые могут быть переопределены в качестве перехватчиков для выполнения определенных действий или изменения поведения базового класса. Приведенный ниже пример не является исчерпывающим, но иллюстрирует применение методов, которые удобно использовать в повседневной работе.

Листинг 14.48. cmd_illustrate_methods.py

```
try:
    import gnureadline
    import sys
    sys.modules['readline'] = gnureadline
except ImportError:
    pass

import cmd


class Illustrate(cmd.Cmd):
    "Иллюстрирует использование метода базового класса."

    def cmdloop(self, intro=None):
        print('cmdloop({})'.format(intro))
        return cmd.Cmd.cmdloop(self, intro)

    def preloop(self):
        print('preloop()')

    def postloop(self):
        print('postloop()')

    def parseline(self, line):
        print('parseline({!r}) =>'.format(line), end='')
        ret = cmd.Cmd.parseline(self, line)
        print(ret)
        return ret

    def onecmd(self, s):
        print('onecmd({})'.format(s))
        return cmd.Cmd.onecmd(self, s)

    def emptyline(self):
        print('emptyline()')
        return cmd.Cmd.emptyline(self)
```

```
    def default(self, line):
        print('default({})'.format(line))
        return cmd.Cmd.default(self, line)

    def precmd(self, line):
        print('precmd({})'.format(line))
        return cmd.Cmd.precmd(self, line)

    def postcmd(self, stop, line):
        print('postcmd({}, {})'.format(stop, line))
        return cmd.Cmd.postcmd(self, stop, line)

    def do_greet(self, line):
        print('hello,', line)

    def do_EOF(self, line):
        "Exit"
        return True


if __name__ == '__main__':
    Illustrate().cmdloop('Illustrating the methods of cmd.Cmd')
```

Метод `cmdloop()` — это основной рабочий цикл интерпретатора. Как правило, его не приходится переопределять, поскольку имеются перехватчики `preloop()` и `postloop()`.

На каждой итерации цикла `cmdloop()` вызывается метод `onecmd()`, передающий команду соответствующему обработчику. Фактическая строка ввода анализируется методом `parseline()`, который возвращает кортеж, содержащий команду и оставшуюся часть строки.

Если строка пуста, вызывается метод `emptyline()`, и заданная по умолчанию реализация вновь выполняет предыдущую команду. Если строка содержит команду, то сначала вызывается метод `precmd()`, а затем выполняется поиск и вызов соответствующего обработчика. Если обработчик команды не найден, вместо него вызывается метод `default()`. После этого вызывается метод `postcmd()`.

Ниже представлен пример рабочего сеанса с добавленными инструкциями вывода на печать.

```
$ python3 cmd_illustrate_methods.py

cmdloop(Illustrating the methods of cmd.Cmd)
preloop()
Illustrating the methods of cmd.Cmd
(Cmd) greet Bob
precmd(greet Bob)
onecmd(greet Bob)
parseline(greet Bob) => ('greet', 'Bob', 'greet Bob')
hello, Bob
postcmd(None, greet Bob)
(Cmd) ^Dprecmd(EOF)
onecmd(EOF)
```

```
parseline(EOF) => ('EOF', '', 'EOF')
postcmd(True, EOF)
postloop()
```

14.5.6. Конфигурирование класса Cmd с помощью атрибутов

Для управления интерпретаторами команд можно использовать не только описанные ранее методы, но и ряд атрибутов. В атрибуте `prompt` можно задать строку, которая будет выводиться в каждом приглашении ко вводу очередной команды. Атрибут `intro` содержит текст приветствия, выводимого в начале работы программы. Если необходимо изменить этот текст, то можно передать методу `cmdloop()` новый вариант текста в качестве аргумента или задать его непосредственно в классе. Для форматирования текста выводимой справки используются атрибуты `doc_header`, `misc_header`, `undoc_header` и `ruler`.

Листинг 14.49. `cmd_attributes.py`

```
import cmd


class HelloWorld(cmd.Cmd):

    prompt = 'prompt: '
    intro = "Simple command processor example."

    doc_header = 'doc_header'
    misc_header = 'misc_header'
    undoc_header = 'undoc_header'

    ruler = '-'

    def do_prompt(self, line):
        "Изменить интерактивную подсказку"
        self.prompt = line + ': '

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    HelloWorld().cmdloop()
```

В этом примере демонстрируется обработчик команд, предоставляющий пользователю возможность управления подсказкой в интерактивных сеансах.

```
$ python3 cmd_attributes.py

Simple command processor example.
prompt: prompt hello
hello: help

doc_header
```

```
----------
help  prompt

undoc_header
------------
EOF

hello:
```

14.5.7. Выполнение команд оболочки

Класс `Cmd` дополняет стандартную обработку команд двумя специальными префиксами. Вопросительный знак (`?`) эквивалентен встроенной команде `help` и может использоваться наравне с ней. Восклицательный знак (`!`) соответствует методу `do_shell()` и предназначен для выполнения других команд, как показано в следующем примере.

Листинг 14.50. `cmd_do_shell.py`

```
import cmd
import subprocess


class ShellEnabled(cmd.Cmd):

    last_output = ''

    def do_shell(self, line):
        "Run a shell command"
        print("running shell command:", line)
        sub_cmd = subprocess.Popen(line,
                                   shell=True,
                                   stdout=subprocess.PIPE)
        output = sub_cmd.communicate()[0].decode('utf-8')
        print(output)
        self.last_output = output

    def do_echo(self, line):
        """Print the input, replacing '$out' with
        the output of the last shell command
        """
        # Явно ненадежный вариант
        print(line.replace('$out', self.last_output))

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    ShellEnabled().cmdloop()
```

В этой реализации команды `echo` строка `$out` заменяется выводом предыдущей команды оболочки.

```
$ python3 cmd_do_shell.py

(Cmd) ?

Documented commands (type help <topic>):
========================================
echo  help  shell

Undocumented commands:
======================
EOF

(Cmd) ? shell
Run a shell command
(Cmd) ? echo
Print the input, replacing '$out' with
        the output of the last shell command
(Cmd) shell pwd
running shell command: pwd
.../pymotw-3/source/cmd

(Cmd) ! pwd
running shell command: pwd
.../pymotw-3/source/cmd

(Cmd) echo $out
.../pymotw-3/source/cmd
```

14.5.8. Альтернативные варианты ввода

По умолчанию объект `Cmd` взаимодействует с пользователем посредством модуля `readline` (раздел 14.3), но также существует возможность передачи последовательности команд в стандартный поток ввода с помощью стандартных средств перенаправления ввода оболочки Unix.

```
$ echo help | python3 cmd_do_help.py

(Cmd)
Documented commands (type help <topic>):
========================================
greet help

Undocumented commands:
======================
EOF

(Cmd)
```

Чтобы обеспечить чтение ввода непосредственно из файла, в сценарий необходимо внести некоторые изменения. Поскольку модуль `readline` (раздел 14.3) взаимодействует с терминалом, а не со стандартным потоком ввода, при чтении

ввода из файла его следует отключить. Кроме того, чтобы избавиться от лишних подсказок, загромождающих экран, можно задать в качестве подсказки пустую строку. В приведенной ниже видоизмененной версии примера *HelloWorld* показано, как открыть файл и передать его содержимое сценарию в качестве ввода.

Листинг 14.51. `cmd_file.py`

```
import cmd


class HelloWorld(cmd.Cmd):

    # Отключить использование модуля rawinput
    use_rawinput = False

    # Не отображать подсказку после чтения каждой команды
    prompt = ''

    def do_greet(self, line):
        print("hello,", line)

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    import sys
    with open(sys.argv[1], 'rt') as input:
        HelloWorld(stdin=input).cmdloop()
```

Установив для переменной `use_rawinput` значение `False`, а для подсказки — пустую строку, можно вызвать сценарий для работы с входным файлом, содержащим по одной команде в каждой строке.

Листинг 14.52. `cmd_file.txt`

```
greet
greet Alice and Bob
```

Выполнение примера с этими входными данными дает следующий вывод.

```
$ python3 cmd_file.py cmd_file.txt

hello,
hello, Alice and Bob
```

14.5.9. Извлечение команд из переменной `sys.argv`

Команды могут передаваться программе не только из консольного ввода или файла, но и в виде аргументов командной строки. Это можно сделать путем непосредственного вызова метода `onecmd()`, как показано в следующем примере.

Листинг 14.53. cmd_argv.py

```
import cmd


class InteractiveOrCommandLine(cmd.Cmd):
    """Принимает команды через обычную интерактивную подсказку
    или из командной строки.
    """

    def do_greet(self, line):
        print('hello,', line)

    def do_EOF(self, line):
        return True


if __name__ == '__main__':
    import sys
    if len(sys.argv) > 1:
        InteractiveOrCommandLine().onecmd(' '.join(sys.argv[1:]))
    else:
        InteractiveOrCommandLine().cmdloop()
```

Поскольку методу `onecmd()` передается единственный аргумент в виде строки, то перед передачей аргументов программе их необходимо конкатенировать.

```
$ python3 cmd_argv.py greet Command-Line User

hello, Command-Line User

$ python3 cmd_argv.py

(Cmd) greet Interactive User
hello, Interactive User
(Cmd)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `cmd`[11].
- `cmd2`[12]. Улучшенная версия модуля `cmd`, предлагающая дополнительные возможности.
- GNU Readline[13]. Библиотека функций, обеспечивающих возможность редактирования входных строк в процессе их ввода.
- `readline` (раздел 14.3). Интерфейс стандартной библиотеки Python, предназначенный для работы с библиотекой GNU Readline.
- `subprocess` (раздел 10.1). Модуль, используемый для управления другими процессами и их выводом.

[11] https://docs.python.org/3.5/library/cmd.html

[12] http://pypi.python.org/pypi/cmd2

[13] http://tiswww.case.edu/php/chet/readline/rltop.html

14.6. shlex: лексический анализ синтаксисов в стиле командной оболочки Unix

Модуль shlex реализует класс, предназначенный для анализа простых видов синтаксиса в стиле оболочки Unix. Его можно использовать для написания специализированных языков или для анализа строк, содержащих кавычки (задача не настолько простая, как может показаться на первый взгляд).

14.6.1. Анализ строк, содержащих кавычки

При обработке входного текста часто возникает проблема идентификации последовательности слов, заключенной в кавычки, как единого целого. Разбиение текста на основе кавычек не всегда работает так, как ожидается, особенно при наличии вложенных кавычек. Обратимся к следующему примеру.

```
This string has embedded "double quotes" and
'single quotes' in it, and even "a 'nested example'".
```

Действуя прямолинейно, можно было бы сконструировать регулярное выражение для нахождения частей текста вне кавычек и отделения их от текста, заключенного в кавычки, или наоборот. Однако результирующее регулярное выражение было бы чересчур сложным и чреватым ошибками, обусловленными, например, такими факторами, как наличие апострофов, и даже простыми опечатками. Лучше всего использовать полноценный лексический анализатор, например анализатор, предоставляемый модулем shlex. Следующий сценарий выводит лексемы, идентифицированные во входном файле с помощью класса shlex.

Листинг 14.54. shlex_example.py

```
import shlex
import sys

if len(sys.argv) != 2:
    print('Please specify one filename on the command line.')
    sys.exit(1)

filename = sys.argv[1]
with open(filename, 'r') as f:
    body = f.read()
print('ORIGINAL: {!r}'.format(body))
print()

print('TOKENS:')
lexer = shlex.shlex(body)
for token in lexer:
    print('{!r}'.format(token))
```

При запуске этого примера с файлом, содержащим текст с кавычками, анализатор создает корректный список лексем (TOKENS).

```
$ python3 shlex_example.py quotes.txt

ORIGINAL: 'This string has embedded "double quotes" and\n\'singl
```

```
e quotes\' in it, and even "a \'nested example\'".\n'

TOKENS:
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
','
'and'
'even'
'"a \'nested example\'"'
'.'
```

Данный анализатор корректно обрабатывает также изолированные кавычки и апострофы. Пусть имеется следующий входной файл:

```
This string has an embedded apostrophe, doesn't it?
```

При анализе лексемы с апострофом никаких проблем не возникает.

```
$ python3 shlex_example.py apostrophe.txt

ORIGINAL: "This string has an embedded apostrophe, doesn't it?"
TOKENS:
'This'
'string'
'has'
'an'
'embedded'
'apostrophe'
','
"doesn't"
'it'
'?'
```

14.6.2. Создание безопасных строк для командных оболочек

Функция `quote()` выполняет обратную операцию, экранируя существующие и добавляя отсутствующие кавычки для строк, чтобы сделать их безопасными для использования в командах оболочки.

Листинг 14.55. `shlex_quote.py`

```
import shlex

examples = [
```

```
    "Embedded'SingleQuote",
    'Embedded"DoubleQuote',
    'Embedded Space',
    '~SpecialCharacter',
    r'Back\slash',
]

for s in examples:
    print('ORIGINAL : {}'.format(s))
    print('QUOTED   : {}'.format(shlex.quote(s)))
    print()
```

Обычно при вызове конструктора `subprocess.Popen()` безопаснее передавать аргументы в виде списка. Тем не менее в ситуациях, когда это невозможно, функция `quote()` обеспечивает определенную защиту, гарантируя, что специальные и пробельные символы будут надлежащим образом заключены в кавычки.

```
$ python3 shlex_quote.py

ORIGINAL : Embedded'SingleQuote
QUOTED   : 'Embedded'"'"'SingleQuote'

ORIGINAL : Embedded"DoubleQuote
QUOTED   : 'Embedded"DoubleQuote'

ORIGINAL : Embedded Space
QUOTED   : 'Embedded Space'

ORIGINAL : ~SpecialCharacter
QUOTED   : '~SpecialCharacter'

ORIGINAL : Back\slash
QUOTED   : 'Back\slash'
```

14.6.3. Встроенные комментарии

Поскольку данный анализатор предназначен для работы с командными языками, он нуждается в обработке комментариев. По умолчанию любой текст, следующий за символом #, считается частью комментария и игнорируется. В силу самой природы анализатора поддерживаются лишь односимвольные префиксы комментариев. Набор используемых для этой цели символов можно конфигурировать с помощью свойства `commenters`.

```
$ python3 shlex_example.py comments.txt

ORIGINAL: 'This line is recognized.\n# But this line is ignored.
\nAnd this line is processed.'

TOKENS:
'This'
'line'
'is'
```

```
'recognized'
'.'
'And'
'this'
'line'
'is'
'processed'
'.'
```

14.6.4. Разбиение строк на лексемы

Функция `split()` предоставляется в качестве удобной обертки для парсера. Ее можно использовать для разбиения существующей строки на лексемы.

Листинг 14.56. `shlex_split.py`

```
import shlex

text = """This text has "quoted parts" inside it."""
print('ORIGINAL: {!r}'.format(text))
print()

print('TOKENS:')
print(shlex.split(text))
```

Результатом является список.

```
$ python3 shlex_split.py

ORIGINAL: 'This text has "quoted parts" inside it.'

TOKENS:
['This', 'text', 'has', 'quoted parts', 'inside', 'it.']
```

14.6.5. Другие источники лексем

Класс `shlex` включает несколько конфигурационных свойств, управляющих его поведением. Свойство `source` поддерживает повторное использование кода (или конфигурации), позволяя одному потоку лексем включать другой. Это средство аналогично оператору `source` командной оболочки Bourne shell, чем и объясняется его название.

Листинг 14.57. `shlex_source.py`

```
import shlex

text = "This text says to source quotes.txt before continuing."
print('ORIGINAL: {!r}'.format(text))
print()

lexer = shlex.shlex(text)
lexer.wordchars += '.'
lexer.source = 'source'
```

```
print('TOKENS:')
for token in lexer:
    print('{!r}'.format(token))
```

Строка `source quotes.txt`, содержащаяся в исходном тексте, обрабатывается специальным образом. Поскольку свойству `source` лексического анализатора присвоено значение `"source"`, то каждый раз, когда это значение встречается в тексте, следующая лексема воспринимается как имя файла, содержимое которого автоматически включается в анализируемый текст. Чтобы имя файла с расширением было воспринято как единая лексема, необходимо добавить символ `.` (точка) в список символов, включаемых в слова. В противном случае текст `quotes.txt` будет преобразован в три лексемы: `quotes`, `.` и `txt`. Выполнение этого примера дает следующий вывод.

```
$ python3 shlex_source.py

ORIGINAL: 'This text says to source quotes.txt before
continuing.'

TOKENS:
'This'
'text'
'says'
'to'
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
','
'and'
'even'
'"a \'nested example\'"'
'.'
'before'
'continuing.'
```

Для загрузки дополнительного ввода свойство `source` использует метод `sourcehook()`. Как следствие, подкласс `shlex` может предоставить альтернативную реализацию, загружающую данные из расположений, не являющихся файлами.

14.6.6. Управление анализатором

В предыдущем примере было продемонстрировано изменение значения атрибута `wordchars`, позволяющего управлять включением символов в состав слов. Точно так же с помощью атрибута `quotes` можно задать дополнительные или альтернативные кавычки. Каждая такая кавычка должна быть одиночным символом,

что делает невозможным создание открывающей и закрывающей кавычек, представленных разными символами (например, анализатор не допускает использование открывающей и закрывающей скобок в качестве пары кавычек).

Листинг 14.58. shlex_table.py

```
import shlex

text = """|Col 1||Col 2||Col 3|"""
print('ORIGINAL: {!r}'.format(text))
print()

lexer = shlex.shlex(text)
lexer.quotes = '|'

print('TOKENS:')
for token in lexer:
    print('{!r}'.format(token))
```

В этом примере каждая ячейка таблицы располагается между двумя символами вертикальной черты.

```
$ python3 shlex_table.py

ORIGINAL: '|Col 1||Col 2||Col 3|'

TOKENS:
'|Col 1|'
'|Col 2|'
'|Col 3|'
```

Также существует возможность управлять пробельными символами, используемыми для разбиения текста на слова.

Листинг 14.59. shlex_whitespace.py

```
import shlex
import sys

if len(sys.argv) != 2:
    print('Please specify one filename on the command line.')
    sys.exit(1)

filename = sys.argv[1]
with open(filename, 'r') as f:
    body = f.read()
print('ORIGINAL: {!r}'.format(body))
print()

print('TOKENS:')
lexer = shlex.shlex(body)
lexer.whitespace += '.,'
for token in lexer:
    print('{!r}'.format(token))
```

Модифицированная версия сценария shlex_example.py, в которой в число пробельных символов включены символы запятой и точки, дает другой результат.

```
$ python3 shlex_whitespace.py quotes.txt

ORIGINAL: 'This string has embedded "double quotes" and\n\'singl
e quotes\' in it, and even "a \'nested example\'".\n'

TOKENS:
'This'
'string'
'has'
'embedded'
'"double quotes"'
'and'
"'single quotes'"
'in'
'it'
'and'
'even'
'"a \'nested example\'"'
```

14.6.7. Обработка ошибок

Если анализатор достигает конца ввода до того, как будут обработаны все необходимые закрывающие кавычки строк, то возбуждается исключение ValueError. В подобных случаях целесообразно проверять некоторые свойства, поддерживаемые анализатором, по мере обработки ввода. Например, свойство infile ссылается на имя обрабатываемого файла (которое может отличаться от имени исходного файла, если один файл служит источником содержимого другого). Значение lineno указывает на номер строки, в которой возникла ошибка. Обычно этому номеру соответствует конец файла, который может располагаться далеко от первой кавычки. В атрибуте token содержится текст, который еще находится в буфере и не включен в действительную лексему. Метод error_leader() создает префикс сообщения в стиле компиляторов Unix, используя который такие редакторы, как *emacs*, могут анализировать ошибки и переходить непосредственно к некорректной строке.

Листинг 14.60. shlex_errors.py

```
import shlex

text = """This line is OK.
This line has an "unfinished quote.
This line is OK, too.
"""

print('ORIGINAL: {!r}'.format(text))
print()

lexer = shlex.shlex(text)
```

```
print('TOKENS:')
try:
    for token in lexer:
        print('{!r}'.format(token))
except ValueError as err:
    first_line_of_error = lexer.token.splitlines()[0]
    print('ERROR: {} {}'.format(lexer.error_leader(), err))
    print('following {!r}'.format(first_line_of_error))
```

Выполнение этого примера дает следующий вывод.

```
$ python3 shlex_errors.py

ORIGINAL: 'This line is OK.\nThis line has an "unfinished quote.
\nThis line is OK, too.\n'

TOKENS:
'This'
'line'
'is'
'OK'
'.'
'This'
'line'
'has'
'an'
ERROR: "None", line 4: No closing quotation
following '"unfinished quote.'
```

14.6.8. Анализ в соответствии с требованиями стандарта POSIX

Используемый по умолчанию стиль анализа обеспечивает обратную совместимость, но не соответствует стандарту POSIX. Чтобы активизировать поведение анализатора в стиле POSIX, следует передать конструктору объекта анализатора аргумент `posix` со значением `True`.

Листинг 14.61. `shlex_posix.py`

```
import shlex

examples = [
    'Do"Not"Separate',
    '"Do"Separate',
    'Escaped \e Character not in quotes',
    'Escaped "\e" Character in double quotes',
    "Escaped '\e' Character in single quotes",
    r"Escaped '\'' \"\'\" single quote",
    r'Escaped "\"" \'\"\' double quote',
    "\"'Strip extra layer of quotes'\"",
]

for s in examples:
```

```
    print('ORIGINAL : {!r}'.format(s))
    print('non-POSIX: ', end='')

    non_posix_lexer = shlex.shlex(s, posix=False)
    try:
        print('{!r}'.format(list(non_posix_lexer)))
    except ValueError as err:
        print('error({})'.format(err))

    print('POSIX    : ', end='')
    posix_lexer = shlex.shlex(s, posix=True)
    try:
        print('{!r}'.format(list(posix_lexer)))
    except ValueError as err:
        print('error({})'.format(err))

    print()
```

Приведенные ниже примеры демонстрируют влияние поведения анализатора на получаемые результаты.

```
$ python3 shlex_posix.py

ORIGINAL : 'Do"Not"Separate'
non-POSIX: ['Do"Not"Separate']
POSIX    : ['DoNotSeparate']

ORIGINAL : '"Do"Separate'
non-POSIX: ['"Do"', 'Separate']
POSIX    : ['DoSeparate']

ORIGINAL : 'Escaped \\e Character not in quotes'
non-POSIX: ['Escaped', '\\', 'e', 'Character', 'not', 'in', 'quotes']
POSIX    : ['Escaped', 'e', 'Character', 'not', 'in', 'quotes']

ORIGINAL : 'Escaped "\\e" Character in double quotes'
non-POSIX: ['Escaped', '"\\e"', 'Character', 'in', 'double', 'quotes']
POSIX    : ['Escaped', '\\e', 'Character', 'in', 'double', 'quotes']

ORIGINAL : "Escaped '\\e' Character in single quotes"
non-POSIX: ['Escaped', "'\\e'", 'Character', 'in', 'single', 'quotes']
POSIX    : ['Escaped', '\\e', 'Character', 'in', 'single', 'quotes']

ORIGINAL : 'Escaped \'\\\'\' \\"\\\'\\" single quote'
non-POSIX: error(No closing quotation)
POSIX    : ['Escaped', '\\ \\"\\"', 'single', 'quote']

ORIGINAL : 'Escaped "\\"" \\\'\\"\\\' double quote'
non-POSIX: error(No closing quotation)
POSIX    : ['Escaped', '"', '\'"\'', 'double', 'quote']

ORIGINAL : '"\'Strip extra layer of quotes\'"'
non-POSIX: ['"\'Strip extra layer of quotes\'"']
POSIX    : ["'Strip extra layer of quotes'"]
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `shlex`[14].
- `cmd` (раздел 14.5). Инструменты для создания интерактивных командных интерпретаторов.
- `argparse` (раздел 14.1). Анализ параметров командной строки.
- `subprocess` (раздел 10.1). Выполняет команды после анализа командной строки.

14.7. `configparser`: работа с конфигурационными файлами

Модуль `configparser` обеспечивает управление конфигурационными файлами, использующими форматы параметров, аналогичные формату INI-файлов Windows. Содержимое конфигурационных файлов может быть организовано в виде именованных *разделов*, каждый из которых содержит индивидуальные *параметры* вместе с их значениями, причем поддерживаются значения нескольких типов, включая целочисленные, с плавающей точкой и булевы. Значения параметров можно объединять с помощью строк форматирования Python, создавая, например, URL-адреса на основе имен хостов и номеров портов.

14.7.1. Формат конфигурационных файлов

Формат файлов, используемый модулем `configparser`, аналогичен формату, используемому в ранних версиях Windows. Он включает один или несколько разделов, каждый из которых может содержать индивидуальные параметры и их значения.

Анализатор идентифицирует разделы конфигурационного файла, находя строки, начинающиеся с символа `[` и заканчивающиеся символом `]`. Значение, заключенное в квадратные скобки, представляет имя раздела и может содержать любые символы за исключением квадратных скобок.

Каждая строка раздела содержит один параметр. Строка начинается с имени параметра, отделенного от значения двоеточием (`:`) или знаком равенства (`=`). В процессе анализа файла пробелы, окружающие разделитель, игнорируются.

Строки, начинающиеся с символа точки с запятой (`;`) или символа решетки (`#`), считаются комментариями. При доступе к конфигурационному файлу из программы они игнорируются.

Ниже приведен пример конфигурационного файла *simple.ini*, содержащего раздел `bug_tracker` с тремя параметрами: `url`, `username` и `password`.

```
# Простой пример конфигурационного файла с комментариями
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
; Не следует хранить пароли в простых текстовых
; конфигурационных файлах
password = SECRET
```

[14] https://docs.python.org/3.5/library/shlex.html

14.7.2. Чтение конфигурационных файлов

Пользователи и системные администраторы часто редактируют конфигурационные файлы с помощью обычных текстовых редакторов для настройки параметров, используемых по умолчанию, чтобы впоследствии приложение могло прочитать файл, проанализировать его и выполнить действия, зависящие от его содержимого. Для чтения конфигурационных файлов используется метод `read()` класса `ConfigParser`.

Листинг 14.62. configparser_read.py

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('simple.ini')

print(parser.get('bug_tracker', 'url'))
```

Эта программа читает файл *simple.ini*, приведенный в предыдущем разделе, и выводит значение параметра `url` из раздела `bug_tracker`.

```
$ python3 configparser_read.py

http://localhost:8080/bugs/
```

Метод `read()` может также получать список имен файлов. Просматривается каждое имя, содержащееся в списке, и если файл существует, то он открывается и читается.

Листинг 14.63. configparser_read_many.py

```
from configparser import ConfigParser
import glob

parser = ConfigParser()

candidates = ['does_not_exist.ini', 'also-does-not-exist.ini',
              'simple.ini', 'multisection.ini']

found = parser.read(candidates)

missing = set(candidates) - set(found)

print('Found config files:', sorted(found))
print('Missing files     :', sorted(missing))
```

Метод `read()` возвращает список с именами успешно загруженных файлов. Просматривая этот список, программа может обнаружить, какие конфигурационные файлы отсутствуют, и принять решение относительно того, следует ли их игнорировать или интерпретировать это как ошибку.

```
$ python3 configparser_read_many.py

Found config files: ['multisection.ini', 'simple.ini']
```

```
Missing files     : ['also-does-not-exist.ini',
'does_not_exist.ini']
```

14.7.2.1. Конфигурационные данные в кодировке Unicode

Для чтения конфигурационных файлов, содержащих данные Unicode, необходимо использовать соответствующую кодировку значений. В приведенном ниже примере файла прежнее значение пароля заменено новым, содержащим символы Unicode, и преобразуется с использованием кодировки UTF-8.

Листинг 14.64. `unicode.ini`

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = †ßëç®ê
```

В процессе открытия файла с использованием подходящего декодера данные в кодировке UTF-8 преобразуются в строки Unicode.

Листинг 14.65. `configparser_unicode.py`

```
from configparser import ConfigParser
import codecs

parser = ConfigParser()
# Открыть файл, используя корректную кодировку
parser.read('unicode.ini', encoding='utf-8')

password = parser.get('bug_tracker', 'password')

print('Password:', password.encode('utf-8'))
print('Type     :', type(password))
print('repr()   :', repr(password))
```

Метод `get()` возвращает строку Unicode. Чтобы строку можно было безопасно вывести, ее необходимо вновь преобразовать с использованием кодировки UTF-8.

```
$ python3 configparser_unicode.py
Password: b'\xc3\x9f\xc3\xa9\xc3\xa7\xc2\xae\xc3\xa9\xe2\x80\xa0
'
Type : <class 'str'>
repr() : '†ßëç®ê'
```

14.7.3. Доступ к конфигурационным параметрам

Класс `ConfigParser` включает методы, предназначенные для обработки структуры содержимого конфигурационного файла, в том числе для получения списка разделов и параметров и их значений. Ниже приведен конфигурационный файл, который содержит два раздела, соответствующих двум разным веб-службам.

```
[bug_tracker]
url = http://localhost:8080/bugs/
username = dhellmann
password = SECRET

[wiki]
url = http://localhost:8080/wiki/
username = dhellmann
password = SECRET
```

В следующем примере демонстрируется исследование конфигурационных данных с помощью методов `sections()`, `options()` и `items()`.

Листинг 14.66. `configparser_structure.py`

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('multisection.ini')

for section_name in parser.sections():
    print('Section:', section_name)
    print('  Options:', parser.options(section_name))
    for name, value in parser.items(section_name):
        print('  {} = {}'.format(name, value))
    print()
```

Каждый из методов `sections()` и `options()` возвращает список строк, тогда как метод `items()` возвращает список кортежей, содержащих пары "имя–значение".

```
Section: bug_tracker
  Options: ['url', 'username', 'password']
  url = http://localhost:8080/bugs/
  username = dhellmann
  password = SECRET

Section: wiki
  Options: ['url', 'username', 'password']
  url = http://localhost:8080/wiki/
  username = dhellmann
  password = SECRET
```

Кроме того, класс `ConfigParser` поддерживает тот же API привязок, что и тип `dict`, действуя как один общий словарь, содержащий отдельные словари для каждого раздела.

Листинг 14.67. `configparser_structure_dict.py`

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('multisection.ini')
```

```
for section_name in parser:
    print('Section:', section_name)
    section = parser[section_name]
    print('  Options:', list(section.keys()))
    for name in section:
        print('  {} = {}'.format(name, section[name]))
    print()
```

Использование API привязок для доступа к тому же конфигурационному файлу дает те же результаты.

```
Section: DEFAULT
  Options: []

Section: bug_tracker
  Options: ['url', 'username', 'password']
  url = http://localhost:8080/bugs/
  username = dhellmann
  password = SECRET

Section: wiki
  Options: ['url', 'username', 'password']
  url = http://localhost:8080/wiki/
  username = dhellmann
  password = SECRET
```

14.7.3.1. Тестирование на наличие значений

Чтобы проверить существование раздела, следует использовать метод `has_section()`, передав ему имя раздела в качестве аргумента.

Листинг 14.68. `configparser_has_section.py`

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('multisection.ini')

for candidate in ['wiki', 'bug_tracker', 'dvcs']:
    print('{:<12}: {}'.format(
        candidate, parser.has_section(candidate)))
```

Проверка существования раздела до вызова метода `get()` может предотвратить возникновение исключений, обусловленных отсутствием данных.

```
$ python3 configparser_has_section.py

wiki        : True
bug_tracker : True
dvcs        : False
```

Для проверки существования параметра внутри раздела следует использовать метод `has_option()`.

Листинг 14.69. configparser_has_option.py

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('multisection.ini')

SECTIONS = ['wiki', 'none']
OPTIONS = ['username', 'password', 'url', 'description']

for section in SECTIONS:
    has_section = parser.has_section(section)
    print('{} section exists: {}'.format(section, has_section))
    for candidate in OPTIONS:
        has_option = parser.has_option(section, candidate)
        print('{}.{:<12}  : {}'.format(
            section, candidate, has_option))
    print()
```

Если указанного раздела не существует, метод `has_option()` возвращает значение `False`.

```
$ python3 configparser_has_option.py

wiki section exists: True
wiki.username      : True
wiki.password      : True
wiki.url           : True
wiki.description   : False

none section exists: False
none.username      : False
none.password      : False
none.url           : False
none.description   : False
```

14.7.3.2. Типы значений

Имена всех разделов и параметров обрабатываются как строки, тогда как значения параметров могут быть строками, целыми числами, числами с плавающей точкой и булевыми значениями. Некоторые строковые значения могут быть использованы для представления булевых значений в конфигурационном файле; при обращении к ним они преобразуются в `True` или `False`. Следующий файл включает примеры с числовыми типами, а также примеры со всеми значениями, которые распознаются анализатором как булевы значения.

Листинг 14.70. types.ini

```
[ints]
positive = 1
negative = -5

[floats]
```

```
positive = 0.2
negative = -3.14

[booleans]
number_true = 1
number_false = 0
yn_true = yes
yn_false = no
tf_true = true
tf_false = false
onoff_true = on
onoff_false = false
```

Класс `ConfigParser` не предпринимает никаких попыток определить тип параметра. Ожидается, что приложение будет использовать корректные методы для извлечения значений желаемого типа. Метод `get()` всегда возвращает строку. Для извлечения целочисленных значений следует использовать метод `getint()`, значений с плавающей точкой — метод `getfloat()`, а булевых значений — метод `getboolean()`.

Листинг 14.71. `configparser_value_types.py`

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('types.ini')

print('Integers:')
for name in parser.options('ints'):
    string_value = parser.get('ints', name)
    value = parser.getint('ints', name)
    print('  {:<12} : {!r:<7} -> {}'.format(
        name, string_value, value))

print('\nFloats:')
for name in parser.options('floats'):
    string_value = parser.get('floats', name)
    value = parser.getfloat('floats', name)
    print('  {:<12} : {!r:<7} -> {:0.2f}'.format(
        name, string_value, value))

print('\nBooleans:')
for name in parser.options('booleans'):
    string_value = parser.get('booleans', name)
    value = parser.getboolean('booleans', name)
    print('  {:<12} : {!r:<7} -> {}'.format(
        name, string_value, value))
```

Выполнение этой программы дает следующие результаты.

```
Integers:
  positive     : '1'     -> 1
  negative     : '-5'    -> -5
```

```
Floats:
  positive     : '0.2'   -> 0.20
  negative     : '-3.14' -> -3.14

Booleans:
  number_true  : '1'     -> True
  number_false : '0'     -> False
  yn_true      : 'yes'   -> True
  yn_false     : 'no'    -> False
  tf_true      : 'true'  -> True
  tf_false     : 'false' -> False
  onoff_true   : 'on'    -> True
  onoff_false  : 'false' -> False
```

С помощью аргумента `converters` можно передать конструктору `ConfigParser` дополнительные функции преобразования. Каждый конвертер получает единственное входное значение, которое он преобразует в соответствующий возвращаемый тип.

Листинг 14.72. `configparser_custom_types.py`

```
from configparser import ConfigParser
import datetime


def parse_iso_datetime(s):
    print('parse_iso_datetime({!r})'.format(s))
    return datetime.datetime.strptime(s, '%Y-%m-%dT%H:%M:%S.%f')


parser = ConfigParser(
    converters={
        'datetime': parse_iso_datetime,
    }
)
parser.read('custom_types.ini')

string_value = parser['datetimes']['due_date']
value = parser.getdatetime('datetimes', 'due_date')
print('due_date : {!r} -> {!r}'.format(string_value, value))
```

Добавление конвертера приводит к тому, что класс `ConfigParser` автоматически создает метод для извлечения значений данного типа, используя имя типа, указанное в аргументе `converters`. В данном примере конвертер `'datetime'` добавляет новый метод `getdatetime()`.

```
$ python3 configparser_custom_types.py

parse_iso_datetime('2015-11-08T11:30:05.905898')
due_date : '2015-11-08T11:30:05.905898' -> datetime.datetime(201
5, 11, 8, 11, 30, 5, 905898)
```

Преобразующие методы можно также добавлять непосредственно в подклассы `ConfigParser`.

14.7.3.3. Параметры, используемые как флаги

Обычно анализатор требует задания явных значений для каждого параметра. Но если конструктору `ConfigParser` передан аргумент `allow_no_value` со значением `True`, то во входном файле допускается задание параметров без значений, и такие параметры можно использовать в качестве флагов.

Листинг 14.73. `configparser_allow_no_value.py`

```
import configparser

# Требуется задание значений параметров
try:
    parser = configparser.ConfigParser()
    parser.read('allow_no_value.ini')
except configparser.ParsingError as err:
    print('Could not parse:', err)

# Разрешить использование параметров без значений
print('\nTrying again with allow_no_value=True')
parser = configparser.ConfigParser(allow_no_value=True)
parser.read('allow_no_value.ini')
for flag in ['turn_feature_on', 'turn_other_feature_on']:
    print('\n', flag)
    exists = parser.has_option('flags', flag)
    print('  has_option:', exists)
    if exists:
        print('         get:', parser.get('flags', flag))
```

Если для какого-либо параметра, имеющегося во входном файле, значение не задано явно, то метод `has_option()` сообщит о том, что параметр существует, а метод `get()` вернет для такого параметра значение `None`.

```
$ python3 configparser_allow_no_value.py

Could not parse: Source contains parsing errors:
'allow_no_value.ini'
        [line  2]: 'turn_feature_on\n'

Trying again with allow_no_value=True

 turn_feature_on
  has_option: True
         get: None

 turn_other_feature_on
  has_option: False
```

14.7.3.4. Многострочные строки

Строковые значения могут занимать несколько строк, если последующие строки выделены отступом.

```
[example]
message = This is a multiline string.
  With two paragraphs.

  They are separated by a completely empty line.
```

В пределах многострочных значений, выделенных отступами, пустые строки интерпретируются как часть значения и сохраняются.

```
$ python3 configparser_multiline.py

This is a multiline string.
With two paragraphs.

They are separated by a completely empty line.
```

14.7.4. Изменение параметров

В то время как основным способом конфигурирования экземпляров `ConfigParser` является чтение параметров из файлов, также допускается добавление параметров посредством вызовов метода `add_section()` для создания нового раздела и метода `set()` для добавления или изменения параметра.

Листинг 14.74. `configparser_populate.py`

```
import configparser

parser = configparser.SafeConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://localhost:8080/bugs')
parser.set('bug_tracker', 'username', 'dhellmann')
parser.set('bug_tracker', 'password', 'secret')

for section in parser.sections():
    print(section)
    for name, value in parser.items(section):
        print('  {} = {!r}'.format(name, value))
```

Все параметры должны задаваться в виде строк, даже если они будут извлекаться в виде целочисленных значений, значений с плавающей точкой или булевых значений.

```
$ python3 configparser_populate.py

bug_tracker
  url = 'http://localhost:8080/bugs'
  username = 'dhellmann'
  password = 'secret'
```

Для удаления разделов и параметров из объектов `ConfigParser` следует использовать методы `remove_section()` и `remove_option()` соответственно.

Листинг 14.75. configparser_remove.py

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('multisection.ini')

print('Read values:\n')
for section in parser.sections():
    print(section)
    for name, value in parser.items(section):
        print('  {} = {!r}'.format(name, value))

parser.remove_option('bug_tracker', 'password')
parser.remove_section('wiki')

print('\nModified values:\n')
for section in parser.sections():
    print(section)
    for name, value in parser.items(section):
        print('  {} = {!r}'.format(name, value))
```

Одновременно с удалением раздела удаляются все содержащиеся в нем параметры.

```
Read values:

bug_tracker
  url = 'http://localhost:8080/bugs/'
  username = 'dhellmann'
  password = 'SECRET'
wiki
  url = 'http://localhost:8080/wiki/'
  username = 'dhellmann'
  password = 'SECRET'

Modified values:

bug_tracker
  url = 'http://localhost:8080/bugs/'
  username = 'dhellmann'
```

14.7.5. Сохранение конфигурационных файлов

Как только объект `ConfigParser` будет заполнен нужными данными, его можно сохранить в файле, вызвав метод `write()`. Этот подход может быть использован для создания пользовательского интерфейса, предназначенного для изменения настроек параметров без написания дополнительного кода управления файлом.

Листинг 14.76. configparser_write.py

```
import configparser
import sys
```

```
parser = configparser.ConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url', 'http://localhost:8080/bugs')
parser.set('bug_tracker', 'username', 'dhellmann')
parser.set('bug_tracker', 'password', 'secret')

parser.write(sys.stdout)
```

Метод `write()` получает в качестве аргумента файловый объект. Данные записываются в формате INI, чтобы их можно было вновь проанализировать с помощью объекта `ConfigParser`.

```
$ python3 configparser_write.py

[bug_tracker]
url = http://localhost:8080/bugs
username = dhellmann
password = secret
```

Предупреждение

После чтения, изменения и повторной записи конфигурационного файла комментарии, содержащиеся в исходном файле, не сохраняются.

14.7.6. Пути поиска параметров

Для поиска параметров объект `ConfigParser` использует многоступенчатый процесс. Еще до того, как приступить к поиску, проверяется имя раздела. Если указанного раздела не существует и именем параметра не является специальное значение `DEFAULT`, то возбуждается исключение `NoSectionError`.

1. Если имя параметра встречается в словаре `vars`, передаваемом методу `get()`, возвращается значение из этого словаря.
2. Если имя параметра встречается в указанном разделе, возвращается значение из этого раздела.
3. Если имя параметра встречается в разделе `DEFAULT`, возвращается это значение.
4. Если имя параметра встречается в словаре `defaults`, передаваемом конструктору, возвращается это значение.

Если имя не обнаружено ни в одном из этих расположений, возбуждается исключение `NoOptionError`.

Описанное поведение при выполнении процедуры поиска можно продемонстрировать на примере следующего конфигурационного файла.

```
[DEFAULT]
file-only = value from DEFAULT section
init-and-file = value from DEFAULT section
from-section = value from DEFAULT section
from-vars = value from DEFAULT section
```

```
[sect]
section-only = value from section in file
from-section = value from section in file
from-vars = value from section in file
```

Тестовая программа, приведенная в следующем листинге, включает настройки по умолчанию для параметров, не указанных в конфигурационном файле, и переопределяет некоторые значения, заданные в файле.

Листинг 14.77. configparser_defaults.py

```
import configparser

# Определить имена параметров
option_names = [
    'from-default',
    'from-section', 'section-only',
    'file-only', 'init-only', 'init-and-file',
    'from-vars',
]

# Инициализировать анализатор рядом значений по умолчанию
DEFAULTS = {
    'from-default': 'value from defaults passed to init',
    'init-only': 'value from defaults passed to init',
    'init-and-file': 'value from defaults passed to init',
    'from-section': 'value from defaults passed to init',
    'from-vars': 'value from defaults passed to init',
}
parser = configparser.ConfigParser(defaults=DEFAULTS)

print('Defaults before loading file:')
defaults = parser.defaults()
for name in option_names:
    if name in defaults:
        print('  {:<15} = {!r}'.format(name, defaults[name]))

# Загрузить конфигурационный файл
parser.read('with-defaults.ini')

print('\nDefaults after loading file:')
defaults = parser.defaults()
for name in option_names:
    if name in defaults:
        print('  {:<15} = {!r}'.format(name, defaults[name]))

# Задать ряд локальных переопределений
vars = {'from-vars': 'value from vars'}

# Отобразить значения всех параметров
print('\nOption lookup:')
for name in option_names:
    value = parser.get('sect', name, vars=vars)
    print('  {:<15} = {!r}'.format(name, value))
```

```
# Отобразить сообщения об ошибках для несуществующих параметров
print('\nError cases:')
try:
    print('No such option :', parser.get('sect', 'no-option'))
except configparser.NoOptionError as err:
    print(err)

try:
    print('No such section:', parser.get('no-sect', 'no-option'))
except configparser.NoSectionError as err:
    print(err)
```

Вывод иллюстрирует происхождение значений параметров, а также способы перекрытия существующих значений значениями по умолчанию, взятыми из различных источников.

```
$ python3 configparser_defaults.py

Defaults before loading file:
  from-default    = 'value from defaults passed to init'
  from-section    = 'value from defaults passed to init'
  init-only       = 'value from defaults passed to init'
  init-and-file   = 'value from defaults passed to init'
  from-vars       = 'value from defaults passed to init'

Defaults after loading file:
  from-default    = 'value from defaults passed to init'
  from-section    = 'value from DEFAULT section'
  file-only       = 'value from DEFAULT section'
  init-only       = 'value from defaults passed to init'
  init-and-file   = 'value from DEFAULT section'
  from-vars       = 'value from DEFAULT section'

Option lookup:
  from-default    = 'value from defaults passed to init'
  from-section    = 'value from section in file'
  section-only    = 'value from section in file'
  file-only       = 'value from DEFAULT section'
  init-only       = 'value from defaults passed to init'
  init-and-file   = 'value from DEFAULT section'
  from-vars       = 'value from vars'

Error cases:
No option 'no-option' in section: 'sect'
No section: 'no-sect'
```

14.7.7. Объединение значений с помощью интерполяции

Класс `ConfigParser` предоставляет средство под названием *интерполяция*, которое можно использовать для объединения значений. Это средство активизируется извлечением значений, содержащих стандартные строки форматирования Python. Каждый из именованных параметров, указанных в извлекаемом значе-

нии, поочередно заменяется его значением, пока не исчерпается список требуемых подстановок.

Приведенный ранее в этом разделе пример с URL-адресом можно переписать с использованием интерполяции, тем самым упрощая изменение лишь части значения. Например, следующий конфигурационный файл выделяет из URL-адреса протокол, имя хоста и номер порта в качестве отдельных параметров.

```
[bug_tracker]
protocol = http
server = localhost
port = 8080
url = %(protocol)s://%(server)s:%(port)s/bugs/
username = dhellmann
password = SECRET
```

По умолчанию интерполяция выполняется при каждом вызове `get()`. Чтобы извлечь исходное значение без использования интерполяции, следует передать конструктору значение `True` в аргументе `raw`.

Листинг 14.78. `configparser_interpolation.py`

```
import configparser

parser = configparser.ConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url',
           'http://%(server)s:%(port)s/bugs')

try:
    print(parser.get('bug_tracker', 'url'))
except configparser.InterpolationMissingOptionError as err:
    print('ERROR:', err)
```

Поскольку значение вычисляется методом `get()`, изменение одной из настроек, используемых значением `url`, изменяет возвращаемое значение.

```
$ python3 configparser_interpolation.py

Original value       : http://localhost:8080/bugs/
Altered port value   : http://localhost:9090/bugs/
Without interpolation: %(protocol)s://%(server)s:%(port)s/bugs/
```

14.7.7.1. Использование значений по умолчанию

Значения, используемые для интерполяции, не обязаны встречаться в том же разделе, что и исходный параметр. Значения по умолчанию могут смешиваться с подстановочными значениями.

```
[DEFAULT]
url = %(protocol)s://%(server)s:%(port)s/bugs/
protocol = http
```

```
server = bugs.example.com
port = 80

[bug_tracker]
server = localhost
port = 8080
username = dhellmann
password = SECRET
```

В случае использования этой конфигурации значение `url` происходит из раздела `DEFAULT`, а подстановочные значения берутся из раздела `bug_tracker` с обращением к разделу `DEFAULT` для извлечения значений, не найденных в первом расположении.

Листинг 14.79. `configparser_interpolation_defaults.py`

```
from configparser import ConfigParser

parser = ConfigParser()
parser.read('interpolation_defaults.ini')

print('URL:', parser.get('bug_tracker', 'url'))
```

Значения `hostname` и `port` происходят из раздела `bug_tracker`, а значение `protocol` — из раздела `DEFAULT`.

```
$ python3 configparser_interpolation_defaults.py

URL: http://localhost:8080/bugs/
```

14.7.7.2. Ошибки подстановки

Чтобы избежать проблем, обусловленных рекурсивными ссылками, процесс подстановки значений прекращается, как только количество шагов превысит значение `MAX_INTERPOLATION_DEPTH`.

Листинг 14.80. `configparser_interpolation_recursion.py`

```
import configparser

parser = configparser.ConfigParser()

parser.add_section('sect')
parser.set('sect', 'opt', '%(opt)s')

try:
    print(parser.get('sect', 'opt'))
except configparser.InterpolationDepthError as err:
    print('ERROR:', err)
```

При попытке выполнить слишком большое количество подстановок возбуждается исключение `InterpolationDepthError`.

```
$ python3 configparser_interpolation_recursion.py

ERROR: Recursion limit exceeded in value substitution: option 'o
pt' in section 'sect' contains an interpolation key which cannot
 be substituted in 10 steps. Raw value: '%(opt)s'
```

Отсутствие значений приводит к возбуждению исключения `MissingOption Error`.

Листинг 14.81. configparser_interpolation_error.py

```
import configparser

parser = configparser.ConfigParser()

parser.add_section('bug_tracker')
parser.set('bug_tracker', 'url',
           'http://%(server)s:%(port)s/bugs')

try:
    print(parser.get('bug_tracker', 'url'))
except configparser.InterpolationMissingOptionError as err:
    print('ERROR:', err)
```

В связи с тем что параметр `server` нигде не определен, невозможно создать `url`.

```
$ python3 configparser_interpolation_error.py

ERROR: Bad value substitution: option 'url' in section
'bug_tracker' contains an interpolation key 'server' which is
not a valid option name. Raw value:
'http://%(server)s:%(port)s/bugs'
```

14.7.7.3. Экранирование специальных символов

Поскольку символом `%` начинаются инструкции интерполяции, литеральное значение `%` должно экранироваться: `%%`.

```
[escape]
value = a literal %% must be escaped
```

Чтение этого значения не требует никаких дополнительных действий.

Листинг 14.82. configparser_escape.py

```
from configparser import ConfigParser
import os

filename = 'escape.ini'
config = ConfigParser()
config.read([filename])

value = config.get('escape', 'value')

print(value)
```

При чтении этого значения последовательность символов %% автоматически преобразуется в символ %.

```
$ python3 configparser_escape.py

a literal % must be escaped
```

14.7.7.4. Расширенная интерполяция

Класс ConfigParser поддерживает альтернативные реализации интерполяции, определяемые параметром interpolation. Объект, предоставляемый в качестве аргумента interpolation, должен реализовывать API, определяемый классом Interpolation. Например, использование интерполяции Extended Interpolation вместо BasicInterpolation, используемой по умолчанию, поддерживает другой синтаксис, в котором переменные указываются с помощью символов ${}.

Листинг 14.83. configparser_extendedinterpolation.py

```
from configparser import ConfigParser, ExtendedInterpolation

parser = ConfigParser(interpolation=ExtendedInterpolation())
parser.read('extended_interpolation.ini')

print('Original value       :', parser.get('bug_tracker', 'url'))

parser.set('intranet', 'port', '9090')
print('Altered port value   :', parser.get('bug_tracker', 'url'))

print('Without interpolation:', parser.get('bug_tracker', 'url',
                                          raw=True))
```

В случае расширенной интерполяции доступ к значениям из других разделов осуществляется за счет использования имени раздела в качестве префикса, отделяемого двоеточием (:) от имени переменной.

```
[intranet]
server = localhost
port = 8080

[bug_tracker]
url = http://${intranet:server}:${intranet:port}/bugs/
username = dhellmann
password = SECRET
```

Возможность обращаться к значениям из других разделов файла обеспечивает совместное использование иерархии значений и позволяет избавиться от размещения всех значений, заданных по умолчанию, в разделе DEFAULTS.

```
$ python3 configparser_extendedinterpolation.py

Original value       : http://localhost:8080/bugs/
```

```
Altered port value   : http://localhost:9090/bugs/
Without interpolation: http://${intranet:server}:${intranet:port
}/bugs/
```

14.7.7.5. Отключение интерполяции

Чтобы отключить интерполяцию, достаточно передать значение `None` вместо объекта `Interpolation`.

Листинг 14.84. `configparser_nointerpolation.py`

```
from configparser import ConfigParser

parser = ConfigParser(interpolation=None)
parser.read('interpolation.ini')

print('Without interpolation:', parser.get('bug_tracker', 'url'))
```

При отключенной интерполяции любой синтаксис, который мог бы быть обработан объектом интерполяции, безопасно игнорируется.

```
$ python3 configparser_nointerpolation.py

Without interpolation: %(protocol)s://%(server)s:%(port)s/bugs/
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `configparser`[15].
- `ConfigObj`[16]. Усовершенствованный анализатор конфигурационных файлов с поддержкой таких возможностей, как валидация содержимого.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `configparser` (раздел A.6.10).

14.8. `logging`: механизм структурированного журналирования

Модуль `logging` определяет стандартный API для журналирования событий, связанных с ошибками и изменением состояния, а также информационных и отладочных сообщений, поступающих из библиотек и приложений. Важным преимуществом того, что API журналирования предоставляется стандартной библиотекой, является то, что все модули Python могут участвовать в журналировании событий, поэтому журнал приложения может включать сообщения, поступающие из сторонних модулей.

14.8.1. Журналирование событий компонентов

Система журналирования состоит из четырех взаимодействующих типов объектов. Каждый модуль или приложение, которому необходимо протоколировать

[15] https://docs.python.org/3.5/library/configparser.html

[16] http://configobj.readthedocs.org/en/latest/configobj.html

некоторые события, использует экземпляр Logger для добавления информации в журналы. Его вызов создает объект LogRecord, который сохраняет информацию в памяти до тех пор, пока она не будет обработана. Экземпляр Logger может иметь ряд объектов Handler, сконфигурированных для получения и обработки записей журнала. Для преобразования записей журнала в выводимые сообщения объект Handler использует объект Formatter.

14.8.2. Отличия в журналировании событий приложений и библиотек

Средства журналирования используются как разработчиками приложений, так и авторами библиотек, однако мотивы у них могут быть разными.

Разработчики приложений конфигурируют модуль logging таким образом, чтобы сообщения направлялись в определенные каналы вывода. Например, они могут классифицировать сообщения в зависимости от уровня важности информации или целевого объекта, для которого она предназначена. Все обработчики, использующие для протоколирования журнальных сообщений файлы, расположения HTTP GET/POST, почтовые адреса, доступные через протокол SMTP, типовые сокеты и специфические для ОС механизмы журналирования, включены в модуль logging, но разработчики могут дополнительно создавать пользовательские классы для случаев, не обрабатываемых встроенными классами.

Разработчики библиотек также могут использовать журналирование для собственных целей, но применение этого модуля потребует от них приложения меньших усилий. Все, что им потребуется сделать, — это создать экземпляр Logger для каждого контекста, используя подходящее имя, а затем записывать сообщения в журнал, используя стандартные уровни важности сообщений. В связи с тем что библиотека использует API журналирования с последовательной системой именования и выбора уровня важности сообщений, приложение можно сконфигурировать для отображения или сокрытия этих сообщений в соответствии с конкретной необходимостью.

14.8.3. Запись журнала в файл

Большинство приложений конфигурируются для записи журнала в файл. Используйте функцию basicConfig() для установки обработчика по умолчанию, чтобы отладочные сообщения записывались в файл.

Листинг 14.85. `logging_file_example.py`

```
import logging

LOG_FILENAME = 'logging_example.out'
logging.basicConfig(
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)

logging.debug('This message should go to the log file')

with open(LOG_FILENAME, 'rt') as f:
```

```
    body = f.read()

print('FILE:')
print(body)
```

Если выполнить этот сценарий, то сообщение запишется в файл *logging_example.out*.

```
$ python3 logging_file_example.py

FILE:
DEBUG:root:This message should go to the log file
```

14.8.4. Циклическое создание файлов журнала

Повторное выполнение предыдущего сценария приведет к добавлению сообщений в существующий файл. Чтобы при каждом запуске программы создавался новый файл, можно указать строку `'w'` в качестве аргумента `filemode` при вызове функции `basicConfig()`. Однако более эффективное управление этим процессом обеспечивает класс `RotatingFileHandler`, который автоматически создает новый файл журнала и при этом сохраняет его предыдущую версию.

Листинг 14.86. `logging_rotatingfile_example.py`

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Установить специфический регистратор с требуемым
# порогом важности сообщений
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Добавить в регистратор обработчик протоколируемых сообщений
handler = logging.handlers.RotatingFileHandler(
    LOG_FILENAME,
    maxBytes=20,
    backupCount=5,
)
my_logger.addHandler(handler)

# Записать сообщения в журнал
for i in range(20):
    my_logger.debug('i = %d' % i)

# Вывести список созданных файлов
logfiles = glob.glob('%s*' % LOG_FILENAME)
for filename in logfiles:
    print(filename)
```

В результате создаются шесть отдельных файлов, каждый из которых содержит часть журнала истории приложения.

```
$ python3 logging_rotatingfile_example.py

logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

В этом примере самой последней версией журнала всегда является файл *logging_rotatingfile_example.out*. Каждый раз, когда достигается предельное количество журналов, определяемое аргументом `backupCount`, этот файл переименовывается путем добавления к его имени суффикса `.1`. Каждый последующий файл переименовывается путем увеличения суффикса на единицу (`.1` переходит в `.2` и т.д.), а файл с суффиксом `.5` удаляется.

Примечание

Очевидно, что установленная в данном примере длина журнала слишком мала, чтобы ее можно было использовать на практике. В реальных программах следует устанавливать для аргумента `maxBytes` более подходящие значения.

14.8.5. Уровни важности сообщений

API модуля `logging` предлагает еще одну полезную возможность – генерацию сообщений, относящихся к различным уровням важности (уровням протоколирования). Это означает, что код можно оснастить средствами отладки, но в то же время установить порог протоколирования таким, чтобы отладочные сообщения не записывались в производственных условиях. Перечень уровней протоколирования сообщений, определенных в модуле `logging`, приведен в табл. 14.2.

Журнальное сообщение отображается лишь в том случае, если обработчик и регистратор настроены для генерирования сообщений этого или более высокого уровня. Например, если сообщение имеет уровень протоколирования `CRITICAL`, а для регистратора установлен уровень `ERROR`, то сообщение будет сгенерировано (50 > 40). Если сообщение имеет уровень протоколирования `WARNING`, а регистратор настроен для генерирования сообщений уровня `ERROR`, то генерации сообщения не произойдет (30 < 40).

Таблица 14.2. Уровни важности сообщений

Уровень	Значение
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`UNSET`	0

Листинг 14.87. `logging_level_example.py`

```
import logging
import sys

LEVELS = {
    'debug': logging.DEBUG,
    'info': logging.INFO,
    'warning': logging.WARNING,
    'error': logging.ERROR,
    'critical': logging.CRITICAL,
}

if len(sys.argv) > 1:
    level_name = sys.argv[1]
    level = LEVELS.get(level_name, logging.NOTSET)
    logging.basicConfig(level=level)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical error message')
```

Запустите сценарий несколько раз с разными аргументами, такими как `debug` или `warning`, чтобы увидеть, какие сообщения отображаются при задании различных уровней протоколирования.

```
$ python3 logging_level_example.py debug

DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message

$ python3 logging_level_example.py info

INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message
```

14.8.6. Именование экземпляров регистратора

Слово `root` фигурировало во всех предыдущих регистрируемых сообщениях, поскольку код использовал корневой регистратор. Простой способ указания источника происхождения конкретного сообщения заключается в использовании отдельного объекта-регистратора для каждого модуля; сообщения, посылаемые данному регистратору, включают его имя. В следующем примере демонстрируется способ протоколирования сообщений, поступающих из различных модулей, который упрощает отслеживание источника сообщения.

Листинг 14.88. logging_modules_example.py

```
import logging

logging.basicConfig(level=logging.WARNING)

logger1 = logging.getLogger('package1.module1')
logger2 = logging.getLogger('package2.module2')

logger1.warning('This message comes from one module')
logger2.warning('This comes from another module')
```

В каждой из приведенных ниже строк вывода отображаются разные имена модулей.

```
$ python3 logging_modules_example.py

WARNING:package1.module1:This message comes from one module
WARNING:package2.module2:This comes from another module
```

14.8.7. Иерархическое дерево регистраторов сообщений

Экземпляры `Logger` конфигурируются в виде древовидной структуры на основании их имен (рис. 14.1). В типичных случаях каждое приложение или библиотека определяет базовое имя, а регистраторы индивидуальных модулей задаются как дочерние объекты. У корневого регистратора нет имени.

Древовидная структура удобна для конфигурирования системы протоколирования сообщений тем, что она избавляет от необходимости создания собственного набора обработчиков для каждого регистратора сообщений. Если регистратор не имеет собственных обработчиков, то сообщение передается для обработки его родителю. Таким образом, для большинства приложений требуется конфигурировать обработчики лишь для корневого регистратора (root), и вся протоколируемая информация будет собираться и отправляться в одно и то же место (рис. 14.2).

Кроме того, древовидная структура предоставляет возможность устанавливать разные уровни протоколирования, а также разные обработчики и форматировщики для разных частей приложения или библиотеки. Эта гибкость позволяет программисту контролировать, какие сообщения регистрируются и куда они поступают (рис. 14.3).

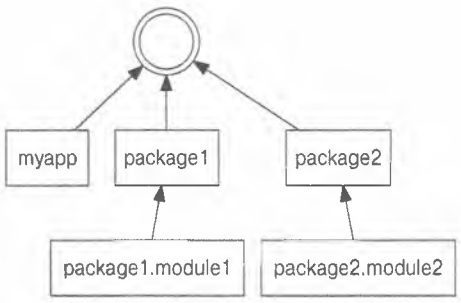

Рис. 14.1. Пример дерева регистрации сообщений

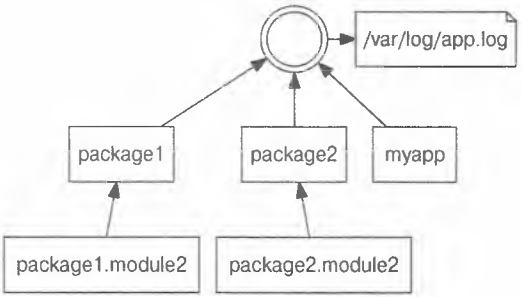

Рис. 14.2. Один обработчик регистрируемых сообщений

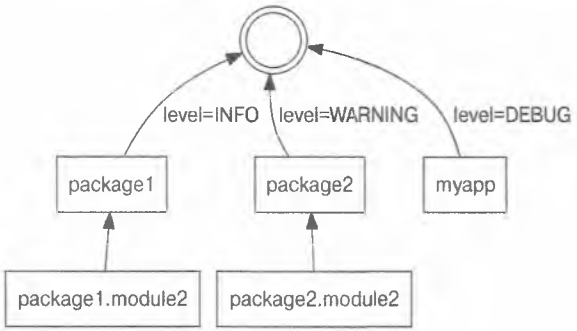

Рис. 14.3. Использование различных уровней протоколирования и обработчиков

14.8.8. Интеграция с модулем `warnings`

Модуль `logging` интегрируется с модулем `warnings` (раздел 18.1) посредством функции `captureWarnings()`, которая конфигурирует модуль `warnings` таким образом, чтобы его сообщения пропускались через систему протоколирования, а не выводились непосредственно.

Листинг 14.89. `logging_capture_warnings.py`

```
import logging
import warnings

logging.basicConfig(
    level=logging.INFO,
)

warnings.warn('This warning is not sent to the logs')

logging.captureWarnings(True)

warnings.warn('This warning is sent to the logs')
```

Предупреждающие сообщения посылаются регистратору с именем `py.warnings` с использованием уровня протоколирования `WARNING`.

```
$ python3 logging_capture_warnings.py

logging_capture_warnings.py:13: UserWarning: This warning is not
 sent to the logs
  warnings.warn('This warning is not sent to the logs')
WARNING:py.warnings:logging_capture_warnings.py:17: UserWarning:
 This warning is sent to the logs
  warnings.warn('This warning is sent to the logs')
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `logging`[17]. Документация по модулю `logging` весьма обширна; она включает практические руководства и дополнительный справочный материал, выходящий за рамки примеров, представленных в этом разделе.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `logging` (раздел A.6.25).
- `warnings` (раздел 18.1). Предупреждения о возможных нефатальных ошибках.
- `logging_tree` (Brandon Rhodes)[18]. Независимый пакет, позволяющий отображать дерево регистратора сообщений для приложения.
- `Logging Cookbook`[19]. Часть документации стандартной библиотеки с примерами использования модуля `logging` для различных задач.

14.9. `fileinput`: библиотека фильтров для утилит командной строки

Модуль `fileinput` — это библиотека, позволяющая создавать утилиты командной строки, которые действуют в качестве фильтра при обработке текстовых файлов.

14.9.1. Преобразование M3U-файлов в каналы RSS

Примером фильтра может служить программа **m3utorss**[20], предназначенная для преобразования наборов файлов формата MP3 в RSS-каналы, которые могут использоваться совместно в качестве подкастов. Программа получает входные данные из одного или нескольких файлов в формате *m3u*, содержащих списки MP3-файлов, подлежащих распространению. Результатом работы программы является список имен файлов, выводимый на консоль. В процессе итеративной обработки входного списка имен файлов программа должна выполнить следующие действия:

1) открыть каждый файл;
2) прочитать каждую строку файла;

[17] https://docs.python.org/3.5/library/logging.html
[18] https://pypi.python.org/pypi/logging_tree
[19] https://docs.python.org/3.5/howto/logging-cookbook.html
[20] https://pypi.python.org/pypi/m3utorss

3) определить, относится ли строка к MP3-файлу;

4) если это так, добавить новый элемент в канал RSS;

5) вывести результаты.

Написание кода, реализующего этот процесс, не составило бы особого труда. Он не слишком сложен, и даже умеренного его тестирования хватило бы для того, чтобы организовать корректную обработку ошибок. Однако программа значительно упростится, если использовать модуль `fileinput`, который позаботится обо всех деталях.

```
for line in fileinput.input(sys.argv[1:]):
    mp3filename = line.strip()
    if not mp3filename or mp3filename.startswith('#'):
        continue
    item = SubElement(rss, 'item')
    title = SubElement(item, 'title')
    title.text = mp3filename
    encl = SubElement(item, 'enclosure',
                      {'type': 'audio/mpeg',
                       'url': mp3filename})
```

Функция `input()` получает в качестве аргумента список проверяемых имен файлов. Если этот список пустой, модуль читает данные из стандартного ввода. Результатом выполнения функции `input()` является итератор, который возвращает отдельные строки, прочитанные из обрабатываемых файлов. Вызывающему коду остается лишь организовать цикл для просмотра каждой строки, пропуская пробелы и комментарии в процессе поиска ссылок на MP3-файлы.

Ниже приведен полный листинг этой программы.

Листинг 14.90. `fileinput_example.py`

```
import fileinput
import sys
import time
from xml.etree.ElementTree import Element, SubElement, tostring
from xml.dom import minidom

# Создать узлы rss и channel
rss = Element('rss',
              {'xmlns:dc': "http://purl.org/dc/elements/1.1/",
               'version': '2.0'})
channel = SubElement(rss, 'channel')
title = SubElement(channel, 'title')
title.text = 'Sample podcast feed'
desc = SubElement(channel, 'description')
desc.text = 'Generated for PyMOTW'
pubdate = SubElement(channel, 'pubDate')
pubdate.text = time.asctime()
gen = SubElement(channel, 'generator')
gen.text = 'https://pymotw.com/'

for line in fileinput.input(sys.argv[1:]):
```

```
    mp3filename = line.strip()
    if not mp3filename or mp3filename.startswith('#'):
        continue
    item = SubElement(rss, 'item')
    title = SubElement(item, 'title')
    title.text = mp3filename
    encl = SubElement(item, 'enclosure',
                      {'type': 'audio/mpeg',
                       'url': mp3filename})

rough_string = tostring(rss)
reparsed = minidom.parseString(rough_string)
print(reparsed.toprettyxml(indent="  "))
```

Содержимое используемого в примере пробного входного файла, содержащего имена нескольких MP3-файлов, представлено в приведенном ниже листинге.

Листинг 14.91. `sample_data.m3u`

```
# This is a sample m3u file.
episode-one.mp3
episode-two.mp3
```

Выполнение сценария `fileinput_example.py` с указанием пробного файла приводит к получению следующих XML-данных, использующих формат RSS.

```
$ python3 fileinput_example.py sample_data.m3u

<?xml version="1.0" ?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <title>Sample podcast feed</title>
    <description>Generated for PyMOTW</description>
    <pubDate>Sun Jul 10 10:45:01 2016</pubDate>
    <generator>https://pymotw.com/</generator>
  </channel>
  <item>
    <title>episode-one.mp3</title>
    <enclosure type="audio/mpeg" url="episode-one.mp3"/>
  </item>
  <item>
    <title>episode-two.mp3</title>
    <enclosure type="audio/mpeg" url="episode-two.mp3"/>
  </item>
</rss>
```

14.9.2. Метаданные хода выполнения процесса

В предыдущем примере имя файла и номер обрабатываемой строки не имели значения. Однако некоторые поисковые инструменты, такие как команда `grep` и ей подобные, могут нуждаться в такой информации. Модуль `fileinput` включает функции, обеспечивающие доступ к метаданным о текущей строке (`filename()`, `filelineno()` и `lineno()`).

Листинг 14.92. fileinput_grep.py

```
import fileinput
import re
import sys

pattern = re.compile(sys.argv[1])

for line in fileinput.input(sys.argv[2:]):
    if pattern.search(line):
        if fileinput.isstdin():
            fmt = '{lineno}:{line}'
        else:
            fmt = '{filename}:{lineno}:{line}'
        print(fmt.format(filename=fileinput.filename(),
                         lineno=fileinput.filelineno(),
                         line=line.rstrip()))
```

Для поиска вхождений строки "fileinput" в источнике данных для этого примера можно использовать базовый цикл поиска по шаблону.

```
$ python3 fileinput_grep.py fileinput *.py

fileinput_change_subnet.py:10:import fileinput
fileinput_change_subnet.py:17:for line in fileinput.input(files,
 inplace=True):
fileinput_change_subnet_noisy.py:10:import fileinput
fileinput_change_subnet_noisy.py:18:for line in fileinput.input(
files, inplace=True):
fileinput_change_subnet_noisy.py:19: if fileinput.isfirstline
():
fileinput_change_subnet_noisy.py:21: fileinput.filena
me()))
fileinput_example.py:6:"""Example for fileinput module.
fileinput_example.py:10:import fileinput
fileinput_example.py:30:for line in fileinput.input(sys.argv[1:]
):
fileinput_grep.py:10:import fileinput
fileinput_grep.py:16:for line in fileinput.input(sys.argv[2:]):
fileinput_grep.py:18: if fileinput.isstdin():
fileinput_grep.py:22: print(fmt.format(filename=fileinput
.filename(),
fileinput_grep.py:23: lineno=fileinput.f
ilelineno(),
```

Текст также можно читать из стандартного ввода.

```
$ cat *.py | python fileinput_grep.py fileinput

10:import fileinput
17:for line in fileinput.input(files, inplace=True):
29:import fileinput
37:for line in fileinput.input(files, inplace=True):
```

```
38:    if fileinput.isfirstline():
40:            fileinput.filename()))
54:"""Example for fileinput module.
58:import fileinput
78:for line in fileinput.input(sys.argv[1:]):
101:import fileinput
107:for line in fileinput.input(sys.argv[2:]):
109:        if fileinput.isstdin():
113:        print(fmt.format(filename=fileinput.filename(),
114:                         lineno=fileinput.filelineno(),
```

14.9.3. Фильтрация на месте

Другой распространенной операцией, выполняемой по отношению к файлам, является изменение содержимого существующего файла, а не создание нового с измененным содержимым. Например, на платформах Unix иногда требуется изменить содержимое файла *hosts* для обновления диапазона IP-адресов подсети.

Листинг 14.93. *etc_hosts.txt* до модификаций

```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
fe80::1%lo0     localhost
10.16.177.128  hubert hubert.hellfly.net
10.16.177.132  cubert cubert.hellfly.net
10.16.177.136  zoidberg zoidberg.hellfly.net
```

Безопасным способом автоматического внесения подобных изменений является создание нового файла на основе ввода с последующей заменой оригинала его измененной копией. Модуль `fileinput` поддерживает такой подход посредством использования параметра `inplace`.

Листинг 14.94. `fileinput_change_subnet.py`

```
import fileinput
import sys

from_base = sys.argv[1]
to_base = sys.argv[2]
files = sys.argv[3:]

for line in fileinput.input(files, inplace=True):
    line = line.rstrip().replace(from_base, to_base)
    print(line)
```

Несмотря на то что в сценарии используется функция `print()`, результаты не отображаются на экране, поскольку модуль `fileinput` перенаправляет стандартный поток вывода в изменяемый файл.

```
$ python3 fileinput_change_subnet.py 10.16 10.17 etc_hosts.txt
```

Обновленная версия файла содержит измененные IP-адреса всех серверов в сети 10.16.0.0/16.

Листинг 14.95. *etc_hosts.txt* **после модицикаций**

```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.
##
127.0.0.1       localhost
255.255.255.255 broadcasthost
::1             localhost
fe80::1%lo0     localhost
10.17.177.128  hubert hubert.hellfly.net
10.17.177.132  cubert cubert.hellfly.net
10.17.177.136  zoidberg zoidberg.hellfly.net
```

Прежде чем начнется обработка файла, создается его резервная копия с тем же именем, но с добавленным расширением *.bak*.

Листинг 14.96. `fileinput_change_subnet_noisy.py`

```
import fileinput
import glob
import sys

from_base = sys.argv[1]
to_base = sys.argv[2]
files = sys.argv[3:]

for line in fileinput.input(files, inplace=True):
    if fileinput.isfirstline():
        sys.stderr.write('Started processing {}\n'.format(
            fileinput.filename()))
        sys.stderr.write('Directory contains: {}\n'.format(
            glob.glob('etc_hosts.txt*')))
    line = line.rstrip().replace(from_base, to_base)
    print(line)

sys.stderr.write('Finished processing\n')
sys.stderr.write('Directory contains: {}\n'.format(
    glob.glob('etc_hosts.txt*')))
```

Резервный файл удаляется сразу же после закрытия входного файла.

```
$ python3 fileinput_change_subnet_noisy.py 10.16. 10.17.
etc_hosts.txt
Started processing etc_hosts.txt
Directory contains: ['etc_hosts.txt', 'etc_hosts.txt.bak']
Finished processing
Directory contains: ['etc_hosts.txt']
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `fileinput`[21].
- `m3utorss`[22]. Сценарий для преобразования M3u-файлов, содержащих списки файлов MP3, в файлы RSS, пригодные для использования в качестве каналов подкастов.
- `xml.etree`. Более подробное описание использования класса `ElementTree` для получения XML-файлов.

14.10. `atexit`: вызов функций интерпретатором при завершении работы программы

Модуль `atexit` предоставляет интерфейс для регистрации функций завершения, которые должны вызываться интерпретатором перед нормальным завершением работы программы.

14.10.1. Регистрация функций завершения

В следующем примере выполняется явная регистрация функции обратного вызова с помощью функции `register()`.

Листинг 14.97. `atexit_simple.py`

```
import atexit


def all_done():
    print('all_done()')

print('Registering')
atexit.register(all_done)
print('Registered')
```

Поскольку программа не выполняет никаких других действий, функция `all_done()` вызывается немедленно.

```
$ python3 atexit_simple.py

Registering
Registered
all_done()
```

[21] https://docs.python.org/3.5/library/fileinput.html

[22] https://pypi.python.org/pypi/m3utorss

Также допускается регистрация нескольких функций с передачей им аргументов. Эту возможность удобно использовать, в частности, для отключения от баз данных или удаления временных файлов при завершении работы программы. Вместо того чтобы поддерживать список ресурсов, которые требуется освобождать, можно зарегистрировать для каждого ресурса соответствующую функцию очистки.

Листинг 14.98. atexit_multiple.py

```
import atexit


def my_cleanup(name):
    print('my_cleanup({})'.format(name))

atexit.register(my_cleanup, 'first')
atexit.register(my_cleanup, 'second')
atexit.register(my_cleanup, 'third')
```

Очередность выполнения функций очистки обратна очередности их регистрации. Благодаря этому операции по освобождению ресурсов, связанных с отдельными модулями, могут выполняться в порядке, обратном порядку импорта модулей (а значит, и вызова их функций atexit), что уменьшает вероятность возникновения конфликтов между зависимостями.

```
$ python3 atexit_multiple.py

my_cleanup(third)
my_cleanup(second)
my_cleanup(first)
```

14.10.2. Синтаксис декораторов

Функции, не требующие задания аргументов, можно регистрировать, используя функцию register() в качестве декоратора. Этот альтернативный синтаксис удобно применять в случае функций очистки ресурсов, воздействующих на глобальные данные уровня модуля.

Листинг 14.99. atexit_decorator.py

```
import atexit

@atexit.register
def all_done():
    print('all_done()')

print('starting main program')
```

Поскольку подобные функция регистрируется во время их определения, важно быть уверенным в том, что они работают корректно, даже если модуль не выполняет никакой другой работы. Если освобождаемые ресурсы не инициализировались, то вызов функции обратного вызова, запускаемой при выходе из программы, не должен приводить к ошибке.

```
$ python3 atexit_decorator.py

starting main program
all_done()
```

14.10.3. Отмена регистрации функций обратного вызова

Чтобы отменить регистрацию функций обратного вызова, которые должны вызываться при выходе из программы, следует использовать функцию `unregister()`.

Листинг 14.100. `atexit_unregister.py`

```
import atexit


def my_cleanup(name):
    print('my_cleanup({})'.format(name))

atexit.register(my_cleanup, 'first')
atexit.register(my_cleanup, 'second')
atexit.register(my_cleanup, 'third')

atexit.unregister(my_cleanup)
```

При этом отменяются все вызовы одной и той же функции, независимо от того, сколько раз она была зарегистрирована.

```
$ python3 atexit_unregister.py
```

Отмена регистрации функции обратного вызова, которая не была зарегистрирована, не считается ошибкой.

Листинг 14.101. `atexit_unregister_not_registered.py`

```
import atexit


def my_cleanup(name):
    print('my_cleanup({})'.format(name))

if False:
    atexit.register(my_cleanup, 'never registered')

atexit.unregister(my_cleanup)
```

Поскольку функция `unregister()` игнорирует неизвестные функции обратного вызова, ее можно использовать даже в тех случаях, когда последовательность регистрации этих функций неизвестна.

```
$ python3 atexit_unregister_not_registered.py
```

14.10.4. Случаи, когда функции обратного вызова модуля atexit не вызываются

Функции обратного вызова, зарегистрированные с помощью модуля atexit, не вызываются в случае наступления следующих событий:

- выполнение программы преждевременно прекращается в результате поступления сигнала;
- явно вызвана функция os._exit();
- возникла фатальная ошибка в интерпретаторе.

Чтобы продемонстрировать, что именно происходит в случае преждевременного прекращения выполнения программы в результате поступления сигнала, можно обновить пример, который использовался при описании модуля subprocess (раздел 10.1). В этом примере участвуют два файла, содержащие родительскую и дочернюю программы соответственно. Родительская программа запускает дочернюю, затем выдерживает паузу, после чего прекращает выполнение дочерней программы.

Листинг 14.102. atexit_signal_parent.py

```
import os
import signal
import subprocess
import time

proc = subprocess.Popen('./atexit_signal_child.py')
print('PARENT: Pausing before sending signal...')
time.sleep(1)
print('PARENT: Signaling child')
os.kill(proc.pid, signal.SIGTERM)
```

Дочерняя программа устанавливает функцию обратного вызова, срабатывающую при нормальном завершении программы, а затем "засыпает" до получения сигнала.

Листинг 14.103. atexit_signal_child.py

```
import atexit
import time
import sys


def not_called():
    print('CHILD: atexit handler should not have been called')

print('CHILD: Registering atexit handler')
sys.stdout.flush()
atexit.register(not_called)

print('CHILD: Pausing to wait for signal')
sys.stdout.flush()
time.sleep(5)
```

Выполнение этого сценария дает следующие результаты.

```
$ python3 atexit_signal_parent.py

CHILD: Registering atexit handler
CHILD: Pausing to wait for signal
PARENT: Pausing before sending signal...
PARENT: Signaling child
```

Дочерняя программа не выводит сообщение, встроенное в функцию not_called().

Используя вызов os._exit(), программист может избежать вызова функции обратного вызова при завершении работы программы.

Листинг 14.104. atexit_os_exit.py

```
import atexit
import os


def not_called():
    print('This should not be called')

print('Registering')
atexit.register(not_called)
print('Registered')

print('Exiting...')
os._exit(0)
```

Поскольку в данном примере не используется обычный путь выхода из программы, интерпретатор не вызывает функцию завершения. Кроме того, функция print в ней не будет выполнена, и сброс буфера для нее не потребуется, поэтому пример выполняется без использования параметра командной строки -u, активизирующего буферизованный ввод-вывод.

```
$ python3 -u atexit_os_exit.py

Registering
Registered
Exiting...
```

Чтобы гарантировать выполнение функций обратного вызова, следует предоставить программе возможность выполнить все предусмотренные в ней инструкции или вызвать функцию sys.exit().

Листинг 14.105. atexit_sys_exit.py

```
import atexit
import sys

def all_done():
   print('all_done()')
```

```
print('Registering')
atexit.register(all_done)
print('Registered')

print('Exiting...')
sys.exit()
```

В этом примере вызывается функция `sys.exit()`, которая обеспечивает выполнение зарегистрированных функций обратного вызова при выходе из программы.

```
$ python3 atexit_sys_exit.py

Registering
Registered
Exiting...
all_done()
```

14.10.5. Обработка исключений

Трассировочная информация об исключениях, возбужденных в функциях обратного вызова модуля `atexit`, выводится на консоль. Последнее из исключений возбуждается повторно и используется в качестве итогового сообщения об ошибке, возникшей в программе.

Листинг 14.106. `atexit_exception.py`

```
import atexit


def exit_with_exception(message):
  raise RuntimeError(message)

atexit.register(exit_with_exception, 'Registered first')
atexit.register(exit_with_exception, 'Registered second')
```

Порядок регистрации функций обратного вызова определяет порядок их выполнения. Если ошибка в одной из этих функций вызывает ошибку в другой (зарегистрированной ранее, но выполненной позже), то итоговое сообщение об ошибке может не быть наиболее полезным для отображения пользователю.

```
$ python3 atexit_exception.py

Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "atexit_exception.py", line 11, in exit_with_exception
    raise RuntimeError(message)
RuntimeError: Registered second
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "atexit_exception.py", line 11, in exit_with_exception
    raise RuntimeError(message)
RuntimeError: Registered first
```

Как правило, лучше всего обрабатывать и протоколировать все исключения, возникающие в функциях очистки, без вывода соответствующих сообщений на консоль: отображение ошибок после аварийного завершения работы программы придает результирующему выводу неаккуратный вид.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `atexit`[23].
- Раздел 17.2.4. Глобальная обработка неперехваченных исключений.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `atexit` (раздел A.6.5).

14.11. sched: планирование запуска событий

Модуль `sched` реализует обобщенный планировщик событий, обеспечивающий запуск задач в определенное время. Класс `scheduler` использует функцию `time()` для определения текущего времени и функцию `delay()` для задания паузы определенной длительности. Фактические единицы времени неважны, поэтому интерфейс достаточно гибок для того, чтобы его можно было использовать для многих целей.

Функция `time()` вызывается без аргументов и должна возвращать число, представляющее текущее время. Функция `delay()` вызывается с единственным целочисленным аргументом, используя ту же шкалу, что и функция `time()`, и должна выждать в течение заданного промежутка времени, прежде чем вернуть результат. По умолчанию используются функции `monotonic()` и `sleep()` из модуля `time` (раздел 4.1), но в приведенных в этом разделе примерах используется функция `time.time()`, которая также удовлетворяет необходимым требованиям, поскольку упрощает чтение вывода.

Для поддержки многопоточных приложений функция `delay()` должна вызываться с аргументом `0` после генерации каждого события, чтобы гарантировать возможность запуска каждого потока.

14.11.1. Запуск событий с задержкой

Можно планировать запуск событий с задержкой или на определенное время. Чтобы запланировать запуск события с задержкой, следует использовать метод `enter()`, который имеет четыре аргумента:

- число, представляющее длительность задержки;
- приоритет;
- вызываемая функция;
- кортеж аргументов, передаваемых функции.

В этом примере запуск двух различных событий назначается через 2 и 3 секунды соответственно. Когда наступает время запуска события, вызывается функция `print_event()`, которая выводит текущее время и имя аргумента, переданного событию.

[23] `https://docs.python.org/3.5/library/atexit.html`

Листинг 14.107. sched_basic.py

```
import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)


def print_event(name, start):
    now = time.time()
    elapsed = int(now - start)
    print('EVENT: {} elapsed={} name={}'.format(
        time.ctime(now), elapsed, name))

start = time.time()
print('START:', time.ctime(start))
scheduler.enter(2, 1, print_event, ('first', start))
scheduler.enter(3, 1, print_event, ('second', start))

scheduler.run()
```

Выполнение этой программы дает следующие результаты.

```
$ python3 sched_basic.py

START: Sun Sep 4 16:21:01 2016
EVENT: Sun Sep 4 16:21:03 2016 elapsed=2 name=first
EVENT: Sun Sep 4 16:21:04 2016 elapsed=3 name=second
```

Время, выводимое для первого события, составляет 2 секунды, а время, выведенное для второго события, — 3 секунды после запуска программы.

14.11.2. Перекрывающиеся события

Вызов метода `run()` блокируется до тех пор, пока не будут обработаны все события. Каждое событие выполняется в одном и том же потоке, и поэтому в тех случаях, когда выполнение события занимает больше времени, чем промежуток между событиями, наступает перекрытие событий. Это перекрытие разрешается переносом последующих событий на более позднее время. Таким образом, события не теряются, однако некоторые из них могут быть вызваны позже запланированного момента времени. В следующем примере функция `long_event()` просто засыпает на 2 секунды, но точно так же эта задержка могла бы быть обусловлена выполнением интенсивных вычислений или блокирующими операциями ввода-вывода.

Листинг 14.108. sched_overlap.py

```
import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)
```

```
def long_event(name):
    print('BEGIN EVENT :', time.ctime(time.time()), name)
    time.sleep(2)
    print('FINISH EVENT:', time.ctime(time.time()), name)

print('START:', time.ctime(time.time()))
scheduler.enter(2, 1, long_event, ('first',))
scheduler.enter(3, 1, long_event, ('second',))

scheduler.run()
```

В результате этого второе событие запускается сразу же после завершения первого, поскольку прохождение первого события занимает достаточно много времени для того, чтобы перенести время запуска второго события.

```
$ python3 sched_overlap.py

START: Sun Sep 4 16:21:04 2016
BEGIN EVENT : Sun Sep 4 16:21:06 2016 first
FINISH EVENT: Sun Sep 4 16:21:08 2016 first
BEGIN EVENT : Sun Sep 4 16:21:08 2016 second
FINISH EVENT: Sun Sep 4 16:21:10 2016 second
```

14.11.3. Приоритеты событий

Если несколько событий запланированы на одно и то же время, то для определения очередности их запуска используются значения приоритета.

Листинг 14.109. sched_priority.py

```
import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)


def print_event(name):
    print('EVENT:', time.ctime(time.time()), name)

now = time.time()
print('START:', time.ctime(now))
scheduler.enterabs(now + 2, 2, print_event, ('first',))
scheduler.enterabs(now + 2, 1, print_event, ('second',))

scheduler.run()
```

В этом примере необходимо обеспечить запуск событий строго в одно и то же время, поэтому вместо метода `enter()` используется метод `enterabs()`, аргументом которого является время, а не задержка времени запуска события.

```
$ python3 sched_priority.py

START: Sun Sep 4 16:21:10 2016
```

```
EVENT: Sun Sep 4 16:21:12 2016 second
EVENT: Sun Sep 4 16:21:12 2016 first
```

14.11.4. Отмена событий

Обе функции, `enter()` и `enterabs()`, возвращают ссылку на событие, которая впоследствии может быть использована для его отмены. Поскольку метод `run()` блокируется, для отмены события необходимо использовать другой поток. В данном примере планировщик запускается в дополнительном потоке, а основной поток используется для отмены события.

Листинг 14.110. `sched_cancel.py`

```
import sched
import threading
import time

scheduler = sched.scheduler(time.time, time.sleep)

# Задать глобальную переменную, значение которой будет
# изменяться потоками
counter = 0

def increment_counter(name):
    global counter
    print('EVENT:', time.ctime(time.time()), name)
    counter += 1
    print('NOW:', counter)

print('START:', time.ctime(time.time()))
e1 = scheduler.enter(2, 1, increment_counter, ('E1',))
e2 = scheduler.enter(3, 1, increment_counter, ('E2',))

# Запустить поток для выполнения событий
t = threading.Thread(target=scheduler.run)
t.start()

# Отменить в основном потоке первое из запланированных событий
scheduler.cancel(e1)

# Выждать, пока не завершится выполнение планировщика в потоке
t.join()
print('FINAL:', counter)
```

Были запланированы два события, однако впоследствии первое из них было отменено. Выполняется лишь второе событие, поэтому переменная `counter` инкрементируется только один раз.

```
$ python3 sched_cancel.py
START: Sun Sep 4 16:21:13 2016
EVENT: Sun Sep 4 16:21:16 2016 E2
```

```
NOW: 1
FINAL: 1
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `sched`[24].
- `time` (раздел 4.1). Описание модуля `time`.

[24] https://docs.python.org/3.5/library/sched.html

Глава 15

Интернационализация и локализация приложений

Python поставляется с двумя модулями, позволяющими подготовить приложение к использованию нескольких языков и региональных настроек. Модуль `gettext` (раздел 15.1) обеспечивает отображение командных подсказок и сообщений об ошибках на языке, понятном пользователю, за счет создания каталогов с сообщениями, переведенными на различные языки. Модуль `locale` (раздел 15.2) изменяет форматирование чисел, денежных единиц, значений даты и времени с учетом таких региональных стандартов, как обозначение отрицательных значений или символ местной денежной единицы. Оба модуля взаимодействуют с другими инструментами и рабочей средой, обеспечивая согласованность приложения на языке Python с остальными программами, установленными в данной системе.

15.1. `gettext`: каталоги сообщений

Модуль `gettext` предоставляет совместимую с библиотекой GNU gettext реализацию, распознающую только код на языке Python, с помощью которой можно управлять сообщениями, переведенными на другие языки, и каталогами, в которых они содержатся. Инструменты, доступные в исходном дистрибутиве Python, позволяют извлекать сообщения из набора исходных файлов, создавать каталоги, содержащие переводы сообщений, и использовать их на этапе выполнения для отображения информации на языке, понятном пользователю.

Каталоги сообщений можно также использовать для настройки других параметров, в том числе для создания интерфейсов-оберток.

Примечание

Несмотря на то что в документации стандартной библиотеки утверждается, что Python включает все необходимые инструменты, программа `pygettext.py` неспособна извлекать сообщения, обернутые вызовом функции `ngettext()`, даже с использованием соответствующих параметров командной строки. В приведенных в этом разделе примерах вместо программы `pygettext.py` используется утилита `xgettext` из набора инструментов GNU gettext.

15.1.1. Обзор процесса перевода сообщений

Процесс настройки и использования сообщений, переведенных на различные языки, включает следующие пять этапов.

1. *Идентификация и маркирование в исходном коде тех литеральных строк, которые содержат сообщения, подлежащие переводу на другие языки.*

Начните с идентификации в исходном коде сообщений, которые должны быть переведены на другой язык, и пометьте соответствующие литеральные строки, чтобы программа, предназначенная для их извлечения, могла их найти.

2. *Извлечение сообщений.*

   Идентифицировав подлежащие переводу строки, используйте программу `xgettext` для их извлечения и создайте *шаблон перевода* в виде *.pot*-файла. Шаблон перевода — это текстовый файл, содержащий копии всех идентифицированных строк и поля, в которые должны быть вставлены переводы сообщений.

3. *Перевод сообщений*

   Передайте переводчику копию *.pot*-файла, изменив его расширение на *.po*. Файл с расширением *.po* — это редактируемый файл, используемый в качестве входного на этапе компиляции. Переводчик должен обновить текст заголовка в файле и предоставить перевод для каждой строки.

4. *Компиляция каталога переведенных сообщений.*

   Получив от переводчика готовый текстовый *.po*-файл, скомпилируйте его в двоичный формат каталога с помощью функции `msgfmt`. Двоичный формат используется при поиске подстановочных значений в каталоге во время выполнения.

5. *Загрузка и активизация подходящего каталога сообщений во время выполнения.*

   Окончательным шагом является добавление в программу нескольких строк, обеспечивающих конфигурирование и загрузку каталога сообщений, а также настройку функции `translation()`. Это можно сделать несколькими способами, однако все они требуют компромиссных решений.

В оставшейся части раздела мы подробно рассмотрим каждый из этих этапов по отдельности, начав с внесения в код необходимых изменений.

15.1.2. Создание каталога сообщений на основе исходного кода

Работа модуля `gettext` заключается в поиске литеральных строк, содержащихся в базе данных, и извлечении соответствующего перевода для каждой строки. Обычным приемом является связывание поисковой функции с именем `_` (одиночный символ подчеркивания), позволяющим избавиться от загромождения кода множеством вызовов функций с длинными именами.

Программа для извлечения сообщений, `xgettext`, ищет сообщения, которые встроены в вызовы функций, выполняющих поиск в каталоге. Она распознает различные исходные языки и использует для каждого из них соответствующий парсер. При наличии поисковых функций, имеющих псевдонимы (а также в случае добавления новых функций), необходимо передать программе `xgettext` имена дополнительных символов, которые следует учитывать при извлечении сообщений.

В представленном ниже сценарии имеется одно сообщение, подготовленное к переводу.

Листинг 15.1. `gettext_example.py`

```
t = gettext.translation(
    'example_domain', 'locale',
    fallback=True,
)
_ = t.gettext

print(_('This message is in the script.'))
```

Текст `"This message is in the script."` — это сообщение, которое должно быть заменено соответствующим переводом из каталога сообщений. Включение резервного режима (`fallback`) обеспечивает вывод встроенного сообщения, если при выполнении сценария каталог сообщений окажется недоступным.

```
$ python3 gettext_example.py

This message is in the script.
```

Следующим шагом является извлечение сообщения и создание *.pot*-файла с помощью программы `pygettext.py` или `xgettext`.

```
$ xgettext -o example.pot gettext_example.py
```

Содержимое результирующего файла приведено ниже.

Листинг 15.2. `example.pot`

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2016-07-10 10:45-0400\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: gettext_example.py:19
msgid "This message is in the script."
msgstr ""
```

Каталоги сообщений создаются в каталогах, структурированных по признакам *домена* и *языка*. Домен предоставляется приложением или библиотекой, и обычно ему соответствует какое-либо уникальное значение, например имя приложения. В сценарии `gettext_example.py` домен определяется значением `example_domain`. Значение параметра, определяющего язык, предоставляется пользовательской средой времени выполнения через одну из переменных среды `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` или `LANG`, в зависимости от конфигурации и платформы. Все примеры, приведенные в этом разделе, выполнялись с использованием значения `en_US` для параметра языка.

Следующим после подготовки шаблона шагом является создание требуемой структуры каталогов и копирование шаблона в соответствующее расположение. Во всех приведенных ниже примерах в качестве корневого каталога сообщений используется каталог `locale`, принадлежащий дереву каталогов PyMOTW, но, как правило, для этих целей лучше использовать каталог, к которому можно обращаться из любого расположения в системе, чтобы доступ к нему имели все пользователи. Полный путь к исходному файлу каталога сообщений имеет следующий вид: `$localedir/$language/LC_MESSAGES/$domain.po`, а фактическим каталогом сообщений является файл с расширением *.mo*.

Чтобы создать каталог, следует скопировать файл шаблона `example.pot` в расположение `locale/en_US/LC_MESSAGES/example.po` и внести в него соответствующие исправления, изменив значения в заголовке и задав альтернативные варианты сообщений (листинг 15.3).

Листинг 15.3. `locale/en_US/LC_MESSAGES/example.po`

```
# Messages from gettext_example.py.
# Copyright (C) 2009 Doug Hellmann
# Doug Hellmann <doug@doughellmann.com>, 2016.
#
msgid ""
msgstr ""
"Project-Id-Version: PyMOTW-3\n"
"Report-Msgid-Bugs-To: Doug Hellmann <doug@doughellmann.com>\n"
"POT-Creation-Date: 2016-01-24 13:04-0500\n"
"PO-Revision-Date: 2016-01-24 13:04-0500\n"
"Last-Translator: Doug Hellmann <doug@doughellmann.com>\n"
"Language-Team: US English <doug@doughellmann.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: gettext_example.py:16
msgid "This message is in the script."
msgstr "This message is in the en_US catalog."
```

Каталог создается на основе *.po*-файла с помощью утилиты `msgfmt`:

```
$ cd locale/en_US/LC_MESSAGES; msgfmt -o example.mo example.po
```

В качестве домена в сценарии gettext_example.py используется example_domain, однако файл называется example.pot. Чтобы модуль gettext мог найти соответствующий файл с переводом сообщения, эти имена должны совпадать.

Листинг 15.4. gettext_example_corrected.py

```
t = gettext.translation(
    'example', 'locale',
    fallback=True,
)
```

Теперь при запуске сценария будет выводиться не встроенная строка, а сообщение из каталога.

```
$ python3 gettext_example_corrected.py

This message is in the en_US catalog.
```

15.1.3. Поиск каталогов сообщений во время выполнения

Как уже отмечалось, *каталог локалей*, содержащий каталоги сообщений, организуется на основе поддерживаемых языков, тогда как имена каталогов сообщений формируются на основе *доменов программ*. Различные операционные системы определяют собственные значения по умолчанию, но модулю gettext ничего неизвестно обо всех этих умолчаниях. Он использует заданный по умолчанию каталог локалей sys.prefix + '/share/locale', но в большинстве случаев безопаснее явно задавать значение каталога locale, чем надеяться на то, что значение по умолчанию всегда является корректным. Функция find() позволяет находить нужный каталог сообщений во время выполнения.

Листинг 15.5. gettext_find.py

```
import gettext

catalogs = gettext.find('example', 'locale', all=True)
print('Catalogs:', catalogs)
```

Часть пути, ассоциированная с языком, берется из одной из нескольких переменных среды, пригодных для конфигурирования средств локализации (LANGUAGE, LC_ALL, LC_MESSAGES и LANG). Фактически используется первая найденная переменная из числа указанных. Можно выбрать несколько языков, разделив идентифицирующие их значения символами двоеточия (:). Чтобы продемонстрировать, как это работает, ниже в качестве примера приведены результаты нескольких запусков сценария gettext_find.py.

```
$ cd locale/en_CA/LC_MESSAGES; msgfmt -o example.mo example.po
$ cd ../../..
$ python3 gettext_find.py

Catalogs: ['locale/en_US/LC_MESSAGES/example.mo']

$ LANGUAGE=en_CA python3 gettext_find.py
```

```
Catalogs: ['locale/en_CA/LC_MESSAGES/example.mo']

$ LANGUAGE=en_CA:en_US python3 gettext_find.py

Catalogs: ['locale/en_CA/LC_MESSAGES/example.mo',
'locale/en_US/LC_MESSAGES/example.mo']

$ LANGUAGE=en_US:en_CA python3 gettext_find.py

Catalogs: ['locale/en_US/LC_MESSAGES/example.mo',
'locale/en_CA/LC_MESSAGES/example.mo']
```

Несмотря на то что функция `find()` отображает полный список каталогов, лишь первый из них в этой последовательности фактически загружается для поиска сообщений.

```
$ python3 gettext_example_corrected.py

This message is in the en_US catalog.

$ LANGUAGE=en_CA python3 gettext_example_corrected.py

This message is in the en_CA catalog.

$ LANGUAGE=en_CA:en_US python3 gettext_example_corrected.py

This message is in the en_CA catalog.

$ LANGUAGE=en_US:en_CA python3 gettext_example_corrected.py

This message is in the en_US catalog.
```

15.1.4. Грамматические формы для множественного числа

В то время как простая подстановка сообщений позволяет справиться с большинством проблем их перевода, встречающиеся в сообщениях формы множественного числа имен существительных обрабатывается модулем `gettext` как специальные случаи. В зависимости от языка различия между формами единственного и множественного числа могут требовать изменения не только окончаний отдельных членов предложения, но и всей структуры сообщения. Кроме того, формы множественного числа в сообщениях также могут зависеть от включаемой в сообщение величины числовой характеристики. Чтобы упростить (а в некоторых случаях сделать вообще возможным) управление множественными формами сообщений, для их запроса предоставляется отдельный набор функций.

Листинг 15.6. `gettext_plural.py`

```
from gettext import translation
import sys

t = translation('plural', 'locale', fallback=False)
num = int(sys.argv[1])
```

```
msg = t.ngettext('{num} means singular.',
                 '{num} means plural.', num)

# Все еще требуется самостоятельно добавлять значения в сообщение
print(msg.format(num=num))
```

Для доступа к нескольким подстановочным сообщениям следует использовать функцию `ngettext()`, которая получает в качестве аргументов сообщения, требующие перевода, и счетчик элементов.

```
$ xgettext -L Python -o plural.pot gettext_plural.py
```

Ввиду существования альтернативных форм сообщения, требующих перевода, подстановочные значения возвращаются в виде массива. Использование массива облегчает перевод в случае языков с несколькими формами множественного числа (например, в польском языке существуют различные формы для указания относительного количества).

Листинг 15.7. `plural.pot`

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2016-07-10 10:45-0400\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"

#: gettext_plural.py:15
#, python-brace-format
msgid "{num} means singular."
msgid_plural "{num} means plural."
msgstr[0] ""
msgstr[1] ""
```

Кроме предоставления строк перевода необходимо уведомить библиотеку о способе формирования грамматических форм множественного числа, чтобы ей было известно, как индексировать массив для любого заданного значения счетчика. Строка `"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"` включает два значения, которые должны быть заменены вручную: `nplurals` — целое

число, указывающее на размер массива (количество используемых переводов), и plural — выражение на языке C, преобразующее указанное количество в индекс массива при поиске перевода. Литеральная строка n заменяется количественной величиной, переданной функции ungettext().

Можно привести пример из английского языка, включающий две формы числа. Количество 0 трактуется как множественное ("0 bananas"). Строка Plural-Forms в этом случае приобретает следующий вид:

```
Plural-Forms: nplurals=2; plural=n != 1;
```

Перевод, соответствующий единственному числу, должен помещаться в позицию 0, а перевод, соответствующий множественному числу, — в позицию 1.

Листинг 15.8. locale/en_US/LC_MESSAGES/plural.po

```
# Messages from gettext_plural.py
# Copyright (C) 2009 Doug Hellmann
# This file is distributed under the same license
# as the PyMOTW package.
# Doug Hellmann <doug@doughellmann.com>, 2016.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PyMOTW-3\n"
"Report-Msgid-Bugs-To: Doug Hellmann <doug@doughellmann.com>\n"
"POT-Creation-Date: 2016-01-24 13:04-0500\n"
"PO-Revision-Date: 2016-01-24 13:04-0500\n"
"Last-Translator: Doug Hellmann <doug@doughellmann.com>\n"
"Language-Team: en_US <doug@doughellmann.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=2; plural=n != 1;"

#: gettext_plural.py:15
#, python-format
msgid "{num} means singular."
msgid_plural "{num} means plural."
msgstr[0] "In en_US, {num} is singular."
msgstr[1] "In en_US, {num} is plural."
```

Предварительно скомпилировав каталог, выполним тестовый сценарий несколько раз, чтобы продемонстрировать, как различные значения N преобразуются в индексы строк перевода.

```
$ cd locale/en_US/LC_MESSAGES/; msgfmt -o plural.mo plural.po
$ cd ../../..
$ python3 gettext_plural.py 0

In en_US, 0 is plural.

$ python3 gettext_plural.py 1
```

```
In en_US, 1 is singular.

$ python3 gettext_plural.py 2

In en_US, 2 is plural.
```

15.1.5. Локализация приложений и модулей

Цели, которые преследуются при переводе текста, определяют способ установки и использования модуля gettext в теле кода.

15.1.5.1. Локализация приложения

В случае переводов, действие которых распространяется на все приложение, автор программы может прибегнуть к глобальной установке такой, например, функции, как ngettext(), используя пространство имен __builtins__, поскольку он самостоятельно контролирует высокоуровневый код и ему понятен весь набор соответствующих требований, которые должны быть удовлетворены.

Листинг 15.9. gettext_app_builtin.py

```
import gettext

gettext.install(
    'example',
    'locale',
    names=['ngettext'],
)
print(_('This message is in the script.'))
```

Функция install() связывает функцию gettext() с именем _() в пространстве имен __builtins__. Она также добавляет функцию ngettext() и другие функции, перечисленные в аргументе names.

15.1.5.2. Локализация модуля

Изменение пространства имен __builtins__ библиотеки или отдельного модуля — не совсем хорошая идея, поскольку это может породить конфликты с глобальными значениями приложения. Вместо этого лучше импортировать или повторно связать функции перевода вручную на верхнем уровне модуля.

Листинг 15.10. gettext_module_global.py

```
import gettext

t = gettext.translation(
    'example',
    'locale',
    fallback=False,
)
_ = t.gettext
ngettext = t.ngettext

print(_('This message is in the script.'))
```

15.1.6. Переключение вариантов перевода

Во всех предыдущих примерах на протяжении всего времени выполнения программы использовался один и тот же каталог сообщений. Однако в некоторых ситуациях, особенно в случае веб-приложений, в разное время должны использоваться разные каталоги сообщений, причем это должно осуществляться без выхода из приложения или переустановки переменных среды. В подобных случаях удобно использовать API на основе классов, предоставляемых модулем `gettext`. Вызовы этого API остаются в основном теми же, что и глобальные вызовы, описанные в данном разделе, однако предоставление объекта каталога сообщений, которым можно манипулировать непосредственно, обеспечивает возможность использования нескольких каталогов сообщений.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `gettext`[1].
- `locale` (раздел 15.2). Модуль, содержащий другие инструменты локализации.
- GNU gettext[2]. Все форматы каталога сообщений, API и другие вспомогательные средства для этого модуля основаны на оригинальном пакете GNU gettext. С этим пакетом совместимы форматы файлов каталогов, а сценарии командной строки имеют аналогичные (если не идентичные) параметры. В руководстве *GNU gettext utilities*[3] содержится подробное описание форматов файлов и GNU-версий инструментов для работы с ними.
- Формы множественного числа[4]. Обработка форм множественного числа в словах и предложениях на различных языках.
- *Internationalization and Nationalization* (Martin von Löwis)[5]. Статья с описанием методик интернационализации приложений Python.
- Поддержка интернационализации приложений в Django[6]. Неплохой источник информации относительно использования модуля `gettext`, включающий примеры из реальных приложений.

15.2. `locale`: API локализации

Модуль `locale` является частью библиотеки Python, обеспечивающей поддержку интернационализации и локализации приложений. Она предоставляет стандартный способ выполнения операций, которые могут зависеть от языка общения или местоположения пользователя. В число таких операций входят, в частности, форматирование денежных значений, сравнение строк с целью их сортировки и работа с датами. Библиотека не поддерживает перевод текста с одного языка на другой (см. описание модуля `gettext` в разделе 15.1) и преобразование текста в кодировку Unicode (см. описание модуля `codecs` в разделе 6.10).

---

[1] `https://docs.python.org/3.5/library/gettext.html`

[2] `www.gnu.org/software/gettext/`

[3] `www.gnu.org/software/gettext/manual/gettext.html`

[4] `www.gnu.org/software/gettext/manual/gettext.html#Plural-forms`

[5] `http://legacy.python.org/workshops/1997-10/proceedings/loewis.html`

[6] `https://docs.djangoproject.com/en/dev/topics/i18n/`

Примечание

Изменение локали может влиять на все приложение в целом, поэтому рекомендуется не изменять ее в библиотеке, а позволить приложению самостоятельно устанавливать ее в случае необходимости. В приведенных в этом разделе примерах локаль изменяется несколько раз в небольшой программе, чтобы продемонстрировать различия в настройке разнообразных локалей. Вероятно, в случае реальных приложений целесообразно устанавливать локаль лишь один раз при запуске приложения или получении веб-запроса, а не изменять ее многократно.

В этом разделе обсуждаются некоторые высокоуровневые функции, входящие в модуль `locale`. Другие функции — низкоуровневые (`format_string()`) или связанные с управлением локалью приложения (`resetlocale()`).

15.2.1. Проверка региональных настроек

Наиболее распространенный способ дать пользователю возможность изменять настройки локали приложения — использовать переменную среды (`LC_ALL`, `LC_CTYPE`, `LANG` или `LANGUAGE`, в зависимости от платформы). Это позволяет вызывать функцию `setlocale()` без жестко заданных значений.

Листинг 15.11. `locale_env.py`

```
import locale
import os
import pprint

# Значения по умолчанию, заданные в пользовательской среде
locale.setlocale(locale.LC_ALL, '')

print('Environment settings:')
for env_name in ['LC_ALL', 'LC_CTYPE', 'LANG', 'LANGUAGE']:
    print('  {} = {}'.format(
        env_name, os.environ.get(env_name, ''))
    )

# Что собой представляет локаль?
print('\nLocale from environment:', locale.getlocale())

template = """
Numeric formatting:

  Decimal point      : "{decimal_point}"
  Grouping positions : {grouping}
  Thousands separator: "{thousands_sep}"

Monetary formatting:

  International currency symbol   : "{int_curr_symbol!r}"
  Local currency symbol           : {currency_symbol!r}
  Symbol precedes positive value  : {p_cs_precedes}
  Symbol precedes negative value  : {n_cs_precedes}
  Decimal point                   : "{mon_decimal_point}"
```

```
  Digits in fractional values     : {frac_digits}
  Digits in fractional values,
                  international  : {int_frac_digits}
  Grouping positions              : {mon_grouping}
  Thousands separator             : "{mon_thousands_sep}"
  Positive sign                   : "{positive_sign}"
  Positive sign position          : {p_sign_posn}
  Negative sign                   : "{negative_sign}"
  Negative sign position          : {n_sign_posn}

"""

sign_positions = {
    0: 'Surrounded by parentheses',
    1: 'Before value and symbol',
    2: 'After value and symbol',
    3: 'Before value',
    4: 'After value',
    locale.CHAR_MAX: 'Unspecified',
}

info = {}
info.update(locale.localeconv())
info['p_sign_posn'] = sign_positions[info['p_sign_posn']]
info['n_sign_posn'] = sign_positions[info['n_sign_posn']]

print(template.format(**info))
```

Метод `localeconv()` возвращает словарь, содержащий правила данной локали. С полным списком названий и определений можно ознакомиться в документации стандартной библиотеки.

На компьютере Mac, работающем под управлением OS X 10.11.6, выполнение этого сценария без присваивания переменным среды каких-либо значений дало следующие результаты:

```
$ export LANG=; export LC_CTYPE=; python3 locale_env.py

Environment settings:
  LC_ALL =
  LC_CTYPE =
  LANG =
  LANGUAGE =

Locale from environment: ('None', 'None')

Numeric formatting:

  Decimal point      : "."
  Grouping positions : []
  Thousands separator: ""

Monetary formatting:
```

```
  International currency symbol   : "''"
  Local currency symbol            : '$'
  Symbol precedes positive value   : 127
  Symbol precedes negative value   : 127
  Decimal point                    : ""
  Digits in fractional values      : 127
  Digits in fractional values,
                  international    : 127
  Grouping positions               : []
  Thousands separator              : ""
  Positive sign                    : ""
  Positive sign position           : Unspecified
  Negative sign                    : ""
  Negative sign position           : Unspecified
```

Выполнив этот же сценарий с установленной переменной LANG, можно увидеть, как при этом изменится локаль и установленная по умолчанию кодировка символов.

США (en_US)

```
$ LANG=en_US LC_CTYPE=en_US LC_ALL=en_US python3 locale_env.py

Environment settings:
  LC_ALL = en_US
  LC_CTYPE = en_US
  LANG = en_US
  LANGUAGE =

Locale from environment: ('en_US', 'ISO8859-1')

Numeric formatting:

  Decimal point      : "."
  Grouping positions : [3, 3, 0]
  Thousands separator: ","

Monetary formatting:

  International currency symbol   : "'USD'"
  Local currency symbol            : '$'
  Symbol precedes positive value   : 1
  Symbol precedes negative value   : 1
  Decimal point                    : "."
  Digits in fractional values      : 2
  Digits in fractional values,
                  international    : 2
  Grouping positions               : [3, 3, 0]
  Thousands separator              : ","
  Positive sign                    : ""
  Positive sign position           : Before value and symbol
  Negative sign                    : "-"
  Negative sign position           : Before value and symbol
```

Франция (fr_FR)

```
$ LANG=fr_FR LC_CTYPE=fr_FR LC_ALL=fr_FR python3 locale_env.py

Environment settings:
  LC_ALL = fr_FR
  LC_CTYPE = fr_FR
  LANG = fr_FR
  LANGUAGE =

Locale from environment: ('fr_FR', 'ISO8859-1')

Numeric formatting:

  Decimal point      : ","
  Grouping positions : [127]
  Thousands separator: ""

Monetary formatting:

  International currency symbol   : "'EUR'"
  Local currency symbol           : 'Eu'
  Symbol precedes positive value  : 0
  Symbol precedes negative value  : 0
  Decimal point                   : ","
  Digits in fractional values     : 2
  Digits in fractional values,
                  international  : 2
  Grouping positions              : [3, 3, 0]
  Thousands separator             : " "
  Positive sign                   : ""
  Positive sign position          : Before value and symbol
  Negative sign                   : "-"
  Negative sign position          : After value and symbol
```

Испания (es_ES)

```
$ LANG=es_ES LC_CTYPE=es_ES LC_ALL=es_ES python3 locale_env.py

Environment settings:
  LC_ALL = es_ES
  LC_CTYPE = es_ES
  LANG = es_ES
  LANGUAGE =

Locale from environment: ('es_ES', 'ISO8859-1')

Numeric formatting:

  Decimal point      : ","
  Grouping positions : [127]
  Thousands separator: ""
```

```
Monetary formatting:

  International currency symbol   : "'EUR'"
  Local currency symbol           : 'Eu'
  Symbol precedes positive value  : 0
  Symbol precedes negative value  : 0
  Decimal point                   : ","
  Digits in fractional values     : 2
  Digits in fractional values,
                  international   : 2
  Grouping positions              : [3, 3, 0]
  Thousands separator             : " "
  Positive sign                   : ""
  Positive sign position          : Before value and symbol
  Negative sign                   : "-"
  Negative sign position          : Before value and symbol
```

Португалия (pt_PT)

```
$ LANG=pt_PT LC_CTYPE=pt_PT LC_ALL=pt_PT python3 locale_env.py

Environment settings:
  LC_ALL = pt_PT
  LC_CTYPE = pt_PT
  LANG = pt_PT
  LANGUAGE =

Locale from environment: ('pt_PT', 'ISO8859-1')

Numeric formatting:

  Decimal point      : ","
  Grouping positions : []
  Thousands separator: " "

Monetary formatting:

  International currency symbol   : "'EUR'"
  Local currency symbol           : 'Eu'
  Symbol precedes positive value  : 0
  Symbol precedes negative value  : 0
  Decimal point                   : "."
  Digits in fractional values     : 2
  Digits in fractional values,
                  international   : 2
  Grouping positions              : [3, 3, 0]
  Thousands separator             : " "
  Positive sign                   : ""
  Positive sign position          : Before value and symbol
  Negative sign                   : "-"
  Negative sign position          : Before value and symbol
```

Польша (pl_PL)

```
$ LANG=pl_PL LC_CTYPE=pl_PL LC_ALL=pl_PL python3 locale_env.py

Environment settings:
  LC_ALL = pl_PL
  LC_CTYPE = pl_PL
  LANG = pl_PL
  LANGUAGE =

Locale from environment: ('pl_PL', 'ISO8859-2')

Numeric formatting:

  Decimal point      : ","
  Grouping positions : [3, 3, 0]
  Thousands separator: " "

Monetary formatting:

  International currency symbol   : "'PLN'"
  Local currency symbol           : 'z'
  Symbol precedes positive value  : 1
  Symbol precedes negative value  : 1
  Decimal point                   : ","
  Digits in fractional values     : 2
  Digits in fractional values,
                  international  : 2
  Grouping positions              : [3, 3, 0]
  Thousands separator             : " "
  Positive sign                   : ""
  Positive sign position          : After value and symbol
  Negative sign                   : "-"
  Negative sign position          : After value and symbol
```

15.2.2. Денежные единицы

Результаты предыдущего примера демонстрируют, что изменение локали приводит к соответствующему изменению символа валюты и символа, отделяющего дробную часть числа от целой. В следующем примере выводятся положительные и отрицательные денежные значения для нескольких локалей, перебираемых в цикле.

Листинг 15.12. `locale_currency.py`

```
import locale

sample_locales = [
    ('USA', 'en_US'),
    ('France', 'fr_FR'),
    ('Spain', 'es_ES'),
    ('Portugal', 'pt_PT'),
    ('Poland', 'pl_PL'),
]
```

```
for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    print('{:>10}: {:>10}  {:>10}'.format(
        name,
        locale.currency(1234.56),
        locale.currency(-1234.56),
    ))
```

Результаты приведены ниже.

```
$ python3 locale_currency.py

       USA:   $1234.56   -$1234.56
    France: 1234,56 Eu  1234,56 Eu-
     Spain: 1234,56 Eu  -1234,56 Eu
  Portugal: 1234.56 Eu  -1234.56 Eu
    Poland: łz 1234,56 ł z 1234,56-
```

15.2.3. Форматирование чисел

Числа, не связанные с денежными значениями, также могут форматироваться по-разному в зависимости от локали. В частности, это относится к символу `grouping`, разделяющему группы разрядов в больших числах.

Листинг 15.13. `locale_grouping.py`

```
import locale

sample_locales = [
    ('USA', 'en_US'),
    ('France', 'fr_FR'),
    ('Spain', 'es_ES'),
    ('Portugal', 'pt_PT'),
    ('Poland', 'pl_PL'),
]

print('{:>10} {:>10} {:>15}'.format(
    'Locale', 'Integer', 'Float')
)
for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)

    print('{:>10}'.format(name), end=' ')
    print(locale.format('%10d', 123456, grouping=True), end=' ')
    print(locale.format('%15.2f', 123456.78, grouping=True))
```

Для форматирования чисел без символа валюты следует использовать не метод `currency()`, а метод `format()`.

```
$ python3 locale_grouping.py

    Locale     Integer           Float
       USA     123,456      123,456.78
```

```
    France     123456       123456,78
     Spain     123456       123456,78
  Portugal     123456       123456,78
    Poland    123 456      123 456,78
```

Чтобы преобразовать числа, отформатированные в соответствии с определенной локалью, в нормализованные числа, формат которых не связан с локалью, следует использовать функцию `delocalize()`.

Листинг 15.14. `locale_delocalize.py`

```
import locale

sample_locales = [
    ('USA', 'en_US'),
    ('France', 'fr_FR'),
    ('Spain', 'es_ES'),
    ('Portugal', 'pt_PT'),
    ('Poland', 'pl_PL'),
]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    localized = locale.format('%0.2f', 123456.78, grouping=True)
    delocalized = locale.delocalize(localized)
    print('{:>10}: {:>10}  {:>10}'.format(
        name,
        localized,
        delocalized,
    ))
```

Разделители групп разрядов удаляются, а разделитель целой и десятичной части чисел преобразуется в символ точки (`.`).

```
$ python3 locale_delocalize.py

       USA: 123,456.78   123456.78
    France:  123456,78   123456.78
     Spain:  123456,78   123456.78
  Portugal:  123456,78   123456.78
    Poland: 123 456,78   123456.78
```

15.2.4. Анализ чисел

Кроме генерации вывода в различных форматах модуль `locale` может оказаться полезным для синтаксического анализа вводимых чисел. Он включает функции `atoi()` и `atof()`, преобразующие строки в целочисленные значения и значения с плавающей точкой на основании соглашений о форматировании чисел, принятых для заданной локали.

Листинг 15.15. `locale_atof.py`

```
import locale

sample_data = [
    ('USA', 'en_US', '1,234.56'),
    ('France', 'fr_FR', '1234,56'),
    ('Spain', 'es_ES', '1234,56'),
    ('Portugal', 'pt_PT', '1234.56'),
    ('Poland', 'pl_PL', '1 234,56'),
]

for name, loc, a in sample_data:
    locale.setlocale(locale.LC_ALL, loc)
    print('{:>10}: {:>9} => {:f}'.format(
        name,
        a,
        locale.atof(a),
    ))
```

Парсер распознает символы разделителя групп разрядов и десятичной точки текущей локали.

```
$ python3 locale_atof.py
       USA:  1,234.56 => 1234.560000
    France:   1234,56 => 1234.560000
     Spain:   1234,56 => 1234.560000
  Portugal:   1234.56 => 1234.560000
    Poland:  1 234,56 => 1234.560000
```

15.2.5. Дата и время

Другим важным аспектом локализации является форматирование значений даты и времени.

Листинг 15.16. `locale_date.py`

```
import locale
import time

sample_locales = [
    ('USA', 'en_US'),
    ('France', 'fr_FR'),
    ('Spain', 'es_ES'),
    ('Portugal', 'pt_PT'),
    ('Poland', 'pl_PL'),
]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    format = locale.nl_langinfo(locale.D_T_FMT)
    print('{:>10}: {}'.format(name, time.strftime(format)))
```

В этом примере для вывода значений даты и времени используется строка форматирования, соответствующая текущей локали.

```
$ python3 locale_date.py

       USA: Fri Aug 5  17:33:31 2016
    France: Ven  5 août 17:33:31 2016
     Spain: vie  5 ago 17:33:31 2016
  Portugal: Sex  5 Ago 17:33:31 2016
    Poland: ptk  5 sie 17:33:31 2016
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `locale`[7].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `locale` (раздел A.6.24).
- `gettext` (раздел 15.1). Каталоги сообщений с вариантами перевода на другие языки.

[7] https://docs.python.org/3.5/library/locale.html

Глава 16

Инструменты разработки

За время существования Python постепенно сформировалась обширная экосистема модулей, которая облегчает жизнь разработчикам, избавляя их от необходимости создавать все с нуля. Та же самая философия применяется и к инструментам, лежащим в основе деятельности разработчика, даже если они не используются в окончательной версии программы. В этой главе обсуждаются модули Python, упрощающие решение таких повседневных задач разработки, как тестирование, отладка и профилирование приложений.

Простейшая форма оказания помощи разработчику — предоставление ему документации используемого кода. Модуль `pydoc` (раздел 16.1) генерирует форматированную справочную информацию из строк документирования, которыми должен снабжаться исходный код любого важного модуля.

Python включает два фреймворка автоматизированного тестирования, обеспечивающих автоматическую проверку работы кода и контроль корректности его поведения. Модуль `doctest` (раздел 16.2) извлекает тестовые сценарии из включенных в документацию примеров, которые могут либо встраиваться в код, либо предоставляться в виде независимых файлов. Модуль `unittest` (раздел 16.3) — это полноценный фреймворк автоматизированного тестирования с поддержкой *фикстур* (fixtures), предопределенных наборов тестов и обнаружения тестов.

Модуль `trace` (раздел 16.4) обеспечивает мониторинг выполнения программ на языке Python, создавая отчет, в котором отображаются данные о том, сколько раз выполнялась каждая строка кода. Эту информацию можно использовать для поиска путей выполнения, не охваченных набором автоматизированных тестов, и исследования графа вызовов функций с целью определения зависимостей между модулями.

Написание и выполнение тестов позволяет обнаруживать проблемы в большинстве программ. Python упрощает процесс отладки, поскольку необработанные ошибки обычно выводятся на консоль в виде трассировочной информации. В тех случаях, когда программа выполняется не в среде текстовой консоли, для подготовки аналогичного вывода в файл журнала или диалоговое окно сообщений можно использовать модуль `traceback` (раздел 16.5). Если стандартных трассировочных данных оказывается недостаточно, то для получения такой более подробной информации, как значения локальных переменных на каждом уровне стека, и просмотра контекста исходного кода следует использовать модуль `cgitb` (раздел 16.6). Для вывода сообщений об ошибках в веб-приложениях модуль `cgitb` обеспечивает форматирование трассировочной информации в формате HTML.

Как только локализован участок кода, в котором возникла проблема, дальнейший способ ее разрешения заключается в пошаговом выполнении кода с помощью интерактивного отладчика, предлагаемого модулем `pdb` (раздел 16.7), который позволяет быстро исследовать маршрут выполнения кода, приводящий к возникновению проблемной ситуации. Кроме того, модуль `pdb` облегчает прове-

дение экспериментов с использованием активных объектов, что позволяет значительно уменьшить количество итераций, необходимых для внесения в код соответствующих изменений и устранения ошибки.

Следующим шагом после того, как программа отлажена и работает корректно, является улучшение ее производительности. Используя модули `profile` (раздел 16.8) и `timeit` (раздел 16.9), разработчик может измерить скорость выполнения программы и выявить те ее части, которые работают медленнее всего и нуждаются в улучшении.

В языках, подобных Python, в которых пробелы являются элементом синтаксиса, важно строго придерживаться правил создания отступов в коде. Модуль `tabnanny` (раздел 16.10) предоставляет средство, позволяющее обнаруживать случаи некорректного использования отступов. Это средство можно применять в тестах для того, чтобы гарантировать соответствие кода минимальным стандартам до его проверки в репозитории исходного кода.

Программы на языке Python выполняются посредством их предоставления интерпретатору в виде байт-компилированной версии исходного кода программы. Байт-компилированные версии могут создаваться либо на лету, либо однократно во время помещения программы в пакет. Модуль `compileall` (раздел 16.11) предоставляет интерфейс, используемый программами-установщиками и менеджерами пакетов для создания файлов, содержащих байт-код модуля. Его также можно использовать в среде разработки для того, чтобы гарантировать отсутствие синтаксических ошибок и создать байт-компилированные файлы для включения в пакеты при выпуске программ.

Что касается уровня исходного кода, то модуль `pyclbr` (раздел 16.12) предоставляет обозреватель классов, который может использоваться текстовым редактором или иной программой для просмотра исходного кода программы на языке Python с целью исследования таких элементов, как функции или классы. Эти действия осуществляются без импортирования кода и благодаря этому не сопровождаются отрицательными побочными эффектами.

Виртуальные окружения Python, управляемые модулем `venv` (раздел 16.13), определяют изолированные среды, предназначенные для установки и выполнения программ. Они упрощают тестирование одной и той же программы с различными версиями зависимостей и установку различных программ на одном компьютере без риска возникновения конфликтов между ними.

Чтобы воспользоваться всеми преимуществами обширной экосистемы модулей, фреймворков и утилит, доступных через каталог пакетов PPI (Python Package Index), необходимо использовать установщик пакетов. Программа установки пакетов Python, `pip`, не распространяется вместе с интерпретатором ввиду того, что новые версии языка выпускаются значительно реже по сравнению с желательной частотой обновления данного инструмента. Для установки последней версии программы `pip` можно использовать модуль `ensurepip` (раздел 16.14).

16.1. pydoc: оперативная справка для модулей

Модуль `pydoc` импортирует модуль Python и использует его содержимое для генерации текста справки во время выполнения. Выходная информация включает строки документирования всех элементов, входящих в состав модуля, таких как классы, методы и функции.

16.1.1. Получение справки в формате простого текста

Если вызвать pydoc как утилиту командной строки, задав имя модуля Python в качестве аргумента, то она выведет на консоль справку к указанному модулю и его содержимое, используя терминальный пейджер, если таковой сконфигурирован. Например, чтобы получить текст справки по модулю atexit (раздел 14.10), следует выполнить команду pydoc atexit.

```
$ pydoc atexit

Help on built-in module atexit:

NAME
    atexit - allow programmer to define multiple exit functions
to be executed upon normal program termination.

DESCRIPTION
    Two public functions, register and unregister, are defined.

FUNCTIONS
    register(...)
        register(func, *args, **kwargs) -> func

        Register a function to be executed upon normal program
termination

            func - function to be called at exit
            args - optional arguments to pass to func
            kwargs - optional keyword arguments to pass to func

            func is returned to facilitate usage as a decorator.

    unregister(...)
        unregister(func) -> None

        Unregister an exit function which was previously
registered using atexit.register

            func - function to be unregistered

FILE
    (built-in)
```

16.1.2. Справка в формате HTML

Утилита pydoc также способна генерировать справочную информацию в формате HTML, записывая ее в статический файл, расположенный в локальном каталоге, или запуская веб-сервер, позволяющий просматривать текст справки в онлайн-режиме.

```
$ pydoc -w atexit
```

Предыдущая команда создает файл *atexit.html* в текущем каталоге, тогда как следующая команда запускает веб-сервер, прослушивающий запросы по адресу `http://localhost:5000/`.

```
$ pydoc -p 5000

Server ready at http://localhost:5000/
Server commands: [b]rowser, [q]uit
server> q
Server stopped
```

Сервер генерирует информацию на лету в процессе ее просмотра. Команда `b` автоматически открывает окно браузера, а команда `q` останавливает сервер.

16.1.3. Интерактивная справка

Модуль `pydoc` добавляет в список `__builtins__` функцию `help()`, поэтому та же информация доступна в интерактивной оболочке интерпретатора Python.

```
$ python

Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> help('atexit')
Help on module atexit:

NAME
    atexit - allow programmer to define multiple exit functions
to be executed upon normal program termination.

...
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `pydoc`[1].
- `inspect` (раздел 18.4). Модуль `inspect` может использоваться для программного извлечения строк документирования объектов.

16.2. `doctest`: тестирование документации

Модуль `doctest` тестирует исходный код путем выполнения примеров, встроенных в документацию, и проверки того, что они воспроизводят ожидаемые результаты. Данный модуль просматривает строки документирования с целью поиска примеров, выполняет примеры и сравнивает результирующий текст с ожидаемым значением. Многим разработчикам проще работать с модулем `doctest`, чем с модулем `unittest` (раздел 16.3), поскольку простейшие случаи его использования не требуют изучения API. Однако по мере усложнениях примеров отсут-

[1] `https://docs.python.org/3.5/library/pydoc.html`

ствие средств управления фикстурами (конфигурациями тестирования) в модуле doctest приводит к более громоздким тестам по сравнению с тестами, подготовленными с помощью модуля unittest.

16.2.1. Начало работы

Прежде чем приступить к работе с модулем doctests, необходимо создать тестовые примеры с помощью интерактивного интерпретатора и включить их в строки документирования модуля путем копирования и вставки. В приведенном ниже листинге предоставляются два примера для функции my_function().

Листинг 16.1. doctest_simple.py

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

Чтобы выполнить тесты, следует использовать модуль doctest в качестве основной программы, указав параметр -m при его запуске. Как правило, во время выполнения тестов не выводится никакая выходная информация, поэтому ниже используется параметр -v, обеспечивающий получение более информативного вывода.

```
$ python3 -m doctest -v doctest_simple.py

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple
1 items passed all tests:
   2 tests in doctest_simple.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Как правило, сами по себе примеры не могут объяснить работу функций, поэтому модуль doctest обеспечивает использование дополнительных текстовых описаний. Он ищет строки, начинающиеся с подсказки интерпретатора (>>>), которые указывают на начало набора тестовых данных, а признаком конца те-

стового набора служит пустая строка или следующая подсказка интерпретатора. Перемежающийся с тестовыми данными текст игнорируется и может выводиться в любом формате при условии, что он отличается от формата тестовых данных.

Листинг 16.2. doctest_simple_with_docs.py

```
def my_function(a, b):
    """Возвращает a * b.

    Работает с числами:

    >>> my_function(2, 3)
    6

    и строками:

    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

Внедрение дополнительного текста в строку документирования делает ее более информативной для читателя. Поскольку модуль doctest игнорирует этот текст, результаты остаются прежними.

```
$ python3 -m doctest -v doctest_simple_with_docs.py

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple_with_docs
1 items passed all tests:
   2 tests in doctest_simple_with_docs.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

16.2.2. Обработка непредсказуемого вывода

Иногда точный вывод невозможно предсказать, но он все равно должен оставаться тестируемым. Например, значения текущей даты и времени или идентификатор объекта могут меняться от одного прогона теста к другому, используемая по умолчанию точность представления чисел с плавающей точкой зависит от параметров компилятора, а строковое представление контейнерных объектов,

таких как словари, может быть недетерминированным. Несмотря на то что эти условия невозможно контролировать, существуют подходы, позволяющие справляться с подобными ситуациями.

Например, в CPython идентификаторы объектов основаны на адресах структур данных, хранящих объекты.

Листинг 16.3. doctest_unpredictable.py

```
class MyClass:
    pass


def unpredictable(obj):
    """Возвращает новый список, содержащий объект.

    >>> unpredictable(MyClass())
    [<doctest_unpredictable.MyClass object at 0x10055a2d0>]
    """
    return [obj]
```

Значения идентификаторов меняются при каждом запуске программы, поскольку она загружается в разные области памяти.

```
$ python3 -m doctest -v doctest_unpredictable.py

Trying:
    unpredictable(MyClass())
Expecting:
    [<doctest_unpredictable.MyClass object at 0x10055a2d0>]
**********************************************************************
File ".../doctest_unpredictable.py", line 17, in doctest_unpredict
able.unpredictable
Failed example:
    unpredictable(MyClass())
Expected:
    [<doctest_unpredictable.MyClass object at 0x10055a2d0>]
Got:
    [<doctest_unpredictable.MyClass object at 0x1016a4160>]
2 items had no tests:
    doctest_unpredictable
    doctest_unpredictable.MyClass
**********************************************************************
1 items had failures:
   1 of   1 in doctest_unpredictable.unpredictable
1 tests in 3 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

Если тесты включают значения, которые могут изменяться непредвиденным образом, и если фактические значения несущественны для результатов теста, то следует использовать опцию ELLIPSIS, указывающую на то, что при верификации значения некоторые его части можно игнорировать.

Листинг 16.4. doctest_ellipsis.py

```
class MyClass:
    pass


def unpredictable(obj):
    """Возвращает новый список, содержащий объект.

    >>> unpredictable(MyClass()) #doctest: +ELLIPSIS
    [<doctest_ellipsis.MyClass object at 0x...>]
    """
    return [obj]
```

Наличие комментария `#doctest: +ELLIPSIS` после вызова функции `unpredictable()` указывает модулю `doctest` на то, что для данного теста включена опция `ELLIPSIS`. Часть идентификатора, соответствующая адресу объекта в памяти, заменяется символом многоточия (`...`), так что эта часть ожидаемого значения игнорируется. Фактический вывод совпадает с ожидаемым, и тест считается успешно пройденным.

```
$ python3 -m doctest -v doctest_ellipsis.py

Trying:
    unpredictable(MyClass()) #doctest: +ELLIPSIS
Expecting:
    [<doctest_ellipsis.MyClass object at 0x...>]
ok
2 items had no tests:
    doctest_ellipsis
    doctest_ellipsis.MyClass
1 items passed all tests:
   1 tests in doctest_ellipsis.unpredictable
1 tests in 3 items.
1 passed and 0 failed.
Test passed.
```

Однако иногда непредсказуемое значение нельзя игнорировать, поскольку это сделает тест неполным или неточным. Например, простые тесты быстро усложняются в случае обработки типов данных, строковое представление которых может быть непоследовательным. Так, строковая форма словаря может меняться в зависимости от очередности добавления ключей.

Листинг 16.5. doctest_hashed_values.py

```
keys = ['a', 'aa', 'aaa']

print('dict:', {k: len(k) for k in keys})
print('set :', set(keys))
```

Из-за рандомизации хеш-значений и возникновения конфликтов между ключами внутренний порядок следования ключей словаря может быть разным при каждом запуске сценария. Множества используют тот же алгоритм хеширования и ведут себя аналогичным образом.

```
$ python3 doctest_hashed_values.py

dict: {'aa': 2, 'a': 1, 'aaa': 3}
set : {'aa', 'a', 'aaa'}

$ python3 doctest_hashed_values.py

dict: {'a': 1, 'aa': 2, 'aaa': 3}
set : {'a', 'aa', 'aaa'}
```

Наилучшим способом обработки таких специальных случаев является создание тестов, в которых вероятность изменения получаемых значений мала. В случае словарей и множеств это может означать, что вместо того, чтобы полагаться на строковое представление, следует выполнить индивидуальный поиск конкретных ключей, сгенерировать отсортированный список содержимого структуры данных или выполнить проверку на равенство литеральному значению.

Листинг 16.6. doctest_hashed_values_tests.py

```
import collections


def group_by_length(words):
    """Возвращает словарь, группирующий слова в наборы по длине.

    >>> grouped = group_by_length([ 'python', 'module', 'of',
    ... 'the', 'week' ])
    >>> grouped == { 2:set(['of']),
    ...              3:set(['the']),
    ...              4:set(['week']),
    ...              6:set(['python', 'module']),
    ...              }
    True

    """
    d = collections.defaultdict(set)
    for word in words:
        d[len(word)].add(word)
    return d
```

В предыдущем коде один пример фактически интерпретируется как два отдельных теста, в первом из которых не ожидается консольный вывод, а во втором ожидается булев результат операции сравнения.

```
$ python3 -m doctest -v doctest_hashed_values_tests.py

Trying:
    grouped = group_by_length([ 'python', 'module', 'of',
    'the', 'week' ])
Expecting nothing
ok
Trying:
    grouped == { 2:set(['of']),
```

```
                3:set(['the']),
                4:set(['week']),
                6:set(['python', 'module']),
                }
Expecting:
    True
ok
1 items had no tests:
    doctest_hashed_values_tests
1 items passed all tests:
   2 tests in doctest_hashed_values_tests.group_by_length
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

16.2.3. Трассировочная информация

Трассировочная информация представляет особый случай, связанный с изменением данных. Поскольку фигурирующие в ней пути зависят от места установки модуля в файловой системе, написание переносимых тестов было бы невозможным, если работать с путями так же, как и с остальными данными.

Листинг 16.7. `doctest_tracebacks.py`

```
def this_raises():
    """Эта функция всегда возбуждает исключение.

    >>> this_raises()
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/no/such/path/doctest_tracebacks.py", line 14, in
      this_raises
        raise RuntimeError('here is the error')
    RuntimeError: here is the error
    """
    raise RuntimeError('here is the error')
```

Модуль `doctest` предпринимает специальные меры для того, чтобы распознать трассировочную информацию и игнорировать те ее части, которые могут меняться от системы к системе.

```
$ python3 -m doctest -v doctest_tracebacks.py

Trying:
    this_raises()
Expecting:
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/no/such/path/doctest_tracebacks.py", line 14, in
      this_raises
        raise RuntimeError('here is the error')
    RuntimeError: here is the error
ok
```

```
1 items had no tests:
    doctest_tracebacks
1 items passed all tests:
   1 tests in doctest_tracebacks.this_raises
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Фактически игнорируются все строки трассировочной информации, располагающиеся ниже заголовка, поэтому их можно опустить.

Листинг 16.8. doctest_tracebacks_no_body.py

```
def this_raises():
    """Эта функция всегда возбуждает исключение.

    >>> this_raises()
    Traceback (most recent call last):
    RuntimeError: here is the error

    >>> this_raises()
    Traceback (innermost last):
    RuntimeError: here is the error
    """
    raise RuntimeError('here is the error')
```

Когда модулю doctest встречается строка заголовка стека вызовов (либо `Traceback (most recent call last):`, либо `Traceback (innermost last):` в разных версиях Python), он пропускает заголовок и находит имя исключения и сообщение, полностью игнорируя промежуточные строки.

```
$ python3 -m doctest -v doctest_tracebacks_no_body.py

Trying:
    this_raises()
Expecting:
    Traceback (most recent call last):
    RuntimeError: here is the error
ok
Trying:
    this_raises()
Expecting:
    Traceback (innermost last):
    RuntimeError: here is the error
ok
1 items had no tests:
    doctest_tracebacks_no_body
1 items passed all tests:
   2 tests in doctest_tracebacks_no_body.this_raises
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

16.2.4. Обработка пробелов

Выходные результаты реальных приложений обычно включают пробелы — пустые строки, символы табуляции и увеличенные междустрочные интервалы, облегчающие чтение результатов. В частности, причиной возникновения проблем при работе с модулем doctest могут стать пустые строки, поскольку они используются для разделения тестов.

Листинг 16.9. doctest_blankline_fail.py

```
def double_space(lines):
    """Выводит список строк с двойными пробелами.

    >>> double_space(['Line one.', 'Line two.'])
    Line one.

    Line two.

    """
    for l in lines:
        print(l)
        print()
```

Функция double_space() получает список входных строк и выводит их, применяя удвоенный междустрочный интервал.

```
$ python3 -m doctest -v doctest_blankline_fail.py

Trying:
    double_space(['Line one.', 'Line two.'])
Expecting:
    Line one.
**********************************************************************
File ".../doctest_blankline_fail.py", line 12, in doctest_blanklin
e_fail.double_space
Failed example:
    double_space(['Line one.', 'Line two.'])
Expected:
    Line one.
Got:
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
1 items had no tests:
    doctest_blankline_fail
**********************************************************************
1 items had failures:
   1 of   1 in doctest_blankline_fail.double_space
1 tests in 2 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

В этом примере тест не проходит, поскольку пустая строка, следующая за строкой `Line one.` в строке документирования, интерпретируется как признак окончания образца вывода. Чтобы обеспечить сопоставление с пустой строкой, следует заменить ее в образце вывода строкой `<BLANKLINE>`.

Листинг 16.10. doctest_blankline.py

```
def double_space(lines):
    """Выводит список строк с двойными пробелами.

    >>> double_space(['Line one.', 'Line two.'])
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
    """
    for l in lines:
        print(l)
        print()
```

В этом примере модуль `doctest` заменяет фактические пустые строки тем же литералом, прежде чем выполнить сравнение. Теперь фактические и ожидаемые значения совпадают, и тест проходит.

```
$ python3 -m doctest -v doctest_blankline.py

Trying:
    double_space(['Line one.', 'Line two.'])
Expecting:
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
ok
1 items had no tests:
    doctest_blankline
1 items passed all tests:
   1 tests in doctest_blankline.double_space
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Пробелы в пределах строки также могут порождать проблемы при тестировании. В следующем примере за цифрой `6` следует лишний пробел.

Листинг 16.11. doctest_extra_space.py

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

Лишние пробелы могут появляться в коде вследствие редактирования текста методом копирования и вставки. Если такие пробелы оказываются в конце строки, то они могут остаться незамеченными в исходном файле и будут такими же невидимыми в отчете о непрохождении теста.

```
$ python3 -m doctest -v doctest_extra_space.py

Trying:
    my_function(2, 3)
Expecting:
    6
**********************************************************************
File ".../doctest_extra_space.py", lin
e 15, in doctest_extra_space.my_function
Failed example:
    my_function(2, 3)
Expected:
    6
Got:
    6
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_extra_space
**********************************************************************
1 items had failures:
   1 of   2 in doctest_extra_spa+ce.my_function
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Использование одной из таких опций вывода отчета, как REPORT_NDIFF, позволяет отображать различия между фактическими и ожидаемыми значениями с большей степенью детализации, при этом лишние пробелы становятся видимыми.

Листинг 16.12. doctest_ndiff.py

```
def my_function(a, b):
    """
    >>> my_function(2, 3) #doctest: +REPORT_NDIFF
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

Также доступны опции вывода унифицированных (REPORT_UDIFF) и контекстных (REPORT_CDIFF) различий.

```
$ python3 -m doctest -v doctest_ndiff.py

Trying:
    my_function(2, 3) #doctest: +REPORT_NDIFF
Expecting:
    6
**********************************************************************
File "...\doctest_ndiff.py", line 16, in doctest_ndiff.my_function
Failed example:
    my_function(2, 3) #doctest: +REPORT_NDIFF
Differences (ndiff with -expected +actual):
    - 6
    ?  -
    + 6
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_ndiff
**********************************************************************
1 items had failures:
   1 of   2 in doctest_ndiff.my_function
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Иногда желательно добавить дополнительные пробелы в образце вывода для теста, но таким образом, чтобы модуль doctest игнорировал их. Например, читать структуры данных проще, если они распределены по нескольким строкам, даже если их представление вполне могло бы уместиться в одной строке.

```
def my_function(a, b):
    """Возвращает a * b.

    >>> my_function(['A', 'B'], 3) #doctest: +NORMALIZE_WHITESPACE
    ['A', 'B',
     'A', 'B',
     'A', 'B']

    Этот вариант не подходит из-за дополнительного пробела после [
    в списке.
    >>> my_function(['A', 'B'], 2) #doctest: +NORMALIZE_WHITESPACE
    [ 'A', 'B',
      'A', 'B', ]
    """
    return a * b
```

При включенной опции NORMALIZE_WHITESPACE любой пробел в фактическом и ожидаемом значениях считается совпадением. Пробел не может добавляться в ожидаемое значение там, где он отсутствует в выводе, но длина последовательностей пробелов в ожидаемом и фактическом значениях не обязана быть одинако-

вой. Первый тестовый пример соответствует этому требованию, и его тест проходит, даже если входные данные содержат дополнительные пробелы и символы новой строки. Второй пример включает по дополнительному пробелу после символа [и перед символом], и поэтому тест не проходит.

```
$ python3 -m doctest -v doctest_normalize_whitespace.py

Trying:
    my_function(['A', 'B'], 3) #doctest: +NORMALIZE_WHITESPACE
Expecting:
    ['A', 'B',
     'A', 'B',
     'A', 'B',]
**********************************************************************
File "doctest_normalize_whitespace.py", line 13, in doctest_normal
ize_whitespace.my_function
Failed example:
    my_function(['A', 'B'], 3) #doctest: +NORMALIZE_WHITESPACE
Expected:
    ['A', 'B',
     'A', 'B',
     'A', 'B']
Got:
    ['A', 'B', 'A', 'B', 'A', 'B']
Trying:
    my_function(['A', 'B'], 2) #doctest: +NORMALIZE_WHITESPACE
Expecting:
    [ 'A', 'B',
      'A', 'B', ]
**********************************************************************
File "doctest_normalize_whitespace.py", line 21, in doctest_normal
ize_whitespace.my_function
Failed example:
    my_function(['A', 'B'], 2) #doctest: +NORMALIZE_WHITESPACE
Expected:
    [ 'A', 'B',
      'A', 'B', ]
Got:
    ['A', 'B', 'A', 'B']
1 items had no tests:
    doctest_normalize_whitespace
**********************************************************************
1 items had failures:
   2 of   2 in doctest_normalize_whitespace.my_function
2 tests in 2 items.
0 passed and 2 failed.
***Test Failed*** 2 failures.
```

16.2.5. Местонахождение тестов

До сих пор во всех примерах тесты записывались в строках документирования функций, к которым они относятся. Такой подход удобен для пользователей, из-

влекающих справочную информацию из строк документирования с помощью той или иной служебной функции, например pydoc (раздел 16.1), но модуль doctest позволяет искать тесты и в других расположениях. Наиболее очевидными расположениями для поиска дополнительных тестов являются строки документирования, включенные в модуль на любом уровне.

Листинг 16.13. doctest_docstrings.py

```
"""Тесты могут появляться в любой строке документирования в модуле.

Тесты уровня модуля пересекают границы классов и функций.

>>> A('a') == B('b')
False
"""


class A:
    """Простой класс.

    >>> A('instance_name').name
    'instance_name'
    """

    def __init__(self, name):
        self.name = name

    def method(self):
        """Возвращает необычное значение.

        >>> A('name').method()
        'eman'
        """
        return ''.join(reversed(self.name))


class B(A):
    """Еще один простой класс.

    >>> B('different_name').name
    'different_name'
    """
```

Тесты могут включаться в строки документирования на уровне модуля, класса или функции.

```
$ python3 -m doctest -v doctest_docstrings.py

Trying:
    A('a') == B('b')
Expecting:
    False
ok
```

```
Trying:
    A('instance_name').name
Expecting:
    'instance_name'
ok
Trying:
    A('name').method()
Expecting:
    'eman'
ok
Trying:
    B('different_name').name
Expecting:
    'different_name'
ok
1 items had no tests:
    doctest_docstrings.A.__init__
4 items passed all tests:
   1 tests in doctest_docstrings
   1 tests in doctest_docstrings.A
   1 tests in doctest_docstrings.A.method
   1 tests in doctest_docstrings.B
4 tests in 5 items.
4 passed and 0 failed.
Test passed.
```

Иногда тесты должны включаться вместе с исходным кодом, но не в случае текста справки для модуля. В этом случае их следует располагать вне строк документирования. Одним из способов, используемых модулем `doctest` для определения местонахождения других тестов, является поиск переменной `__test__` на уровне модуля. Значением `__test__` должен быть словарь, сопоставляющий имена (заданные в виде строк) со строками, модулями, классами или функциями.

Листинг 16.14. `doctest_private_tests.py`

```
import doctest_private_tests_external

__test__ = {
    'numbers': """
>>> my_function(2, 3)
6

>>> my_function(2.0, 3)
6.0
""",

    'strings': """
>>> my_function('a', 3)
'aaa'

>>> my_function(3, 'a')
'aaa'
""",
```

```
    'external': doctest_private_tests_external,
}


def my_function(a, b):
    """Возвращает a * b
    """
    return a * b
```

Если значением, ассоциированным с ключом, является строка, то она интерпретируется как строка документирования и проверяется на наличие тестов. Если значением является класс или функция, то модуль `doctest` выполняет рекурсивный поиск строк документирования, пытаясь найти в них тесты. В следующем примере строка документирования модуля `doctest_private_tests_external` содержит один тест.

Листинг 16.15. `doctest_private_tests_external.py`

```
"""Внешние тесты, ассоциированные с doctest_private_tests.py.

>>> my_function(['A', 'B', 'C'], 2)
['A', 'B', 'C', 'A', 'B', 'C']
"""
```

Просмотрев файл примера, модуль `doctest` находит в общей сложности пять тестов, подлежащих выполнению.

```
$ python3 -m doctest -v doctest_private_tests.py

Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(3, 'a')
Expecting:
    'aaa'
ok
```

```
2 items had no tests:
    doctest_private_tests
    doctest_private_tests.my_function
3 items passed all tests:
   1 tests in doctest_private_tests.__test__.external
   2 tests in doctest_private_tests.__test__.numbers
   2 tests in doctest_private_tests.__test__.strings
5 tests in 5 items.
5 passed and 0 failed.
Test passed.
```

16.2.6. Внешняя документация

Смешивание тестов с обычным кодом — не единственный способ использования модуля doctest. Точно так же можно использовать примеры, встроенные в файлы документации внешних проектов, такие как файлы ReST (reStrusturedText).

Листинг 16.16. doctest_in_help.py

```
def my_function(a, b):
    """Возвращает a * b
    """
    return a * b
```

Справочная информация для этого пробного модуля сохранена в отдельном файле *doctest_in_help.txt*. Примеры использования модуля включены в справочный текст, и для их нахождения и выполнения можно использовать модуль doctest.

Листинг 16.17. doctest_in_help.txt

```
How to Use doctest_in_help.py
=============================

This library is very simple, since it only has one function called
''my_function()''.

Numbers
=======

''my_function()'' returns the product of its arguments.  For numbers,
that value is equivalent to using the ''*'' operator.

::

    >>> from doctest_in_help import my_function
    >>> my_function(2, 3)
    6

It also works with floating-point values.

::

    >>> my_function(2.0, 3)
```

```
    6.0

Non-Numbers
===========

Because ''*'' is also defined on data types other than numbers,
''my_function()'' works just as well if one of the arguments is a
string, a list, or a tuple.

::

    >>> my_function('a', 3)
    'aaa'

    >>> my_function(['A', 'B', 'C'], 2)
    ['A', 'B', 'C', 'A', 'B', 'C']
```

Тесты, включенные в текстовый файл, можно запускать из командной строки точно так же, как и модули Python.

```
$ python3 -m doctest -v doctest_in_help.txt

Trying:
    from doctest_in_help import my_function
Expecting nothing
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
1 items passed all tests:
   5 tests in doctest_in_help.txt
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

Обычно модуль настраивает тестовую среду выполнения так, чтобы она включала элементы тестируемого модуля, поэтому тестам необязательно импортиро-

вать его явно. Однако в данном случае тесты определены не в модуле Python, и модуль doctest не знает, как настроить глобальное пространство имен. Как следствие, примеры должны самостоятельно импортировать нужный модуль. Все тесты в данном файле разделяют один и тот же контекст, поэтому достаточно импортировать модуль в самом начале файла.

16.2.7. Выполнение тестов

Во всех предыдущих примерах использовалась программа запуска тестов из командной строки, встроенная в модуль doctest. Этот подход прост и удобен в случае тестирования одиночного модуля, но с ростом пакета тестируемых файлов его трудоемкость быстро возрастает. Для подобных случаев разработаны альтернативные подходы, доказавшие на практике свою эффективность.

16.2.7.1. Тестирование модулей

Инструкции для запуска модуля doctest по отношению к определенному исходному коду могут помещаться в конце соответствующих модулей.

Листинг 16.18. doctest_testmod.py

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Вызов функции testmod() лишь в том случае, если именем текущего модуля является __main__, гарантирует, что тесты будут выполняться только тогда, когда модуль вызывается как основная программа.

```
$ python3 doctest_testmod.py -v

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    __main__
```

```
1 items passed all tests:
   2 tests in __main__.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Первым аргументом функции `testmod()` является модуль, содержащий код, который должен быть проверен на наличие тестов. Отдельный сценарий тестирования может использовать эту возможность для импорта реального кода и поочередного выполнения тестов, содержащихся в каждом модуле.

Листинг 16.19. doctest_testmod_other_module.py

```
import doctest_simple

if __name__ == '__main__':
    import doctest
    doctest.testmod(doctest_simple)
```

Тестовый набор для проекта можно создать, импортируя каждый модуль и выполняя его тесты.

```
$ python3 doctest_testmod_other_module.py -v

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple
1 items passed all tests:
   2 tests in doctest_simple.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

16.2.7.2. Тестирование файлов

Функция `testfile()` работает аналогично функции `testmod()`, обеспечивая возможность явного вызова тестов, содержащихся во внешнем файле, из программы тестирования.

Листинг 16.20. doctest_testfile.py

```
import doctest

if __name__ == '__main__':
    doctest.testfile('doctest_in_help.txt')
```

Обе функции, `testmod()` и `testfile()`, включают необязательные параметры, позволяющие управлять поведением тестов посредством опций модуля `doctest`. Более подробное описание этих возможностей можно найти в документации стандартной библиотеки, однако следует отметить, что в большинстве случаев необходимости в использовании указанных опций не возникает.

```
$ python3 doctest_testfile.py -v

Trying:
    from doctest_in_help import my_function
Expecting nothing
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
1 items passed all tests:
   5 tests in doctest_in_help.txt
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

16.2.7.3. Набор тестов `unittest`

Если для тестирования одного и того же кода в различных ситуациях привлекаются модули `unittest` (раздел 16.3) и `doctest`, то для совместного выполнения всех тестов можно использовать интеграцию этих модулей. Для создания наборов тестов, совместимых с API программы запуска тестов модуля `unittest`, можно использовать два класса: `DocTestSuite` и `DocFileSuite`.

Листинг 16.21. `doctest_unittest.py`

```
import doctest
import unittest

import doctest_simple
```

```
suite = unittest.TestSuite()
suite.addTest(doctest.DocTestSuite(doctest_simple))
suite.addTest(doctest.DocFileSuite('doctest_in_help.txt'))

runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

Вместо создания отчетов для отдельных тестов результаты тестов, независимо от их источников, сводятся в единый отчет.

```
$ python3 doctest_unittest.py

my_function (doctest_simple)
Doctest: doctest_simple.my_function ... ok
doctest_in_help.txt
Doctest: doctest_in_help.txt ... ok

----------------------------------------
Ran 2 tests in 0.018s

OK
```

16.2.8. Контекст тестирования

Контекст выполнения, создаваемый модулем `doctest` во время тестирования, содержит копии глобальных значений уровня модуля. Каждый источник тестов (например, функция, класс, модуль) имеет собственный набор глобальных значений, что в определенной степени изолирует тесты и снижает вероятность их влияния друг на друга.

Листинг 16.22. `doctest_test_globals.py`

```
class TestGlobals:

    def one(self):
        """
        >>> var = 'value'
        >>> 'var' in globals()
        True
        """

    def two(self):
        """
        >>> 'var' in globals()
        False
        """
```

Класс `TestGlobals` обладает двумя методами: `one()` и `two()`. Тесты в строке документирования метода `one()` устанавливают глобальную переменную, тогда как тест для метода `two()` ищет эту переменную (но не рассчитывает найти ее).

```
$ python3 -m doctest -v doctest_test_globals.py

Trying:
    var = 'value'
Expecting nothing
ok
Trying:
    'var' in globals()
Expecting:
    True
ok
Trying:
    'var' in globals()
Expecting:
    False
ok
2 items had no tests:
    doctest_test_globals
    doctest_test_globals.TestGlobals
2 items passed all tests:
   2 tests in doctest_test_globals.TestGlobals.one
   1 tests in doctest_test_globals.TestGlobals.two
3 tests in 4 items.
3 passed and 0 failed.
Test passed.
```

Однако это вовсе не означает, что тесты не могут взаимодействовать между собой, если у них есть возможность изменять содержимое общих переменных, определенных в модуле.

Листинг 16.23. doctest_mutable_globals.py

```
_module_data = {}


class TestGlobals:

    def one(self):
        """
        >>> TestGlobals().one()
        >>> 'var' in _module_data
        True
        """
        _module_data['var'] = 'value'

    def two(self):
        """
        >>> 'var' in _module_data
        False
        """
```

Переменная `_module_data` модуля изменяется тестами для метода `one()`, в результате чего тест для модуля `two()` не проходит.

```
$ python3 -m doctest -v doctest_mutable_globals.py

Trying:
    TestGlobals().one()
Expecting nothing
ok
Trying:
    'var' in _module_data
Expecting:
    True
ok
Trying:
    'var' in _module_data
Expecting:
    False
**********************************************************************
File ".../doctest_mutable_globals.py", line 25, in doctest_mutable
_globals.TestGlobals.two
Failed example:
    'var' in _module_data
Expected:
    False
Got:
    True
2 items had no tests:
    doctest_mutable_globals
    doctest_mutable_globals.TestGlobals
1 items passed all tests:
   2 tests in doctest_mutable_globals.TestGlobals.one
**********************************************************************
1 items had failures:
   1 of   1 in doctest_mutable_globals.TestGlobals.two
3 tests in 4 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.
```

Если тестам необходимо использовать глобальные значения (например, для параметризации среды выполнения), их можно передать функциям `testmod()` и `testfile()` для настройки контекста с использованием данных, контролируемых вызывающим кодом.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `doctest`[2].
- *The Mighty Dictionary* (Brandon Rhodes)[3]. Презентация материала, посвященного внутренним механизмам работы словарей, на конференции PyCon 2010.
- `difflib` (раздел 1.4). Библиотека Python содержит инструменты для вычисления различий между последовательностями и, в частности, для сравнения текста.

[2] https://docs.python.org/3.5/library/doctest.html

[3] www.youtube.com/watch?v=C4Kc8xzcA68

- Sphinx[4]. Генератор документации стандартной библиотеки Python, который в силу простоты его использования и способности генерировать аккуратный вывод в нескольких цифровых и печатных форматах был адаптирован для использования в независимых проектах. Sphinx включает расширение для выполнения тестов при обработке исходных файлов документации.
- py.test[5]. Независимая программа для выполнения тестов с поддержкой модуля doctest.
- nose2[6]. Независимая программа для выполнения тестов с поддержкой модуля doctest.
- Manuel[7]. Независимая программа для выполнения тестов на основе документации с дополнительными возможностями извлечения тестовых примеров и интеграции с генератором документации Sphinx.

16.3. unittest: фреймворк автоматизированного тестирования

Фреймворк автоматизированного тестирования из модуля unittest основан на архитектуре XUnit, разработанной Кентом Беком и Эрихом Гаммой. Та же идея повторена во многих других языках программирования, включая C, Perl, Java и Smalltalk. Фреймворк, реализованный в модуле unittest, обеспечивает автоматизацию тестирования за счет поддержки тестовых конфигураций, наборов тестов и средств для их выполнения.

16.3.1. Базовая структура тестов

Тесты, определяемые модулем unittest, формируются из двух составляющих: *фикстур* (fixtures) — кода, управляющего *зависимостями тестов*, и собственно тестов. Отдельные тесты формируются посредством создания подклассов TestCase и переопределения или добавления соответствующих методов. В следующем примере класс SimplisticTest содержит единственный метод test() для выполнения теста, который не должен проходить, если a не равно b.

Листинг 16.24. unittest_simple.py

```
import unittest


class SimplisticTest(unittest.TestCase):

    def test(self):
        a = 'a'
        b = 'a'
        self.assertEqual(a, b)
```

[4] www.sphinx-doc.org
[5] http://doc.pytest.org/en/latest/
[6] https://nose2.readthedocs.io/en/latest/
[7] https://pythonhosted.org/manuel/

16.3.2. Выполнение тестов

Простейший способ выполнения тестов unittest — использование средства автоматического обнаружения тестов, доступного через интерфейс командной строки.

```
$ python3 -m unittest unittest_simple.py

.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Этот сокращенный вывод включает количество времени, которое потребовалось для выполнения тестов, и индикатор состояния для каждого теста (символ . в первой строке вывода говорит о том, что тест прошел). Для получения более подробных результатов тестирования следует использовать параметр -v.

```
$ python3 -m unittest -v unittest_simple.py

test (unittest_simple.SimplisticTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

16.3.3. Результаты тестов

Возможны три исхода тестов (табл. 16.1). Не существует явных способов заставить тест пройти, поэтому статус теста зависит от наличия (или отсутствия) исключений.

Таблица 16.1. Возможные исходы тестовых сценариев

Исход	Описание
ok	Тест проходит
FAIL	Тест не проходит и возбуждает исключение AssertionError
ERROR	Тест возбуждает исключение, отличное от AssertionError

Листинг 16.25. unittest_outcomes.py

```
import unittest


class OutcomesTest(unittest.TestCase):

    def testPass(self):
        return

    def testFail(self):
        self.assertFalse(True)
```

```
    def testError(self):
        raise RuntimeError('Test error!')
```

Если тест не проходит или генерирует ошибку, то в результаты включается трассировочная информация.

```
$ python3 -m unittest unittest_outcomes.py

EF.
======================================================================
ERROR: testError (unittest_outcomes.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_outcomes.py", line 18, in testError
    raise RuntimeError('Test error!')
RuntimeError: Test error!

======================================================================
FAIL: testFail (unittest_outcomes.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_outcomes.py", line 15, in testFail
    self.assertFalse(True)
AssertionError: True is not false

----------------------------------------------------------------------
Ran 3 tests in 0.001s
FAILED (failures=1, errors=1)
```

В предыдущем примере метод `testFail()` не проходит тестирование, и в стеке вызовов отображается номер строки, по вине которой это произошло. Однако выяснение точной причины непрохождения теста зависит лишь от человека, оценивающего результаты тестирования, поскольку это требует изучения кода.

Листинг 16.26. `unittest_failwithmessage.py`

```
import unittest


class FailureMessageTest(unittest.TestCase):

    def testFail(self):
        self.assertFalse(True, 'failure message goes here')
```

Чтобы упростить выяснение причины непрохождения теста, можно передать методам `fail*()` и `assert*()` аргумент `msg`, позволяющий вывести более информативное сообщение об ошибке.

```
$ python3 -m unittest -v unittest_failwithmessage.py

testFail (unittest_failwithmessage.FailureMessageTest) ... FAIL

======================================================================
```

```
FAIL: testFail (unittest_failwithmessage.FailureMessageTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_failwithmessage.py", line 12, in testFail
    self.assertFalse(True, 'failure message goes here')
AssertionError: True is not false : failure message goes here

----------------------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures=1)
```

16.3.4. Подтверждение выполнения условия

Большинство тестов проверяет истинность некоторых условий. Существуют два подхода к написанию таких тестов, и выбор одного из них зависит от намерений автора тестов и ожидаемого результата работы тестируемого кода.

Листинг 16.27. unittest_truth.py

```
import unittest


class TruthTest(unittest.TestCase):

    def testAssertTrue(self):
        self.assertTrue(True)

    def testAssertFalse(self):
        self.assertFalse(False)
```

Если код генерирует значение, эквивалентное значению `True`, то следует использовать метод `assertTrue()`. Если же код генерирует значение, которое эквивалентно значению `False`, то целесообразнее использовать метод `assertFalse()`.

```
$ python3 -m unittest -v unittest_truth.py

testAssertFalse (unittest_truth.TruthTest) ... ok
testAssertTrue (unittest_truth.TruthTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

16.3.5. Тестирование равенства

Модуль `unittest` включает методы, предназначенные для проверки равенства двух значений.

Листинг 16.28. unittest_equality.py

```
import unittest


class EqualityTest(unittest.TestCase):

    def testExpectEqual(self):
        self.assertEqual(1, 3 - 2)

    def testExpectEqualFails(self):
        self.assertEqual(2, 3 - 2)

    def testExpectNotEqual(self):
        self.assertNotEqual(2, 3 - 2)

    def testExpectNotEqualFails(self):
        self.assertNotEqual(1, 3 - 2)
```

В случае непрохождения тестов эти специальные методы генерируют сообщения об ошибках, идентифицирующие сравниваемые значения.

```
$ python3 -m unittest -v unittest_equality.py

testExpectEqual (unittest_equality.EqualityTest) ... ok
testExpectEqualFails (unittest_equality.EqualityTest) ... FAIL
testExpectNotEqual (unittest_equality.EqualityTest) ... ok
testExpectNotEqualFails (unittest_equality.EqualityTest) ...
FAIL

======================================================================
FAIL: testExpectEqualFails (unittest_equality.EqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
File ".../unittest_equality.py", line 15, in
testExpectEqualFails
self.assertEqual(2, 3 - 2)
AssertionError: 2 != 1

======================================================================
FAIL: testExpectNotEqualFails (unittest_equality.EqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
File ".../unittest_equality.py", line 21, in
testExpectNotEqualFails
self.assertNotEqual(1, 3 - 2)
AssertionError: 1 == 1

----------------------------------------------------------------------
Ran 4 tests in 0.001s
FAILED (failures=2)
```

16.3.6. Приблизительное равенство

Кроме проверки на строгое равенство возможно тестирование приблизительного равенства чисел с плавающей точкой с помощью методов `assertAlmostEqual()` и `assertNotAlmostEqual()`.

Листинг 16.29. `unittest_almostequal.py`

```
import unittest


class AlmostEqualTest(unittest.TestCase):

    def testEqual(self):
        self.assertEqual(1.1, 3.3 - 2.2)

    def testAlmostEqual(self):
        self.assertAlmostEqual(1.1, 3.3 - 2.2, places=1)

    def testNotAlmostEqual(self):
        self.assertNotAlmostEqual(1.1, 3.3 - 2.0, places=1)
```

Аргументами этих методов являются сравниваемые значения и количество десятичных знаков после запятой, которые следует учитывать при сравнении.

```
$ python3 -m unittest unittest_almostequal.py

.F.
======================================================================
FAIL: testEqual (unittest_almostequal.AlmostEqualTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_almostequal.py", line 12, in testEqual
    self.assertEqual(1.1, 3.3 - 2.2)
AssertionError: 1.1 != 1.0999999999999996

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

16.3.7. Контейнеры

Кроме обобщенных методов `assertEqual()` и `assertNotEqual()` доступны специальные методы, предназначенные для сравнения контейнерных объектов, таких как списки, словари и множества.

Листинг 16.30. `unittest_equality_container.py`

```
import textwrap
import unittest
```

```
class ContainerEqualityTest(unittest.TestCase):

    def testCount(self):
        self.assertCountEqual(
            [1, 2, 3, 2],
            [1, 3, 2, 3],
        )

    def testDict(self):
        self.assertDictEqual(
            {'a': 1, 'b': 2},
            {'a': 1, 'b': 3},
        )

    def testList(self):
        self.assertListEqual(
            [1, 2, 3],
            [1, 3, 2],
        )

    def testMultiLineString(self):
        self.assertMultiLineEqual(
            textwrap.dedent("""
            This string
            has more than one
            line.
            """),
            textwrap.dedent("""
            This string has
            more than two
            lines.
            """),
        )

    def testSequence(self):
        self.assertSequenceEqual(
            [1, 2, 3],
            [1, 3, 2],
        )

    def testSet(self):
        self.assertSetEqual(
            set([1, 2, 3]),
            set([1, 3, 2, 4]),
        )

    def testTuple(self):
        self.assertTupleEqual(
            (1, 'a'),
            (1, 'b'),
        )
```

Каждый из этих методов сообщает о неравенстве с помощью формата, имеющего смысл для входного типа, что облегчает понимание причин непрохождения теста и внесение соответствующих исправлений.

```
$ python3 -m unittest unittest_equality_container.py

FFFFFFF
======================================================================
FAIL: testCount
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...\unittest_equality_container.py", line 15, in testCount
    [1, 3, 2, 3],
AssertionError: Element counts were not equal:
First has 2, Second has 1:  2
First has 1, Second has 2:  3

======================================================================
FAIL: testDict
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...unittest_equality_container.py", line 21, in testDict
    {'a': 1, 'b': 3},
AssertionError: {'a': 1, 'b': 2} != {'a': 1, 'b': 3}
- {'a': 1, 'b': 2}
?              ^

+ {'a': 1, 'b': 3}
?              ^


======================================================================
FAIL: testList
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...\unittest_equality_container.py", line 27, in testList
    [1, 3, 2],
AssertionError: Lists differ: [1, 2, 3] != [1, 3, 2]

First differing element 1:
2
3

- [1, 2, 3]
+ [1, 3, 2]

======================================================================
FAIL: testMultiLineString
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "...\unittest_equality_container.py", line 41, in
testMultiLineString
    """),
AssertionError: '\nThis string\nhas more than one\nline.\n' !=
'\nThis string has\nmore than two\nlines.\n'

- This string
+ This string has
?            ++++
- has more than one
? ----           --
+ more than two
?           ++
- line.
+ lines.
?     +


======================================================================
FAIL: testSequence
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...\unittest_equality_container.py", line 47, in
testSequence
    [1, 3, 2],
AssertionError: Sequences differ: [1, 2, 3] != [1, 3, 2]

First differing element 1:
2
3

- [1, 2, 3]
+ [1, 3, 2]

======================================================================
FAIL: testSet (unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality_container.py", line 53, in testSet
    set([1, 3, 2, 4]),
AssertionError: Items in the second set but not the first:
4

======================================================================
FAIL: testTuple
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...\unittest_equality_container.py", line 59, in testTuple
    (1, 'b'),
AssertionError: Tuples differ: (1, 'a') != (1, 'b')
```

```
First differing element 1:
'a'
'b'

- (1, 'a')
?      ^

+ (1, 'b')
?      ^


----------------------------------------------------------------------
Ran 7 tests in 0.010s

FAILED (failures=7)
```

Для тестирования принадлежности контейнеру следует использовать метод `assertIn()`.

Листинг 16.31. `unittest_in.py`

```
import unittest


class ContainerMembershipTest(unittest.TestCase):

    def testDict(self):
        self.assertIn(4, {1: 'a', 2: 'b', 3: 'c'})

    def testList(self):
        self.assertIn(4, [1, 2, 3])

    def testSet(self):
        self.assertIn(4, set([1, 2, 3]))
```

Метод `assertIn()` применим в отношении любого объекта, который поддерживает оператор `in` или API контейнеров.

```
$ python3 -m unittest unittest_in.py

FFF
======================================================================
FAIL: testDict (unittest_in.ContainerMembershipTest)
----------------------------------------------------------------------
Traceback (most recent call last):
File ".../unittest_in.py", line 12, in testDict
self.assertIn(4, {1: 'a', 2: 'b', 3: 'c'})
AssertionError: 4 not found in {1: 'a', 2: 'b', 3: 'c'}

======================================================================
FAIL: testList (unittest_in.ContainerMembershipTest)
----------------------------------------------------------------------
Traceback (most recent call last):
File ".../unittest_in.py", line 15, in testList
```

```
self.assertIn(4, [1, 2, 3])
AssertionError: 4 not found in [1, 2, 3]

======================================================================
FAIL: testSet (unittest_in.ContainerMembershipTest)
----------------------------------------------------------------------
Traceback (most recent call last):
File ".../unittest_in.py", line 18, in testSet
self.assertIn(4, set([1, 2, 3]))
AssertionError: 4 not found in {1, 2, 3}

----------------------------------------------------------------------
Ran 3 tests in 0.001s
FAILED (failures=3)
```

16.3.8. Тестирование исключений

Как уже отмечалось, если тест возбуждает исключение, отличное от `AssertionError`, то оно трактуется как ошибка. Это поведение можно использовать для обнаружения ошибок при изменении кода, уже покрытого тестами. Однако в некоторых ситуациях тест должен проверять, что код действительно генерирует исключение. Например, если атрибуту объекта присваивается некорректное значение, то использование метода `assertRaises()` приводит к более понятному коду, чем перехват исключения в тесте. Следующий пример включает два теста, которые можно сравнить между собой на основании этих соображений.

Листинг 16.32. `unittest_exception.py`

```
import unittest


def raises_error(*args, **kwds):
    raise ValueError('Invalid value: ' + str(args) + str(kwds))


class ExceptionTest(unittest.TestCase):

    def testTrapLocally(self):
        try:
            raises_error('a', b='c')
        except ValueError:
            pass
        else:
            self.fail('Did not see ValueError')

    def testAssertRaises(self):
        self.assertRaises(
            ValueError,
            raises_error,
            'a',
            b='c',
        )
```

Оба теста приводят к одинаковым результатам, но результаты второго теста, в котором используется метод `assertRaises()`, более лаконичны.

```
$ python3 -m unittest -v unittest_exception.py

testAssertRaises (unittest_exception.ExceptionTest) ... ok
testTrapLocally (unittest_exception.ExceptionTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

16.3.9. Разделяемые контексты тестов

Конфигурационные объекты (разделяемые контексты), или *фикстуры* (fixtures) – это внешние ресурсы, или зависимости, необходимые тестам. Например, всем тестам одного класса может требоваться экземпляр другого класса, предоставляющий конфигурационные параметры или другие разделяемые ресурсы. К числу других фикстур относятся соединения с базами данных и временные файлы. (Многие будут утверждать, что использование внешних ресурсов лишает тесты статуса "блочных", но такие тесты по-прежнему остаются тестами и выполняют свою полезную роль.)

Модуль `unittest` включает специальные перехватчики, предназначенные для конфигурирования и освобождения любых разделяемых контекстов, необходимых тестам. Чтобы установить разделяемые контексты для каждого отдельного тестового сценария, следует переопределить метод `setUp()` класса `TestCase`. Чтобы освободить ресурсы, занимаемые разделяемыми контекстами, в которых больше нет необходимости, следует использовать метод `tearDown()`. Чтобы использовать один набор разделяемых ресурсов для всех экземпляров тестового класса, необходимо переопределить методы `setUpClass()` и `tearDownClass()` класса `TestCase`. Наконец, чтобы организовать управление особенно дорогостоящими операциями конфигурирования одновременно для всех тестов в пределах модуля, следует использовать функции уровня модуля `setUpModule()` и `tearDownModule()`.

Листинг 16.33. `unittest_fixtures.py`

```
import random
import unittest


def setUpModule():
    print('In setUpModule()')


def tearDownModule():
    print('In tearDownModule()')


class FixturesTest(unittest.TestCase):
```

```
    @classmethod
    def setUpClass(cls):
        print('In setUpClass()')
        cls.good_range = range(1, 10)

    @classmethod
    def tearDownClass(cls):
        print('In tearDownClass()')
        del cls.good_range

    def setUp(self):
        super().setUp()
        print('\nIn setUp()')
        # Выбрать число, заведомо принадлежащее заданному
        # диапазону. Значение "stop" не включается в диапазон,
        # поэтому необходимо следить за тем, чтобы оно
        # не входило в набор допустимых значений для данного
        # варианта.
        self.value = random.randint(
            self.good_range.start,
            self.good_range.stop - 1,
        )

    def tearDown(self):
        print('In tearDown()')
        del self.value
        super().tearDown()

    def test1(self):
        print('In test1()')
        self.assertIn(self.value, self.good_range)

    def test2(self):
        print('In test2()')
        self.assertIn(self.value, self.good_range)
```

При выполнении этого теста очередность выполнения методов разделяемого контекста и теста очевидна.

```
$ python3 -u -m unittest -v unittest_fixtures.py

In setUpModule()
In setUpClass()
test1 (unittest_fixtures.FixturesTest) ...
In setUp()
In test1()
In tearDown()
ok
test2 (unittest_fixtures.FixturesTest) ...
In setUp()
In test2()
In tearDown()
ok
```

```
In tearDownClass()
In tearDownModule()

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Если в процессе очистки ресурсов разделяемых контекстов возникают ошибки, то, возможно, не все методы `tearDown()` будут вызваны. Для полной уверенности в том, что завершающие операции будут выполнены корректно, следует использовать метод `addCleanup()`.

Листинг 16.34. `unittest_addcleanup.py`

```
import random
import shutil
import tempfile
import unittest


def remove_tmpdir(dirname):
    print('In remove_tmpdir()')
    shutil.rmtree(dirname)


class FixturesTest(unittest.TestCase):

    def setUp(self):
        super().setUp()
        self.tmpdir = tempfile.mkdtemp()
        self.addCleanup(remove_tmpdir, self.tmpdir)

    def test1(self):
        print('\nIn test1()')

    def test2(self):
        print('\nIn test2()')
```

В этом примере теста создается временный каталог, который по завершении теста удаляется средствами модуля `shutil` (см. раздел 6.7).

```
$ python3 -u -m unittest -v unittest_addcleanup.py

test1 (unittest_addcleanup.FixturesTest) ...
In test1()
In remove_tmpdir()
ok
test2 (unittest_addcleanup.FixturesTest) ...
In test2()
In remove_tmpdir()
ok
```

```
----------------------------------------------------------------------
Ran 2 tests in 0.003s
OK
```

16.3.10. Повторение тестов с различными входными данными

Часто оказывается полезным выполнять одну и ту же логику теста с различными входными данными. В этом отношении популярен прием, в соответствии с которым вместо того, чтобы определять тестовые методы для каждого небольшого сценария по отдельности, следует создать один тестовый метод, содержащий несколько родственных вызовов утверждений. Суть проблемы, связанной с таким подходом, заключается в том, что коль скоро одно утверждение не выполняется, то все остальные пропускаются. Лучшим решением является использование метода `subTest()` с целью создания контекста для теста в пределах тестового метода. Если после этого тест не проходит, то выводится соответствующее сообщение и продолжается выполнение оставшихся тестов.

Листинг 16.35. `unittest_subtest.py`

```
import unittest


class SubTest(unittest.TestCase):

    def test_combined(self):
        self.assertRegex('abc', 'a')
        self.assertRegex('abc', 'B')
        # Следующие утверждения не проверяются!
        self.assertRegex('abc', 'c')
        self.assertRegex('abc', 'd')

    def test_with_subtest(self):
        for pat in ['a', 'B', 'c', 'd']:
            with self.subTest(pattern=pat):
                self.assertRegex('abc', pat)
```

В этом примере утверждения для шаблонов `'c'` и `'d'` в методе `test_combined()` никогда не будут выполнены. В то же время метод `test_with_subtest()` выполняется и корректно выводит сообщение о дополнительном случае непрохождения теста. Следует отметить, что программа, выполняющая тесты, считает, что существуют только два теста, хотя выводит три сообщения о том, что тесты не прошли.

```
$ python3 -m unittest -v unittest_subtest.py

test_combined (unittest_subtest.SubTest) ... FAIL
test_with_subtest (unittest_subtest.SubTest) ...
======================================================================
FAIL: test_combined (unittest_subtest.SubTest)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "...\unittest_subtest.py", line 13, in test_combined
    self.assertRegex('abc', 'B')
AssertionError: Regex didn't match: 'B' not found in 'abc'

======================================================================
FAIL: test_with_subtest (unittest_subtest.SubTest) (pattern='B')
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...\unittest_subtest.py", line 21, in test_with_subtest
    self.assertRegex('abc', pat)
AssertionError: Regex didn't match: 'B' not found in 'abc'

======================================================================
FAIL: test_with_subtest (unittest_subtest.SubTest) (pattern='d')
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...\unittest_subtest.py", line 21, in test_with_subtest
    self.assertRegex('abc', pat)
AssertionError: Regex didn't match: 'd' not found in 'abc'

----------------------------------------------------------------------
Ran 2 tests in 0.003s

FAILED (failures=3)
```

16.3.11. Пропуск тестов

Иногда полезно иметь возможность пропускать тот или иной тест, если не выполняется некоторое внешнее условие. Например, если пишутся тесты, проверяющие поведение библиотеки при работе с конкретной версией Python, то не имеет смысла выполнять эти тесты в случае использования другой версии. Если требуется, чтобы тесты всегда пропускались, можно декорировать тестовые классы и методы декоратором `skip()`. Для проверки условия перед пропуском тестов можно использовать декораторы `skipIf()` и `skipUnless()`.

Листинг 16.36. `unittest_skip.py`

```
import sys
import unittest


class SkippingTest(unittest.TestCase):

    @unittest.skip('always skipped')
    def test(self):
        self.assertTrue(False)

    @unittest.skipIf(sys.version_info[0] > 2,
                     'only runs on python 2')
    def test_python2_only(self):
        self.assertTrue(False)
```

```
    @unittest.skipUnless(sys.platform == 'Darwin',
                         'only runs on macOS')
    def test_macos_only(self):
        self.assertTrue(True)

    def test_raise_skiptest(self):
        raise unittest.SkipTest('skipping via exception')
```

В случае сложных условий, которые трудно задать в виде одного выражения, передаваемого методу `skipIf()` или `skipUnless()`, тестовый сценарий может обеспечить пропуск теста, непосредственно возбудив исключение `SkipTest`.

```
$ python3 -m unittest -v unittest_skip.py

test (unittest_skip.SkippingTest) ... skipped 'always skipped'
test_macos_only (unittest_skip.SkippingTest) ... skipped 'only
runs on macOS'
test_python2_only (unittest_skip.SkippingTest) ... skipped 'only
runs on python 2'
test_raise_skiptest (unittest_skip.SkippingTest) ... skipped
'skipping via exception'

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK (skipped=4)
```

16.3.12. Игнорирование неудачных тестов

Вместо удаления тестов, которые постоянно не проходят, можно пометить их декоратором `expectedFailure()`, благодаря чему эти случаи непрохождения тестов будут игнорироваться.

Листинг 16.37. `unittest_expectedfailure.py`

```
import unittest


class Test(unittest.TestCase):

    @unittest.expectedFailure
    def test_never_passes(self):
        self.assertTrue(False)

    @unittest.expectedFailure
    def test_always_passes(self):
        self.assertTrue(True)
```

Если ожидается, что тест не должен пройти, а на самом деле он проходит, то это условие трактуется как специальная разновидность непрохождения теста, и в таком случае выводится сообщение о непредвиденном прохождении теста.

```
$ python3 -m unittest -v unittest_expectedfailure.py

test_always_passes (unittest_expectedfailure.Test) ...
unexpected success
test_never_passes (unittest_expectedfailure.Test) ... expected
failure

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (expected failures=1, unexpected successes=1)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `unittest`[8].
- `doctest` (раздел 16.2). Альтернативное средство выполнения тестов, встроенных в строки документирования или содержащихся во внешних файлах документации.
- `nose`[9]. Сторонняя программа для выполнения тестов, предлагающая усовершенствованные средства обнаружения тестов.
- `pytest`[10]. Популярная сторонняя программа для выполнения тестов с поддержкой распределенного выполнения и альтернативной системой управления разделяемыми контекстами.
- `testrepository`[11]. Сторонняя программа для выполнения тестов, используемая в проекте OpenStack, которая поддерживает параллельное выполнение и отслеживание непрошедших тестов.

16.4. `trace`: трассировка выполнения программы

Модуль `trace` может быть полезным для понимания того, как работает программа. Он позволяет наблюдать за выполнением инструкций, создавать отчеты о покрытии кода и исследовать отношения между функциями, вызывающими одна другую.

16.4.1. Пример программы

Представленная ниже программа будет использоваться во всех примерах, приведенных в оставшейся части данного раздела. Эта программа импортирует модуль `recurse` и выполняет содержащиеся в нем функции.

Листинг 16.38. `trace_example/main.py`

```
from recurse import recurse


def main():
```

[8] https://docs.python.org/3.5/library/unittest.html
[9] https://nose.readthedocs.io/en/latest/
[10] http://doc.pytest.org/en/latest/
[11] http://testrepository.readthedocs.io/en/latest/

```
    print('This is the main program.')
    recurse(2)


if __name__ == '__main__':
    main()
```

Функция `recurse()` вызывает саму себя до тех пор, пока аргумент `level` не достигнет значения 0.

Листинг 16.39. `trace_example/recurse.py`

```
def recurse(level):
    print('recurse({})'.format(level))
    if level:
        recurse(level - 1)


def not_called():
    print('This function is never called.')
```

16.4.2. Трассировка выполнения

Модуль `trace` можно вызывать непосредственно из командной строки. Параметр `--trace` задает отображение строк кода по мере их выполнения. Кроме того, в приведенном ниже примере задан фильтр `--ignore-dir`, позволяющий избежать трассировки кода в модуле `importlib` (раздел 19.1) и других модулях стандартной библиотеки Python, что, возможно, представляло бы интерес в других случаях, но в данном простом примере лишь загромождало бы вывод.

```
$ python3 -m trace --ignore-dir=.../lib/python3.5 \
--trace trace_example/main.py

 --- modulename: main, funcname: <module>
main.py(7): """
main.py(10): from recurse import recurse
 --- modulename: recurse, funcname: <module>
recurse.py(7): """
recurse.py(11): def recurse(level):
recurse.py(17): def not_called():
main.py(13): def main():
main.py(17): if __name__ == '__main__':
main.py(18):     main()
 --- modulename: main, funcname: main
main.py(14):     print('This is the main program.')
This is the main program.
main.py(15):     recurse(2)
 --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(2)
recurse.py(13):     if level:
recurse.py(14):         recurse(level - 1)
 --- modulename: recurse, funcname: recurse
```

```
recurse.py(12):     print('recurse({})'.format(level))
recurse(1)
recurse.py(13):     if level:
recurse.py(14):     recurse(level - 1)
 --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(0)
recurse.py(13):     if level:
 --- modulename: trace, funcname: _unsettrace
trace.py(77):         sys.settrace(None)
```

В первой части вывода отображаются операции настройки, выполняемые модулем `trace`. Остальная часть вывода отображает вход в каждую функцию, включая модуль, в котором находится функция, и строки исходного файла по мере их выполнения. Как и следовало ожидать исходя из способа вызова функции `recurse()` в функции `main()`, вход в нее выполнялся три раза.

16.4.3. Покрытие кода

В результате запуска модуля `trace` из командной строки с указанием параметра `--count` создается отчет о покрытии кода, содержащий подробную информацию о том, какие строки выполняются, а какие пропускаются. Поскольку сложные программы состоят, как правило, из нескольких файлов, для каждого из них создается отдельный отчет о покрытии кода. По умолчанию файлы отчетов записываются в тот же каталог, в котором находится модуль, и получают имя модуля, но с расширением *.cover* вместо *.py*.

```
$ python3 -m trace --count trace_example/main.py

This is the main program.
recurse(2)
recurse(1)
recurse(0)
```

В этом примере создаются два выходных файла, содержимое которых представлено в приведенных ниже листингах.

Листинг 16.40. `trace_example/main.cover`

```
1: from recurse import recurse

1: def main():
1:     print 'This is the main program.'
1:     recurse(2)
1:     return

1: if __name__ == '__main__':
1:     main()
```

Листинг 16.41. `trace_example/recurse.cover`

```
1: def recurse(level):
3:     print 'recurse(%s)' % level
```

```
3:      if level:
2:          recurse(level-1)
3:      return

1: def not_called():
       print 'This function is never called.'
```

Примечание

Тот факт, что счетчик строки `def recurse(level):` равен 1, вовсе не говорит о том, что функция выполнялась только один раз. Это значение указывает лишь на то, сколько раз выполнялось *определение* функции. То же самое относится и к строке `def not_called():`, поскольку ее определение также обрабатывалось интерпретатором, даже если она ни разу не вызывалась.

Эту программу можно выполнить несколько раз, возможно, с другими параметрами, для сохранения общих данных о покрытии кода и получения объединенного отчета. При первом запуске модуля `trace` с указанием выходного файла будет выведено сообщение об ошибке, возникающей при попытке загрузить существующие данные для объединения с новыми результатами в условиях, когда выходной файл еще не создан.

```
$ python3 -m trace --coverdir coverdir1 --count \
--file coverdir1/coverage_report.dat trace_example/main.py

This is the main program.
recurse(2)
recurse(1)
recurse(0)
Skipping counts file 'coverdir1/coverage_report.dat': [Errno 2]
No such file or directory: 'coverdir1/coverage_report.dat'

$ python3 -m trace --coverdir coverdir1 --count \
--file coverdir1/coverage_report.dat trace_example/main.py

This is the main program.
recurse(2)
recurse(1)
recurse(0)

$ python3 -m trace --coverdir coverdir1 --count \
--file coverdir1/coverage_report.dat trace_example/main.py

This is the main program.
recurse(2)
recurse(1)
recurse(0)

$ ls coverdir1

coverage_report.dat
```

После получения и записи в файлы .cover достаточного объема информации о покрытии кода можно получить отчет, используя опцию --report.

```
$ python3 -m trace --coverdir coverdir1 --report --summary \
--missing --file coverdir1/coverage_report.dat \
trace_example/main.py

lines   cov%   module   (path)
  537     0%   trace   (.../lib/python3.5/trace.py)
    7   100%   trace_example.main  (trace_example/main.py)
    7    85%   trace_example.recurse
(trace_example/recurse.py)
```

Поскольку в данном случае программа выполнялась три раза, в отчете о покрытии кода выведены значения счетчиков строк, в три раза превышающие предыдущие. Опция --summary добавляет в результаты данные о покрытии кода в процентном выражении. Для модуля recurse этот показатель составил лишь 87%. В *.cover*-файле отражается тот факт, что тело функции not_called() вообще не выполнялось, на что указывает префикс >>>>>>.

Листинг 16.42. coverdir1/trace_example.recurse.cover

```
    3: def recurse(level):
    9:     print('recurse({})'.format(level))
    9:     if level:
    6:         recurse(level - 1)


    3: def not_called():
>>>>>>     print('This function is never called.')
```

16.4.4. Взаимные вызовы функций

В дополнение к информации о покрытии кода модуль trace может собирать данные и создавать отчет о взаимных вызовах функций. Чтобы получить простой список вызываемых функций, следует использовать опцию --listfuncs.

```
$ python3 -m trace --listfuncs trace_example/main.py | \
grep -v importlib

This is the main program.
recurse(2)
recurse(1)
recurse(0)

functions called:
filename: .../lib/python3.5/trace.py, modulename: trace,
funcname: _unsettrace
filename: trace_example/main.py, modulename: main, funcname:
<module>
filename: trace_example/main.py, modulename: main, funcname:
main
filename: trace_example/recurse.py, modulename: recurse,
```

```
funcname: <module>
filename: trace_example/recurse.py, modulename: recurse,
funcname: recurse
```

Для получения более подробной информации относительно того, откуда именно вызываются функции, следует использовать опцию `--trackcalls`.

```
$ python3 -m trace --listfuncs --trackcalls \
trace_example/main.py | grep -v importlib

This is the main program.
recurse(2)
recurse(1)
recurse(0)

calling relationships:

*** .../lib/python3.5/trace.py ***
    trace.Trace.runctx -> trace._unsettrace
  --> trace_example/main.py
    trace.Trace.runctx -> main.<module>

  --> trace_example/recurse.py


*** trace_example/main.py ***
    main.<module> -> main.main
  --> trace_example/recurse.py
    main.main -> recurse.recurse

*** trace_example/recurse.py ***
    recurse.recurse -> recurse.recurse
```

Примечание

Поскольку фильтры `--ignore-dir` и `--ignore-module` не оказывают воздействия на опции `--listfuncs` и `--trackcalls`, то для исключения ненужной части информации из вывода в этом примере используется команда `grep`.

16.4.5. Программный интерфейс

Вызов модуля `trace` из программного кода с помощью объекта `Trace` предоставляет более широкие возможности контроля над его интерфейсом. Объект `Trace` обеспечивает возможность настройки разделяемых контекстов и других зависимостей перед выполнением отслеживаемой функции или команды Python.

Листинг 16.43. `trace_run.py`

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.run('recurse(2)')
```

Поскольку в данном примере трассируется лишь выполнение функции `recurse()`, вывод не включает никакой информации относительно модуля `main.py`.

```
$ python3 trace_run.py

 --- modulename: trace_run, funcname: <module>
<string>(1):  --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(2)
recurse.py(13):     if level:
recurse.py(14):         recurse(level - 1)
 --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(1)
recurse.py(13):     if level:
recurse.py(14):         recurse(level - 1)
 --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(0)
recurse.py(13):     if level:
 --- modulename: trace, funcname: _unsettrace
trace.py(77):         sys.settrace(None)
```

Аналогичный вывод можно получить с помощью метода `runfunc()`.

Листинг 16.44. `trace_runfunc.py`

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.runfunc(recurse, 2)
```

Метод `runfunc()` получает произвольные позиционные и именованные аргументы, которые передаются функции объектом трассировки при ее вызове.

```
$ python3 trace_runfunc.py

 --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(2)
recurse.py(13):     if level:
recurse.py(14):         recurse(level - 1)
 --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(1)
recurse.py(13):     if level:
recurse.py(14):         recurse(level - 1)
 --- modulename: recurse, funcname: recurse
recurse.py(12):     print('recurse({})'.format(level))
recurse(0)
recurse.py(13):     if level:
```

16.4.6. Сохранение результатов

Как и при использовании интерфейса командной строки, информация о количестве вызовов функций и покрытии кода может быть записана в файл. Данные можно сохранить явным образом, используя экземпляр `CoverageResults` из объекта `Trace`.

Листинг 16.45. `trace_CoverageResults.py`

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True, trace=False)
tracer.runfunc(recurse, 2)

results = tracer.results()
results.write_results(coverdir='coverdir2')
```

В этом примере результаты сохраняются в каталоге `coverdir2`.

```
$ python3 trace_CoverageResults.py

recurse(2)
recurse(1)
recurse(0)

$ find coverdir2

coverdir2
coverdir2/trace_example.recurse.cover
```

Выходной файл содержит следующую информацию.

```
       #!/usr/bin/env python
       # encoding: utf-8
       #
       # Copyright (c) 2008 Doug Hellmann All rights reserved.
       #
       """
>>>>>> """

       #end_pymotw_header

>>>>>> def recurse(level):
    3:     print('recurse({})'.format(level))
    3:     if level:
    2:         recurse(level - 1)


>>>>>> def not_called():
>>>>>>     print('This function is never called.')
```

Чтобы сохранить данные о количестве вызовов функций для генерации отчетов, следует использовать аргументы `infile` и `outfile` при вызове конструктора `Trace`.

Листинг 16.46. `trace_report.py`

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True,
                     trace=False,
                     outfile='trace_report.dat')
tracer.runfunc(recurse, 2)

report_tracer = trace.Trace(count=False,
                            trace=False,
                            infile='trace_report.dat')
results = tracer.results()
results.write_results(summary=True, coverdir='/tmp')
```

Имя входного файла, из которого необходимо прочитать ранее сохраненные данные, задается с помощью аргумента `infile`, а имя выходного файла, в который необходимо записать новые результаты после выполнения трассировки, — с помощью аргумента `outfile`. В случае совпадения имен входного и выходного файлов предыдущий код обновляет файл кумулятивными данными.

```
$ python3 trace_report.py

recurse(2)
recurse(1)
recurse(0)
lines   cov%   module   (path)
    7    42%   trace_example.recurse
(.../trace_example/recurse.py)
```

16.4.7. Опции

Конструктор `Trace` поддерживает несколько необязательных параметров, позволяющих управлять поведением объекта во время выполнения.

- `count` — булево значение. Включает подсчет количества вызовов каждой инструкции. По умолчанию принимает значение `True`.
- `countfuncs` — булево значение. Включает вывод списка функций, вызываемых в процессе выполнения. По умолчанию принимает значение `False`.
- `countcallers` — булево значение. Включает отслеживание вызывающих и вызываемых объектов. По умолчанию принимает значение `True`.
- `ignoremods` — последовательность. Список модулей или пакетов, которые будут игнорироваться при отслеживании покрытия кода. Значением по умолчанию является пустой кортеж.
- `ignoredirs` — последовательность. Список каталогов, содержащих модули или пакеты, которые следует игнорировать. По умолчанию — пустой кортеж.

- `infile` – имя файла, содержащего кешированные значения счетчиков. По умолчанию принимает значение `None`.
- `outfile` – имя файла, используемого для сохранения кешированных значений счетчиков. По умолчанию принимает значение `None`, при котором данные не сохраняются.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `trace`[12].
- Раздел 17.2.7. Модуль `sys` включает средства для добавления пользовательской трассировочной функции в интерпретатор во время выполнения.
- `coverage.py` (Ned Batchelder)[13]. Модуль для определения покрытия кода.
- `figleaf` (Titus Brown)[14]. Приложение для определения покрытия кода.

16.5. `traceback`: исключения и стек вызовов

Модуль `traceback` обрабатывает информацию о стеке вызовов для вывода сообщений об ошибках. *Трассировочная информация* содержит данные о стеке вызовов, начиная с точки вызова обработчика исключения и далее до той точки в цепочке вызовов, в которой возникло исключение. Также возможен доступ к трассировочной информации о текущем стеке охватывающих вызовов (без контекста ошибки), что может быть пригодиться для определения путей выполнения, ведущих к вызову данной функции.

Высокоуровневый API модуля `traceback` сохраняет представление стека вызовов в экземплярах `StackSummary` и `FrameSummary`. Эти объекты могут конструироваться на основе трассировочного стека или текущего стека выполняемых вызовов и впоследствии обрабатываться одинаковым способом.

Низкоуровневые функции модуля `traceback` можно разбить на несколько категорий. Одни из них извлекают необработанную трассировочную информацию из текущей среды времени выполнения (либо обработчик исключений для стека вызовов, либо обычный стек). Извлеченный стек вызовов представляет собой последовательность кортежей, содержащих имя файла, номер строки, имя функции и текст строки исходного кода.

Извлеченный трассировочный объект можно форматировать с помощью таких функций, как `format_exception()` и `format_stack()`. Функции форматирования возвращают список строк с сообщениями, отформатированными для вывода на печать. Также доступны сокращенные функции, предназначенные для вывода форматированных значений.

Несмотря на то что функции, содержащиеся в модуле `traceback`, по умолчанию имитируют поведение интерактивного интерпретатора, они также могут обрабатывать исключения в ситуациях, когда дамп полного стека вызовов на консоль нежелателен. Например, веб-приложению может потребоваться форматировать трассировочную информацию в виде HTML-документа, или же IDE может

[12] `https://docs.python.org/3.5/library/trace.html`

[13] `http://nedbatchelder.com/code/modules/coverage.html`

[14] `http://darcs.idyll.org/~t/projects/figleaf/doc/`

понадобиться преобразовать элементы стека вызовов в интерактивный список, щелчки на элементах которого позволяют просматривать исходный код.

16.5.1. Вспомогательные функции

В примерах, приведенных в этом разделе, используется модуль `traceback_example.py`.

Листинг 16.47. `traceback_example.py`

```
import traceback
import sys


def produce_exception(recursion_level=2):
    sys.stdout.flush()
    if recursion_level:
        produce_exception(recursion_level - 1)
    else:
        raise RuntimeError()


def call_function(f, recursion_level=2):
    if recursion_level:
        return call_function(f, recursion_level - 1)
    else:
        return f()
```

16.5.2. Работа со стеком

Для работы с текущим стеком следует сконструировать объект `StackSummary` с помощью генератора фреймов на основе вспомогательной функции `walk_stack()`.

Листинг 16.48. `traceback_stacksummary.py`

```
import traceback
import sys

from traceback_example import call_function


def f():
    summary = traceback.StackSummary.extract(
        traceback.walk_stack(None)
    )
    print(''.join(summary.format()))


print('Calling f() directly:')
f()

print()
```

```
print('Calling f() from 3 levels deep:')
call_function(f)
```

Метод `format()` позволяет получить последовательность форматированных строк, подготовленных к выводу на печать.

```
$ python3 traceback_stacksummary.py

Calling f() directly:
  File "traceback_stacksummary.py", line 18, in f
    traceback.walk_stack(None)
  File "traceback_stacksummary.py", line 24, in <module>
    f()


Calling f() from 3 levels deep:
  File "traceback_stacksummary.py", line 18, in f
    traceback.walk_stack(None)
  File ".../traceback_example.py", line 26, in call_function
    return f()
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File "traceback_stacksummary.py", line 28, in <module>
    call_function(f)
```

Объект `StackSummary` — это итерируемый контейнер, хранящий экземпляры `FrameSummary`.

Листинг 16.49. `traceback_framesummary.py`

```
import traceback
import sys

from traceback_example import call_function

template = (
    '{fs.filename:<26}:{fs.lineno}:{fs.name}:\n'
    '    {fs.line}'
)


def f():
    summary = traceback.StackSummary.extract(
        traceback.walk_stack(None)
    )
    for fs in summary:
        print(template.format(fs=fs))


print('Calling f() directly:')
f()
```

```
print()
print('Calling f() from 3 levels deep:')
call_function(f)
```

Каждый объект `FrameSummary` описывает фрейм стека, включая местоположение контекста выполнения в файлах исходного кода программы.

```
$ python3 traceback_framesummary.py

Calling f() directly:
traceback_framesummary.py :23:f:
    traceback.walk_stack(None)
traceback_framesummary.py :30:<module>:
    f()

Calling f() from 3 levels deep:
traceback_framesummary.py :23:f:
    traceback.walk_stack(None)
.../traceback_example.py:26:call_function:
    return f()
.../traceback_example.py:24:call_function:
    return call_function(f, recursion_level - 1)
.../traceback_example.py:24:call_function:
    return call_function(f, recursion_level - 1)
traceback_framesummary.py :34:<module>:
    call_function(f)
```

16.5.3. Исключение `TracebackException`

Класс `TracebackException` — это высокоуровневый интерфейс для создания объектов `StackSummary` в процессе обработки трассировочной информации.

Листинг 16.50. `traceback_tracebackexception.py`

```
import traceback
import sys

from traceback_example import produce_exception

print('with no exception:')
exc_type, exc_value, exc_tb = sys.exc_info()
tbe = traceback.TracebackException(exc_type, exc_value, exc_tb)
print(''.join(tbe.format()))

print('\nwith exception:')
try:
    produce_exception()
except Exception as err:
    exc_type, exc_value, exc_tb = sys.exc_info()
    tbe = traceback.TracebackException(
        exc_type, exc_value, exc_tb,
    )
    print(''.join(tbe.format()))
```

```
    print('\nexception only:')
    print(''.join(tbe.format_exception_only()))
```

Метод `format()` создает форматированную версию полного объекта трассировки, в то время как метод `format_exception_only()` отображает лишь сообщение о возникновении исключения.

```
$ python3 traceback_tracebackexception.py

with no exception:
None


with exception:
Traceback (most recent call last):
  File "traceback_tracebackexception.py", line 22, in <module>
    produce_exception()
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 19, in produce_exception
    raise RuntimeError()
RuntimeError


exception only:
RuntimeError
```

16.5.4. Низкоуровневые программные интерфейсы исключений

Еще один способ обработки сообщений об исключениях предлагает функция `print_exc()`. Она использует функцию `sys.exc_info()` для получения информации об исключениях, возникших в текущем потоке, форматирует результаты и выводит текст в файл, заданный дескриптором (по умолчанию – файл `sys.stderr`).

Листинг 16.51. `traceback_print_exc.py`

```
import traceback
import sys

from traceback_example import produce_exception

print('print_exc() with no exception:')
traceback.print_exc(file=sys.stdout)
print()

try:
    produce_exception()
except Exception as err:
```

```
    print('print_exc():')
    traceback.print_exc(file=sys.stdout)
    print()
    print('print_exc(1):')
    traceback.print_exc(limit=1, file=sys.stdout)
```

В этом примере в качестве файла задан стандартный поток вывода `sys.stdout`, поэтому информация об исключении и трассировочная информация смешиваются корректным образом.

```
$ python3 traceback_print_exc.py

print_exc() with no exception:
NoneType

print_exc():
Traceback (most recent call last):
  File "traceback_print_exc.py", line 20, in <module>
    produce_exception()
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 19, in produce_exception
    raise RuntimeError()
RuntimeError

print_exc(1):
Traceback (most recent call last):
  File "traceback_print_exc.py", line 20, in <module>
    produce_exception()
RuntimeError
```

Функция `print_exc()` — это сокращенный вариант функции `print_exception()`, использование которой требует явного задания всех аргументов.

Листинг 16.52. `traceback_print_exception.py`

```
import traceback
import sys

from traceback_example import produce_exception

try:
    produce_exception()
except Exception as err:
    print('print_exception():')
    exc_type, exc_value, exc_tb = sys.exc_info()
    traceback.print_exception(exc_type, exc_value, exc_tb)
```

Аргументы для функции `print_exception()` получают с помощью функции `sys.exc_info()`.

```
$ python3 traceback_print_exception.py

Traceback (most recent call last):
  File "traceback_print_exception.py", line 16, in <module>
    produce_exception()
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 17, in produce_exception
    produce_exception(recursion_level - 1)
  File ".../traceback_example.py", line 19, in produce_exception
    raise RuntimeError()
RuntimeError
print_exception():
```

Для подготовки текста к выводу функция `print_exception()` использует функцию `format_exception()`.

Листинг 16.53. traceback_format_exception.py

```
import traceback
import sys
from pprint import pprint

from traceback_example import produce_exception

try:
    produce_exception()
except Exception as err:
    print('format_exception():')
    exc_type, exc_value, exc_tb = sys.exc_info()
    pprint(
        traceback.format_exception(exc_type, exc_value, exc_tb),
        width=65,
    )
```

Функции `format_exception()` передаются те же три аргумента: тип исключения (`exc_type`), значение исключения (`exc_value`) и трассировочная информация (`exc_tb`).

```
$ python3 traceback_format_exception.py

format_exception():
['Traceback (most recent call last):\n',
 '  File "traceback_format_exception.py", line 17, in
<module>\n'
 '    produce_exception()\n',
 '  File '
 '".../traceback_example.py", '
 'line 17, in produce_exception\n'
 '    produce_exception(recursion_level - 1)\n',
 '  File '
 '".../traceback_example.py", '
 'line 17, in produce_exception\n'
```

```
'    produce_exception(recursion_level - 1)\n',
'  File '
'".../traceback_example.py", '
'line 19, in produce_exception\n'
'    raise RuntimeError()\n',
'RuntimeError\n']
```

Чтобы обработать трассировочную информацию иным способом, например применить другое форматирование, следует использовать функцию `extract_tb()`, позволяющую извлечь данные в подходящем виде.

Листинг 16.54. `traceback_extract_tb.py`

```
import traceback
import sys
import os
from traceback_example import produce_exception

template = '{filename:<23}:{linenum}:{funcname}:\n    {source}'

try:
    produce_exception()
except Exception as err:
    print('format_exception():')
    exc_type, exc_value, exc_tb = sys.exc_info()
    for tb_info in traceback.extract_tb(exc_tb):
        filename, linenum, funcname, source = tb_info
        if funcname != '<module>':
            funcname = funcname + '()'
        print(template.format(
            filename=os.path.basename(filename),
            linenum=linenum,
            source=source,
            funcname=funcname)
        )
```

Возвращаемое значение — это список записей, которые соответствуют различным уровням стека, представленного объектом трассировки. Каждая запись представляет собой кортеж, включающий четыре элемента: имя исходного файла, номер строки в этом файле, имя функции и исходный код, находящийся в этой строке, с исключенными пробелами (если он доступен).

```
$ python3 traceback_extract_tb.py

format_exception():
traceback_extract_tb.py:18:<module>:
    produce_exception()
traceback_example.py   :17:produce_exception():
    produce_exception(recursion_level - 1)
traceback_example.py   :17:produce_exception():
    produce_exception(recursion_level - 1)
traceback_example.py   :19:produce_exception():
    raise RuntimeError()
```

16.5.5. Низкоуровневые программные интерфейсы стека

Аналогичный набор функций доступен для выполнения тех же операций по отношению к текущему стеку вызовов, а не к трассировочной информации. Функция `print_stack()` выводит текущий стек без возбуждения исключения.

Листинг 16.55. traceback_print_stack.py

```
import traceback
import sys

from traceback_example import call_function


def f():
    traceback.print_stack(file=sys.stdout)


print('Calling f() directly:')
f()

print()
print('Calling f() from 3 levels deep:')
call_function(f)
```

Вывод выглядит так же, как и трассировочная информация, но без сообщения об ошибке.

```
$ python3 traceback_print_stack.py

Calling f() directly:
  File "traceback_print_stack.py", line 21, in <module>
    f()
  File "traceback_print_stack.py", line 17, in f
    traceback.print_stack(file=sys.stdout)

Calling f() from 3 levels deep:
  File "traceback_print_stack.py", line 25, in <module>
    call_function(f)
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File ".../traceback_example.py", line 24, in call_function
    return call_function(f, recursion_level - 1)
  File ".../traceback_example.py", line 26, in call_function
    return f()
  File "traceback_print_stack.py", line 17, in f
    traceback.print_stack(file=sys.stdout)
```

Функция `format_stack()` подготавливает информацию о стеке вызовов точно так же, как функция `format_exception()` подготавливает к выводу трассировочную информацию.

Листинг 16.56. traceback_format_stack.py

```
import traceback
import sys
from pprint import pprint

from traceback_example import call_function


def f():
    return traceback.format_stack()


formatted_stack = call_function(f)
pprint(formatted_stack)
```

Данная функция возвращает список строк, каждая из которых образует одну строку вывода.

```
$ python3 traceback_format_stack.py

['  File "traceback_format_stack.py", line 21, in <module>\n'
 '    formatted_stack = call_function(f)\n',
 '  File "
 '".../traceback_example.py", '
 'line 24, in call_function\n'
 '    return call_function(f, recursion_level - 1)\n',
 '  File "
 '".../traceback_example.py", '
 'line 24, in call_function\n'
 '    return call_function(f, recursion_level - 1)\n',
 '  File "
 '".../traceback_example.py", '
 'line 26, in call_function\n'
 '    return f()\n',
 '  File "traceback_format_stack.py", line 18, in f\n'
 '    return traceback.format_stack()\n']
```

Функция `extract_stack()` работает подобно функции `extract_tb()`.

Листинг 16.57. traceback_extract_stack.py

```
import traceback
import sys
import os

from traceback_example import call_function

template = '{filename:<26}:{linenum}:{funcname}:\n    {source}'


def f():
    return traceback.extract_stack()
```

```
stack = call_function(f)
for filename, linenum, funcname, source in stack:
    if funcname != '<module>':
        funcname = funcname + '()'
    print(template.format(
        filename=os.path.basename(filename),
        linenum=linenum,
        source=source,
        funcname=funcname)
    )
```

Она также поддерживает аргументы (здесь не используются), позволяющие задавать кадр стека и ограничивать глубину извлечения записей.

```
$ python3 traceback_extract_stack.py

traceback_extract_stack.py:23:<module>:
    stack = call_function(f)
traceback_example.py :24:call_function():
    return call_function(f, recursion_level - 1)
traceback_example.py :24:call_function():
    return call_function(f, recursion_level - 1)
traceback_example.py :26:call_function():
    return f()
traceback_extract_stack.py:20:f():
    return traceback.extract_stack()
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `traceback`[15].
- `sys` (раздел 17.2). Модуль `sys` также включает одноэлементный кортеж для хранения текущего исключения.
- `inspect` (раздел 18.4). Модуль `inspect` включает дополнительные функции, предназначенные для исследования содержимого стека.
- `cgitb` (раздел 16.6). Еще один модуль, предназначенный для форматирования трассировочной информации.

16.6. `cgitb`: подробные отчеты о необработанных исключениях

Модуль `cgitb` — ценный инструмент отладки, предлагаемый стандартной библиотекой. Первоначально он предназначался для отображения информации об ошибках и отладочной информации в веб-приложениях. К сожалению, несмотря на то что позднее он был обновлен за счет включения возможности простого текстового вывода, он не был переименован. Как следствие, данный модуль используется реже, чем он того заслуживает, хотя позволяет получить более подробную трассировочную информацию по сравнению с модулем `traceback` (раздел 16.5).

[15] https://docs.python.org/3.5/library/traceback.html

16.6.1. Стандартные дампы трассировочной информации

При обработке исключений Python по умолчанию выводит в стандартный поток ошибок отчет о трассировке стека вплоть до позиции, в которой возникла ошибка. Этот базовый вывод часто содержит достаточно информации для того, чтобы идентифицировать причину возникновения исключения и устранить проблему.

Листинг 16.58. `cgitb_basic_traceback.py`

```
def func2(a, divisor):
    return a / divisor


def func1(a, b):
    c = b - 5
    return func2(a, c)

func1(1, 5)
```

В этом примере трудноуловимая ошибка содержится в функции `func2()`.

```
$ python3 cgitb_basic_traceback.py

Traceback (most recent call last):
  File "cgitb_basic_traceback.py", line 18, in <module>
    func1(1, 5)
  File "cgitb_basic_traceback.py", line 16, in func1
    return func2(a, c)
  File "cgitb_basic_traceback.py", line 11, in func2
    return a / divisor
ZeroDivisionError: division by zero
```

16.6.2. Активизация вывода подробной трассировочной информации

В то время как базовые трассировочные данные предоставляют достаточно информации для обнаружения ошибки, активизация модуля `cgitb` позволяет получить более подробные сведения. Модуль `cgitb` заменяет функцию `sys.excepthook()` функцией, обеспечивающей расширенную трассировочную информацию.

Листинг 16.59. `cgitb_local_vars.py`

```
import cgitb
cgitb.enable(format='text')
```

Выводимый в следующем примере отчет об ошибке намного информативнее предыдущего. Для каждого кадра стека выводится следующая информация:

- полный путь к файлу исходного кода, а не просто его базовое имя;
- значения аргументов для каждой функции в стеке;

- несколько строк с исходным кодом контекста, окружающего строку, в которой возникло исключение;
- значения переменных в выражении, ставшем причиной возникновения ошибки.

Доступ к переменным в трассировочном стеке исключения может облегчить нахождение логических ошибок, возникающих выше по стеку относительно строки, в которой фактически возбуждается исключение.

```
$ python3 cgitb_local_vars.py

ZeroDivisionError
Python 3.5.2: .../bin/python3
Thu Dec 29 09:30:37 2016

A problem occurred in a Python script. Here is the sequence of
function calls leading up to the error, in the order they
occurred.

 .../cgitb_local_vars.py in <module>()
   18 def func1(a, b):
   19     c = b - 5
   20     return func2(a, c)
   21
   22 func1(1, 5)
func1 = <function func1>

 .../cgitb_local_vars.py in func1(a=1, b=5)
   18 def func1(a, b):
   19     c = b - 5
   20     return func2(a, c)
   21
   22 func1(1, 5)
global func2 = <function func2>
a = 1
c = 0

 .../cgitb_local_vars.py in func2(a=1, divisor=0)
   13
   14 def func2(a, divisor):
   15     return a / divisor
   16
   17
a = 1
divisor = 0
ZeroDivisionError: division by zero
    __cause__ = None
    __class__ = <class 'ZeroDivisionError'>
    __context__ = None
    __delattr__ = <method-wrapper '__delattr__' of
    ZeroDivisionError object>
    __dict__ = {}
```

```
    __dir__ = <built-in method __dir__ of ZeroDivisionError
    object>
    __doc__ = 'Second argument to a division or modulo operation
    was zero.'
    __eq__ = <method-wrapper '__eq__' of ZeroDivisionError object>
    __format__ = <built-in method __format__ of ZeroDivisionError
    object>
    __ge__ = <method-wrapper '__ge__' of ZeroDivisionError object>
    __getattribute__ = <method-wrapper '__getattribute__' of
    ZeroDivisionError object>
    __gt__ = <method-wrapper '__gt__' of ZeroDivisionError object>
    __hash__ = <method-wrapper '__hash__' of ZeroDivisionError
    object>
    __init__ = <method-wrapper '__init__' of ZeroDivisionError
    object>
    __init_subclass__ = <built-in method __init_subclass__ of type
    object>
    __le__ = <method-wrapper '__le__' of ZeroDivisionError
    object>
    __lt__ = <method-wrapper '__lt__' of ZeroDivisionError
    object>
    __ne__ = <method-wrapper '__ne__' of ZeroDivisionError
    object>
    __new__ = <built-in method __new__ of type object>
    __reduce__ = <built-in method __reduce__ of
    ZeroDivisionError
    object>
    __reduce_ex__ = <built-in method __reduce_ex__ of
    ZeroDivisionError object>
    __repr__ = <method-wrapper '__repr__' of ZeroDivisionError
    object>
    __setattr__ = <method-wrapper '__setattr__' of
    ZeroDivisionError object>
    __setstate__ = <built-in method __setstate__ of
    ZeroDivisionError object>
    __sizeof__ = <built-in method __sizeof__ of ZeroDivisionError
    object>
    __str__ = <method-wrapper '__str__' of ZeroDivisionError
    object>
    __subclasshook__ = <built-in method __subclasshook__ of type
    object>
    __suppress_context__ = False
    __traceback__ = <traceback object>
    args = ('division by zero',)
    with_traceback = <built-in method with_traceback of
    ZeroDivisionError object>

The above is a description of an error in a Python program.
Here is the original traceback:

Traceback (most recent call last):
  File "cgitb_local_vars.py", line 22, in <module>
    func1(1, 5)
```

```
  File "cgitb_local_vars.py", line 20, in func1
    return func2(a, c)
  File "cgitb_local_vars.py", line 15, in func2
    return a / divisor
ZeroDivisionError: division by zero
```

В данном случае видно, что истинная причина появления исключения `ZeroDivisionError` связана с вычислением значения `c` в `func1()`, а не с использованием этого значения в `func2()`.

Завершающая часть вывода также включает подробное описание объекта исключения (если он имеет другие атрибуты помимо сообщения, которые могли бы пригодиться при отладке) и дамп трассировочной информации в первоначальной форме.

16.6.3. Локальные переменные в трассировочных стеках вызовов

Код модуля `cgitb`, отвечающий за проверку переменных в кадре стека, который привел к возникновению исключения, достаточно интеллектуален для того, чтобы вычислять и отображать атрибуты объектов.

Листинг 16.60. `cgitb_with_classes.py`

```
import cgitb
cgitb.enable(format='text', context=12)


class BrokenClass:
    """Этот класс содержит ошибку.
    """

    def __init__(self, a, b):
        """Будьте осторожны с передачей сюда аргументов.
        """
        self.a = a
        self.b = b
        self.c = self.a * self.b
        # Really
        # long
        # comment
        # goes
        # here.
        self.d = self.a / self.b
        return

o = BrokenClass(1, 0)
```

Если функция или метод включает множество встроенных комментариев, пробелов или другого кода, занимающего много строк, то заданных по умолчанию пяти строк контекста может оказаться недостаточно для того, чтобы сориентироваться в причинах возникновения ошибки. Если тело функции выходит за рамки окна кода и его не видно на экране, то доступного контекста не хватит для того,

чтобы локализовать ошибку. Использование более протяженного контекста позволяет решить проблему. Размером кода, отображаемым для каждой строки трассировочной информации, можно управлять с помощью целочисленного аргумента `context`, передаваемого функции `enable()`.

Как следует из приведенного ниже вывода, возникновение ошибки связано с атрибутами `self.a` и `self.b`.

```
$ python3 cgitb_with_classes.py

ZeroDivisionError
Python 3.5.2: .../bin/python3
Thu Dec 29 09:30:37 2016

A problem occurred in a Python script. Here is the sequence of
function calls leading up to the error, in the order they
occurred.

 .../cgitb_with_classes.py in <module>()
   21         self.a = a
   22         self.b = b
   23         self.c = self.a * self.b
   24         # Really
   25         # long
   26         # comment
   27         # goes
   28         # here.
   29         self.d = self.a / self.b
   30         return
   31
   32 o = BrokenClass(1, 0)
o undefined
BrokenClass = <class '__main__.BrokenClass'>

 .../cgitb_with_classes.py in
 __init__(self=<__main__.BrokenClass object>, a=1, b=0)
   21         self.a = a
   22         self.b = b
   23         self.c = self.a * self.b
   24         # Really
   25         # long
   26         # comment
   27         # goes
   28         # here.
   29         self.d = self.a / self.b
   30         return
   31
   32 o = BrokenClass(1, 0)
self = <__main__.BrokenClass object>
self.d undefined
self.a = 1
self.b = 0
ZeroDivisionError: division by zero
```

```
    __cause__ = None
    __class__ = <class 'ZeroDivisionError'>
    __context__ = None
    __delattr__ = <method-wrapper '__delattr__' of
    ZeroDivisionError object>
    __dict__ = {}
    __dir__ = <built-in method __dir__ of ZeroDivisionError
    object>
    __doc__ = 'Second argument to a division or modulo operation
    was zero.'
    __eq__ = <method-wrapper '__eq__' of ZeroDivisionError
    object>
    __format__ = <built-in method __format__ of ZeroDivisionError
    object>
    __ge__ = <method-wrapper '__ge__' of ZeroDivisionError object>
    __getattribute__ = <method-wrapper '__getattribute__' of
    ZeroDivisionError object>
    __gt__ = <method-wrapper '__gt__' of ZeroDivisionError
    object>
    __hash__ = <method-wrapper '__hash__' of ZeroDivisionError
    object>
    __init__ = <method-wrapper '__init__' of ZeroDivisionError
    object>
    __le__ = <method-wrapper '__le__' of ZeroDivisionError
    object>
    __lt__ = <method-wrapper '__lt__' of ZeroDivisionError
    object>
    __ne__ = <method-wrapper '__ne__' of ZeroDivisionError
    object>
    __new__ = <built-in method __new__ of type object>
    __reduce__ = <built-in method __reduce__ of
    ZeroDivisionError object>
    __reduce_ex__ = <built-in method __reduce_ex__ of
    ZeroDivisionError object>
    __repr__ = <method-wrapper '__repr__' of ZeroDivisionError
    object>
    __setattr__ = <method-wrapper '__setattr__' of
    ZeroDivisionError object>
    __setstate__ = <built-in method __setstate__ of
    ZeroDivisionError object>
    __sizeof__ = <built-in method __sizeof__ of
    ZeroDivisionError object>
    __str__ = <method-wrapper '__str__' of ZeroDivisionError
    object>
    __subclasshook__ = <built-in method __subclasshook__ of type
    object>
    __suppress_context__ = False
    __traceback__ = <traceback object>
    args = ('division by zero',)
    with_traceback = <built-in method with_traceback of
    ZeroDivisionError object>

The above is a description of an error in a Python program.
```

```
Here is
the original traceback:

Traceback (most recent call last):
  File "cgitb_with_classes.py", line 32, in <module>
    o = BrokenClass(1, 0)
  File "cgitb_with_classes.py", line 29, in __init__
    self.d = self.a / self.b
ZeroDivisionError: division by zero
```

16.6.4. Свойства объекта исключения

Кроме отображения атрибутов локальных переменных из каждого фрейма стека модуль cgitb отображает все свойства объекта исключения. Дополнительные свойства пользовательских исключений выводятся как часть отчета об ошибке.

Листинг 16.61. `cgitb_exception_properties.py`

```
import cgitb
cgitb.enable(format='text')


class MyException(Exception):
    """Добавляет расширенные свойства к заданному исключению
    """

    def __init__(self, message, bad_value):
        self.bad_value = bad_value
        Exception.__init__(self, message)
        return

raise MyException('Normal message', bad_value=99)
```

В данном примере значение свойства bad_value включается в отчет вместе со стандартными значениями message и args.

```
$ python3 cgitb_exception_properties.py

MyException
Python 3.5.2: .../bin/python3
Thu Dec 29 09:30:37 2016

A problem occurred in a Python script. Here is the sequence of
function calls leading up to the error, in the order they
occurred.

 .../cgitb_exception_properties.py in <module>()
   19         self.bad_value = bad_value
   20         Exception.__init__(self, message)
   21         return
   22
   23 raise MyException('Normal message', bad_value=99)
MyException = <class '__main__.MyException'>
```

```
bad_value undefined
MyException: Normal message
    __cause__ = None
    __class__ = <class '__main__.MyException'>
    __context__ = None
    __delattr__ = <method-wrapper '__delattr__' of MyException
    object>
    __dict__ = {'bad_value': 99}
    __dir__ = <built-in method __dir__ of MyException object>
    __doc__ = 'Add extra properties to a special exception\n
    '
    __eq__ = <method-wrapper '__eq__' of MyException object>
    __format__ = <built-in method __format__ of MyException
    object>
    __ge__ = <method-wrapper '__ge__' of MyException object>
    __getattribute__ = <method-wrapper '__getattribute__' of
    MyException object>
    __gt__ = <method-wrapper '__gt__' of MyException object>
    __hash__ = <method-wrapper '__hash__' of MyException object>
    __init__ = <bound method MyException.__init__ of
    MyException('Normal message',)>
    __le__ = <method-wrapper '__le__' of MyException object>
    __lt__ = <method-wrapper '__lt__' of MyException object>
    __module__ = '__main__'
    __ne__ = <method-wrapper '__ne__' of MyException object>
    __new__ = <built-in method __new__ of type object>
    __reduce__ = <built-in method __reduce__ of MyException
    object>
    __reduce_ex__ = <built-in method __reduce_ex__ of MyException
    object>
    __repr__ = <method-wrapper '__repr__' of MyException object>
    __setattr__ = <method-wrapper '__setattr__' of MyException
    object>
    __setstate__ = <built-in method __setstate__ of MyException
    object>
    __sizeof__ = <built-in method __sizeof__ of MyException
    object>
    __str__ = <method-wrapper '__str__' of MyException object>
    __subclasshook__ = <built-in method __subclasshook__ of type
    object>
    __suppress_context__ = False
    __traceback__ = <traceback object>
    __weakref__ = None
    args = ('Normal message',)
    bad_value = 99
    with_traceback = <built-in method with_traceback of
    MyException object>

The above is a description of an error in a Python program.
Here is
the original traceback:

Traceback (most recent call last):
```

```
  File "cgitb_exception_properties.py", line 23, in <module>
    raise MyException('Normal message', bad_value=99)
MyException: Normal message
```

16.6.5. Вывод в формате HTML

Поскольку модуль `cgitb` первоначально разрабатывался для обработки исключений в веб-приложениях, обсуждение будет неполным, если не сказать хотя бы несколько слов о его первоначальном формате вывода HTML. Во всех предыдущих примерах вывод представлял собой простой текст. Для получения вывода в формате HTML следует опустить аргумент `format` (или указать для него значение `"html"`). Большинство современных веб-приложений создается с помощью какого-либо фреймворка, который включает средства вывода сообщений об ошибках, поэтому использование HTML-формы вывода в значительной степени устарело.

16.6.6. Запись трассировочной информации в журнал

Во многих ситуациях вывод подробной трассировочной информации в стандартный поток ошибок является наилучшим решением. Однако в производственной среде ошибки лучше записывать в журнал. Функция `enable()` поддерживает необязательный аргумент `logdir`, определяющий каталог, в который должны записываться журналы с отчетами. В случае предоставления имени каталога каждый отчет о необработанном исключении будет записываться в собственный файл, помещаемый в данный каталог.

Листинг 16.62. `cgitb_log_exception.py`

```
import cgitb
import os

LOGDIR = os.path.join(os.path.dirname(__file__), 'LOGS')

if not os.path.exists(LOGDIR):
    os.makedirs(LOGDIR)

cgitb.enable(
    logdir=LOGDIR,
    display=False,
    format='text',
)


def func(a, divisor):
    return a / divisor

func(1, 0)
```

Несмотря на то что вывод отчетов об ошибках отключен с помощью аргумента `display`, на экран выводится сообщение с указанием местонахождения журнала отчета.

```
$ python3 cgitb_log_exception.py

<p>A problem occurred in a Python script.
.../LOGS/tmptxqq_6yx.txt contains the description of this error.

$ ls LOGS

tmptxqq_6yx.txt

$ cat LOGS/*.txt

ZeroDivisionError
Python 3.5.2: .../bin/python3
Thu Dec 29 09 30:38 2016

A problem occurred in a Python script.  Here is the sequence of
function calls leading up to the error, in the order they occurred.

 .../cgitb_log_exception.py in <module>()
   24
   25 def func(a, divisor):
   26     return a / divisor
   27
   28 func(1, 0)
func = <function func>

 .../cgitb_log_exception.py in func(a=1, divisor=0)
   24
   25 def func(a, divisor):
   26     return a / divisor
   27
   28 func(1, 0)
a = 1
divisor = 0
ZeroDivisionError: division by zero
     __cause__ = None
     __class__ = <class 'ZeroDivisionError'>
     __context__ = None
     __delattr__ = <method-wrapper '__delattr__' of
     ZeroDivisionError object>
     __dict__ = {}
     __dir__ = <built-in method __dir__ of ZeroDivisionError
     object>
     __doc__ = 'Second argument to a division or modulo operation
     was zero.'
     __eq__ = <method-wrapper '__eq__' of ZeroDivisionError object>
     __format__ = <built-in method __format__ of ZeroDivisionError
     object>
     __ge__ = <method-wrapper '__ge__' of ZeroDivisionError object>
     __getattribute__ = <method-wrapper '__getattribute__' of
     ZeroDivisionError object>
     __gt__ = <method-wrapper '__gt__' of ZeroDivisionError object>
     __hash__ = <method-wrapper '__hash__' of ZeroDivisionError
```

```
    object>
    __init__ = <method-wrapper '__init__' of ZeroDivisionError
    object>
    __init_subclass__ = <built-in method __init_subclass__ of type
    object>
    __le__ = <method-wrapper '__le__' of ZeroDivisionError object>
    __lt__ = <method-wrapper '__lt__' of ZeroDivisionError object>
    __ne__ = <method-wrapper '__ne__' of ZeroDivisionError object>
    __new__ = <built-in method __new__ of type object>
    __reduce__ = <built-in method __reduce__ of ZeroDivisionError
    object>
    __reduce_ex__ = <built-in method __reduce_ex__ of
    ZeroDivisionError object>
    __repr__ = <method-wrapper '__repr__' of ZeroDivisionError
    object>
    __setattr__ = <method-wrapper '__setattr__' of
    ZeroDivisionError object>
    __setstate__ = <built-in method __setstate__ of
    ZeroDivisionError object>
    __sizeof__ = <built-in method __sizeof__ of ZeroDivisionError
    object>
    __str__ = <method-wrapper '__str__' of ZeroDivisionError
    object>
    __subclasshook__ = <built-in method __subclasshook__ of type
    object>
    __suppress_context__ = False
    __traceback__ = <traceback object>
    args = ('division by zero',)
    with_traceback = <built-in method with_traceback of
    ZeroDivisionError object>

The above is a description of an error in a Python program.
Here is
the original traceback:

Traceback (most recent call last):
  File "cgitb_log_exception.py", line 28, in <module>
    func(1, 0)
  File "cgitb_log_exception.py", line 26, in func
    return a / divisor
ZeroDivisionError: division by zero
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `cgitb`[16].
- `traceback` (раздел 16.5). Модуль стандартной библиотеки, предназначенный для работы с трассировочной информацией.
- `inspect` (раздел 18.4). Модуль `inspect` включает дополнительные функции, предназначенные для изучения содержимого стека.

[16] https://docs.python.org/3.5/library/cgitb.html

- sys (раздел 17.2). Модуль sys предоставляет доступ к значению текущего исключения и обработчику, который вызывается при возбуждении данного исключения.
- Улучшенный модуль traceback[17]. Обсуждение в списке рассылки для разработчиков на языке Python, касающееся усовершенствования модуля traceback и родственных улучшений, локально используемых другими разработчиками.

16.7. pdb: интерактивный отладчик

Модуль pdb реализует интерактивную среду отладки для программ на языке Python. Он поддерживает приостановку выполнения программы, просмотр значений переменных и выполнение программы в пошаговом режиме, тем самым предоставляя возможность разобраться в том, что именно делает код, и найти ошибки в логике работы программы.

16.7.1. Запуск отладчика

Первое, что необходимо сделать, прежде чем приступить к работе с модулем pdb, — это заставить интерпретатор в нужный момент перейти в режим отладки. В зависимости от условий запуска отладчика и объекта отладки это можно сделать несколькими способами.

16.7.1.1. Вызов отладчика из командной строки

Самым непосредственным способом использования отладчика является его вызов из командной строки с указанием программы, подлежащей выполнению.

Листинг 16.63. pdb_script.py

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann. All rights reserved.
#


class MyObj:

    def __init__(self, num_loops):
        self.count = num_loops

    def go(self):
        for i in range(self.count):
            print(i)
        return

if __name__ == '__main__':
    MyObj(5).go()
```

Запущенный из командной строки отладчик загружает указанный файл с исходным кодом и прекращает выполнение, как только встречается первая инструк-

[17] https://lists.gt.net/python/dev/802870

ция. В данном случае он останавливается перед обработкой определения класса MyObj на строке 8.

```
$ python3 -m pdb pdb_script.py

> .../pdb_script.py(8)<module>()
-> class MyObj(object):
(Pdb)
```

Примечание

Обычно команда pdb включает полный путь к каждому модулю при выводе имени файла. С целью упрощения примеров, приведенных в этом разделе, пути в выводе заменяются многоточиями (...).

16.7.1.2. Вызов отладчика из интерпретатора

В процессе разработки ранних версий модулей многие разработчики используют интерактивную оболочку интерпретатора, обеспечивающую более удобный итеративный способ выполнения команд, не требующий применения циклов типа "сохранить/выполнить/повторить", которые приходится использовать в автономных сценариях. Для вызова отладчика из интерактивного интерпретатора следует использовать функцию run() или runeval().

```
$ python3

Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> import pdb_script
>>> import pdb
>>> pdb.run('pdb_script.MyObj(5).go()')
> <string>(1)<module>()
(Pdb)
```

Аргументом функции run() служит строковое выражение, которое может быть вычислено интерпретатором Python. Отладчик анализирует его, а затем приостанавливает выполнение непосредственно перед вычислением первого выражения. Для навигации по коду и управления его выполнением можно использовать описанные здесь инструкции.

16.7.1.3. Вызов отладчика из программы

В обоих предыдущих примерах отладчик запускался в начале программы. Однако в случае длительно выполняющихся процессов, когда проблема может проявиться на поздних стадиях выполнения, гораздо удобнее запускать отладчик из программы с помощью функции set_trace().

Листинг 16.64. pdb_set_trace.py

```
1 #!/usr/bin/env python3
2 # encoding: utf-8
```

```
 3 #
 4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
 5 #
 6
 7 import pdb
 8
 9
10 class MyObj:
11
12     def __init__(self, num_loops):
13         self.count = num_loops
14
15     def go(self):
16         for i in range(self.count):
17             pdb.set_trace()
18             print(i)
19         return
20
21 if __name__ == '__main__':
22     MyObj(5).go()
```

В этом сценарии отладчик запускается в строке 17, приостанавливая выполнение программы на строке 18.

```
$ python3 ./pdb_set_trace.py

> .../pdb_set_trace.py(18)go()
-> print(i)
(Pdb)
```

Функция `set_trace()` — обычная функция Python, которая может быть вызвана в любой точке программы. Отсюда следует, что вход в отладчик может осуществляться на основании некоторых условий, в том числе посредством обработчика исключений или определенной ветви условной инструкции.

16.7.1.4. Поставарийный вызов отладчика

Отладка программы после ее аварийного завершения называется *поставарийной*. Модуль `pdb` поддерживает поставарийную отладку посредством функций `pm()` и `post_mortem()`.

Листинг 16.65. `pdb_post_mortem.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3 #
 4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
 5 #
 6
 7
 8 class MyObj:
 9
10     def __init__(self, num_loops):
11         self.count = num_loops
```

```
12
13      def go(self):
14          for i in range(self.num_loops):
15              print(i)
16          return
```

В этом примере попытка использовать некорректное имя атрибута в строке 14 приводит к возбуждению исключения `AttributeError`, в результате чего выполнение программы прекращается. Функция `pm()` ищет активный объект `traceback`, содержащий трассировочную информацию, и запускает отладчик в той точке стека вызовов, в которой возникло исключение.

```
$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> from pdb_post_mortem import MyObj
>>> MyObj(5).go()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../pdb_post_mortem.py", line 14, in go
    for i in range(self.num_loops):
AttributeError: 'MyObj' object has no attribute 'num_loops'
>>> import pdb
>>> pdb.pm()
> .../pdb/pdb_post_mortem.py(14)go()
-> for i in range(self.num_loops):
(Pdb)
```

16.7.2. Управление отладчиком

Интерфейс для работы с отладчиком — это язык команд, позволяющих перемещаться по стеку вызовов, исследовать и изменять значения переменных, а также выполнять программу под управлением отладчика. Интерактивный отладчик использует модуль `readline` (см. раздел 14.3) для получения команд и поддерживает завершение команд, имен файлов и функций с помощью клавиши <Tab>. Ввод пустой строки означает повторное выполнение предыдущей команды, если только она не представляет собой операцию над списком.

16.7.2.1. Перемещение по стеку вызовов

Когда запущен отладчик, команда `where` (сокращенно — `w`) обеспечивает точное определение выполняющейся в данный момент строки кода и соответствующей позиции в стеке вызовов. В данном случае выполнение программы останавливается в строке 18 метода `go()` модуля `pdb_set_trace.py`.

```
$ python3 pdb_set_trace.py
> .../pdb_set_trace.py(18)go()
-> print(i)
(Pdb) where
  .../pdb_set_trace.py(22)<module>()
```

```
-> MyObj(5).go()
> .../pdb_set_trace.py(18)go()
-> print(i)
(Pdb)
```

Чтобы добавить контекст, окружающий текущую позицию в коде, следует использовать команду `list (l)`.

```
(Pdb) l
 13              self.count = num_loops
 14
 15          def go(self):
 16              for i in range(self.count):
 17                  pdb.set_trace()
 18  ->              print(i)
 19              return
 20
 21      if __name__ == '__main__':
 22          MyObj(5).go()
[EOF]
(Pdb)
```

По умолчанию выводятся 11 строк исходного программного кода (текущая строка и по 5 строк до и после нее). Команда `list` с одним числовым аргументом задает вывод 11 строк программного кода, начиная с указанной строки.

```
(Pdb) list 14
  9
 10      class MyObj:
 11
 12          def __init__(self, num_loops):
 13              self.count = num_loops
 14
 15          def go(self):
 16              for i in range(self.count):
 17                  pdb.set_trace()
 18  ->              print(i)
 19              return
```

В случае предоставления команде `list` двух аргументов они интерпретируются соответственно как первая и последняя строка диапазона строк, включаемых в вывод.

```
(Pdb) list 7, 19
  7      import pdb
  8
  9
 10      class MyObj:
 11
 12          def __init__(self, num_loops):
 13              self.count = num_loops
 14
 15          def go(self):
```

```
 16              for i in range(self.count):
 17                  pdb.set_trace()
 18  ->              print(i)
 19              return
```

Команда `longlist` (`ll`) выводит исходный код текущей функции или фрейма без указания номеров строк. Ее название "longlist" ("длинный список") объясняется тем, что размер ее вывода может значительно превышать тот, который предусмотрен по умолчанию для команды `list`.

```
(Pdb) longlist
 15          def go(self):
 16              for i in range(self.count):
 17                  pdb.set_trace()
 18  ->              print(i)
 19              return
```

Команда `source` загружает и выводит на экран весь исходный программный код для произвольного класса, функции или модуля.

```
(Pdb) source MyObj
 10      class MyObj:
 11
 12          def __init__(self, num_loops):
 13              self.count = num_loops
 14
 15          def go(self):
 16              for i in range(self.count):
 17                  pdb.set_trace()
 18                  print(i)
 19              return
```

Для перемещения между фреймами в пределах текущего стека вызовов используются команды `up` и `down`. Команда `up` (сокращенно — `u`) выполняет перемещение по стеку в направлении к более ранним фреймам, команда `down` (`d`) — к более поздним фреймам. При каждом перемещении вверх или вниз по стеку отладчик выводит текущий фрейм в том же формате, что и при выводе с помощью команды `where`.

```
(Pdb) up
> .../pdb_set_trace.py(22)<module>()
-> MyObj(5).go()

(Pdb) down
> .../pdb_set_trace.py(18)go()
-> print(i)
```

Числовой аргумент, передаваемый любой из команд `up` и `down`, указывает, на какое количество уровней следует переместиться в соответствующем направлении за один раз.

16.7.2.2. Исследование переменных в стеке

Каждый фрейм в стеке поддерживает набор переменных, включая как локальные переменные выполняемой функции, так и глобальную информацию о состоянии. Модуль pdb предоставляет несколько способов исследования содержимого этих переменных.

Листинг 16.66. pdb_function_arguments.py

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#

import pdb


def recursive_function(n=5, output='to be printed'):
    if n > 0:
        recursive_function(n - 1)
    else:
        pdb.set_trace()
        print(output)
    return

if __name__ == '__main__':
    recursive_function()
```

Команда args (сокращено — a) выводит все аргументы функции, активные в текущем фрейме. Кроме того, чтобы продемонстрировать, как выглядит более глубокий стек при выводе его содержимого, в этом примере используется рекурсивная функция.

```
$ python3 pdb_function_arguments.py
> .../pdb_function_arguments.py(15)recursive_function()
-> print(output)
(Pdb) where
  .../pdb_function_arguments.py(19)<module>()
-> recursive_function()
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
> .../pdb_function_arguments.py(15)recursive_function()
-> print(output)
(Pdb) args
```

```
n = 0
output = to be printed

(Pdb) up
> .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)

(Pdb) args
n = 1
output = to be printed
```

Команда `p` вычисляет выражение, предоставленное в качестве аргумента, и выводит результат. Доступна также функция `print()`, но вместо того чтобы выполняться в качестве команды отладчика, она передается для выполнения интерпретатору.

```
(Pdb) p n
1

(Pdb) print(n)
1
```

Аналогичным образом, если выражению предшествует символ `!`, оно передается для вычисления интерпретатору Python. Это можно использовать для вычисления произвольных выражений Python, включая изменение переменных. В следующем примере, перед тем как позволить отладчику продолжить выполнение программы, изменяется значение переменной `output`. Инструкция, которая следует за вызовом `set_trace()`, выводит на печать переменную `output`, отображая ее измененное значение.

```
$ python3 pdb_function_arguments.py

> .../pdb_function_arguments.py(14)recursive_function()
-> print(output)

(Pdb) !output
'to be printed'

(Pdb) !output='changed value'

(Pdb) continue
changed value
```

В случае более сложных значений, таких как вложенные или крупные структуры данных, для их вывода следует использовать команду "красивой" печати `pp`. В следующей программе выполняется чтение нескольких строк из файла.

Листинг 16.67. `pdb_pp.py`

```
1 #!/usr/bin/env python3
2 # encoding: utf-8
3 #
4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
```

```
 5 #
 6
 7 import pdb
 8
 9 with open('lorem.txt', 'rt') as f:
10     lines = f.readlines()
11
12 pdb.set_trace()
```

Вывод переменной `lines`, получаемый с помощью команды `p`, трудно поддается чтению из-за того, что переход текста на следующую строку может осуществляться неподходящим образом. Команда `pp` использует модуль `pprint` (см. раздел 2.10), форматирующий значения для получения более аккуратного вывода.

```
$ python3 pdb_pp.py

> .../pdb_pp.py(12)<module>()->None
-> pdb.set_trace()
(Pdb) p lines
['Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
\n', 'Donec egestas, enim et consecte tuer ullamcorper, lect
us \n', 'ligula rutrum leo, a elementum el it tortor eu quam
.\n']

(Pdb) pp lines
['Lorem ipsum dolor sit amet, consectetuer adipiscing elit. \n',
 'Donec egestas, enim et consectetuer ullamcorper, lectus \n',
 'ligula rutrum leo, a elementum elit tortor eu quam.\n']
(Pdb)
```

Если требуется интерактивно исследовать содержимое переменных и выполнять эксперименты, можно выйти из отладчика и перейти в стандартный интерактивный режим Python, в котором глобальные переменные, а также локальные переменные из текущего фрейма уже будут заполнены значениями.

```
$ python3 -m pdb pdb_interact.py
> .../pdb_interact.py(7)<module>()
-> import pdb
(Pdb) break 14
Breakpoint 1 at .../pdb_interact.py:14

(Pdb) continue
> .../pdb_interact.py(14)f()
-> print(l, m, n)

(Pdb) p l
['a', 'b']

(Pdb) p m
9

(Pdb) p n
5
```

```
(Pdb) interact
*interactive*

>>> l
['a', 'b']

>>> m
9

>>> n
5
```

Изменяемые объекты, такие как списки, могут быть изменены в интерактивном интерпретаторе. В отличие от этого неизменяемые объекты нельзя изменить, а имена нельзя повторно связать с новыми значениями.

```
>>> l.append('c')
>>> m += 7
>>> n = 3

>>> l
['a', 'b', 'c']

>>> m
16

>>> n
3
```

Для выхода из режима интерактивного интерпретатора и возврата в отладчик следует ввести символ конца файла с помощью комбинации клавиш <Ctrl-D>. В данном примере список `l` был изменен, однако значения переменных `m` и `n` остались прежними.

```
>>> ^D

(Pdb) p l
['a', 'b', 'c']

(Pdb) p m
9

(Pdb) p n
5

(Pdb)
```

16.7.2.3. Пошаговое выполнение программы

Кроме перемещения вверх и вниз по стеку вызовов в то время, когда выполнение программы приостановлено, существует возможность пошагового выполнения программы с той точки, в которой был выполнен вход в отладчик.

Листинг 16.68. pdb_step.py

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann. All rights reserved.
#

import pdb


def f(n):
    for i in range(n):
        j = i * n
        print(i, j)
    return

if __name__ == '__main__':
    pdb.set_trace()
    f(5)
```

Команда step (сокращенная версия — s) выполняет текущую строку исходного кода и приостанавливает программу в следующей точке выполнения, будь то первая инструкция в вызываемой функции или следующая строка текущей функции.

```
$ python3 pdb_step.py

> .../pdb_step.py(18)<module>()
-> f(5)
```

После каждого вызова функции set_trace() интерпретатор приостанавливает выполнение и передает управление отладчику. Результатом выполнения первой команды step является вход в функцию f().

```
(Pdb) step
--Call--
> .../pdb_step.py(10)f()
-> def f(n):
```

Следующая команда step перемещает точку выполнения в первую строку f() и запускает цикл.

```
(Pdb) step
> .../pdb_step.py(11)f()
-> for i in range(n):
```

Повторное выполнение команды step перемещает точку выполнения в первую строку внутри цикла, в которой определяется переменная j.

```
(Pdb) step
> .../pdb_step.py(12)f()
-> j = i * n
```

```
(Pdb) p i
0
```

Значение переменной `i` равно `0`, поэтому после выполнения еще одного шага значение `j` должно быть равным `0`.

```
(Pdb) step
> .../pdb_step.py(13)f()
-> print(i, j)

(Pdb) p j
0

(Pdb)
```

Выполнение программы по одной строке за раз описанным способом может стать утомительным, если точке программы, в которой происходит ошибка, предшествует большое количество строк кода или одна и та же функция вызывается многократно.

Листинг 16.69. `pdb_next.py`

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann. All rights reserved.
#

import pdb


def calc(i, n):
    j = i * n
    return j


def f(n):
    for i in range(n):
        j = calc(i, n)
        print(i, j)
    return

if __name__ == '__main__':
    pdb.set_trace()
    f(5)
```

В этом примере в функции `calc()` ничего не может случиться. Таким образом, ее пошаговое выполнение в цикле функции `f()` затруднило бы чтение полезного вывода, поскольку в этом случае все строки функции `calc()` отображались бы по мере их выполнения.

```
$ python3 pdb_next.py

> .../pdb_next.py(23)<module>()
```

```
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) step
> .../pdb_next.py(17)f()
-> j = calc(i, n)

(Pdb) step
--Call--
> .../pdb_next.py(10)calc()
-> def calc(i, n):

(Pdb) step
> .../pdb_next.py(11)calc()
-> j = i * n

(Pdb) step
> .../pdb_next.py(12)calc()
-> return j

(Pdb) step
--Return--
> .../pdb_next.py(12)calc()->0
-> return j

(Pdb) step
> .../pdb_next.py(18)f()
-> print(i, j)

(Pdb) step
0 0

> .../pdb_next.py(16)f()
-> for i in range(n):
(Pdb)
```

Команда `next` (сокращенная версия — `n`) подобна команде `step`, но не входит в тело функций, вызываемых из выполняемых инструкций. В конечном счете она выполняет весь вызов функции до следующей инструкции в текущей функции как одну операцию.

```
> .../pdb_next.py(16)f()
-> for i in range(n):
(Pdb) step
> .../pdb_next.py(17)f()
-> j = calc(i, n)
```

```
(Pdb) next
> .../pdb_next.py(18)f()
-> print(i, j)

(Pdb)
```

Команда `until` аналогична команде `next`, за исключением того, что она продолжает выполнение программы до тех пор, пока поток выполнения не покинет текущий фрейм стека или не будет достигнута строка, номер которой превышает номер текущей строки. Это, например, означает, что команду `until` можно использовать для выполнения всех команд цикла и приостановки выполнения на следующей строке.

```
$ python3 pdb_next.py

> .../pdb_next.py(23)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) step
> .../pdb_next.py(17)f()
-> j = calc(i, n)

(Pdb) next
> .../pdb_next.py(18)f()
-> print(i, j)

(Pdb) until
0 0
1 5
2 10
3 15
4 20
> .../pdb_next.py(19)f()
-> return

(Pdb)
```

До выполнения команды `until` текущей строкой было строка 18, последняя строка цикла. После выполнения команды `until` точка выполнения находится в строке 19, когда цикл уже завершился.

Чтобы позволить программе выполняться до тех пор, пока не будет достигнута определенная строка, следует передать номер строки команде `until`. В отличие от задания точки останова, номер строки, передаваемый команде `until`, должен превышать текущий номер строки, поэтому данная команда оказывается наиболее полезной для пропуска длинных блоков кода в пределах функции.

```
$ python3 pdb_next.py
> .../pdb_next.py(23)<module>()
-> f(5)
(Pdb) list
 18             print(i, j)
 19         return
 20
 21     if __name__ == '__main__':
 22         pdb.set_trace()
 23  ->     f(5)
[EOF]

(Pdb) until 18
*** "until" line number is smaller than current line number

(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) list
 11      j = i * n
 12      return j
 13
 14
 15  def f(n):
 16  ->    for i in range(n):
 17          j = calc(i, n)
 18          print(i, j)
 19      return
 20
 21  if __name__ == '__main__':
(Pdb) until 19
0 0
1 5
2 10
3 15
4 20
> .../pdb_next.py(19)f()
-> return

(Pdb)
```

Команда `return` — еще один полезный способ быстрого прохождения части функции. Она продолжает выполнение до тех пор, пока функция не будет готова выполнить инструкцию `return`. В этот момент выполнение программы приостанавливается, предоставляя возможность проверить возвращаемое значение, прежде чем будет выполнен возврат из функции.

```
$ python3 pdb_next.py

> .../pdb_next.py(23)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) return
0 0
1 5
2 10
3 15
4 20
--Return--
> .../pdb_next.py(19)f()->None
-> return

(Pdb)
```

16.7.3. Точки останова

По мере роста размеров программ процесс отладки резко замедляется и становится чересчур громоздким даже в случае использования команд `next` и `until`. Вместо того чтобы управлять пошаговым выполнением программы вручную, лучше позволить ей выполняться обычным образом до тех пор, пока она не достигнет точки, в которой отладчик прервет ее выполнение. Для запуска отладчика можно использовать функцию `set_trace()`, но такой подход сработает лишь в том случае, если в программе имеется единственная точка, в которой выполнение программы должно приостановиться. Гораздо удобнее выполнять программу посредством отладчика с заблаговременной передачей ему информации о *точках останова*, в которых выполнение программы должно приостанавливаться.

Листинг 16.70. `pdb_break.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3 #
 4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
 5 #
 6
 7
 8 def calc(i, n):
 9     j = i * n
10     print('j =', j)
11     if j > 0:
```

```
12          print('Positive!')
13      return j
14
15
16  def f(n):
17      for i in range(n):
18          print('i =', i)
19          j = calc(i, n)  # noqa
20      return
21
22  if __name__ == '__main__':
23      f(5)
```

Существует несколько способов задания точек останова для команды `break` (сокращенно — `b`), в том числе с указанием номера строки, файла или функции, в которых обработка должна приостанавливаться. Чтобы задать точку останова на определенной строке текущего файла, следует использовать команду `break` *номер_строки*.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb)
```

Команда `continue` (сокращенная версия — `c`) информирует отладчик о необходимости возобновить выполнение программы до следующей точки останова. В данном примере вызов команды `c` приводит к выполнению первой итерации цикла `for` в функции `f()` и приостановке выполнения программы в теле функции `calc()` при прохождении второй итерации.

Точки останова можно также устанавливать на первой строке функции, указав имя функции вместо номера строки. В следующем примере показано, что произойдет, если добавить точку останова для функции `calc()`.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8
```

```
(Pdb) continue
i = 0
> .../pdb_break.py(9)calc()
-> j = i * n

(Pdb) where
  .../pdb_break.py(23)<module>()
-> f(5)
  .../pdb_break.py(19)f()
-> j = calc(i, n)
> .../pdb_break.py(9)calc()
-> j = i * n

(Pdb)
```

Чтобы задать точку останова в другом файле, следует предварить номер строки или имя функции префиксом в виде имени файла.

Листинг 16.71. pdb_break_remote.py

```
1 #!/usr/bin/env python3
2 # encoding: utf-8
3
4 from pdb_break import f
5
6 f(5)
```

Ниже для программы pdb_break_remote.py в качестве точки останова задана строка 12 в файле pdb_break.py.

```
$ python3 -m pdb pdb_break_remote.py

> .../pdb_break_remote.py(4)<module>()
-> from pdb_break import f
(Pdb) break pdb_break.py:12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb)
```

В качестве имени файла можно указать либо полный путь к файлу, либо относительный путь, доступный через список sys.path.

Для получения текущего списка заданных точек останова следует выполнить команду break без аргументов. Вывод включает имя файла и номер строки для каждой точки останова, а также информацию о том, сколько раз она встретилась.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):

(Pdb) break 12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb) continue
Positive!
i = 2
j = 10
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:12
        breakpoint already hit 2 times

(Pdb)
```

16.7.3.1. Управление точками останова

При добавлении каждой новой точки останова ей присваивается числовой идентификатор. Эти идентификаторы используются для активизации, деактивизации и удаления указанных точек останова в интерактивном режиме. Отключение точки останова с помощью команды `disable` информирует отладчик о том, что выполнение программы при достижении данной строки не должно приостанавливаться. В этом случае точка останова запоминается, но игнорируется.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8

(Pdb) break 12
Breakpoint 2 at .../pdb_break.py:12
```

```
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:8
2   breakpoint   keep yes   at .../pdb_break.py:12

(Pdb) disable 1

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep no    at .../pdb_break.py:8
2   breakpoint   keep yes   at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb)
```

В приведенном ниже сеансе отладки в программе устанавливаются две точки останова, одна из которых затем деактивизируется. Программа выполняется, пока не встретится оставшаяся точка останова, после чего другая точка останова активизируется, прежде чем программа возобновит выполнение.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8

(Pdb) break 18
Breakpoint 2 at .../pdb_break.py:18

(Pdb) disable 1

(Pdb) continue
> .../pdb_break.py(18)f()
-> print('i =', i)

(Pdb) list
 13         return j
 14
 15
 16     def f(n):
 17         for i in range(n):
 18 B->          print('i =', i)
 19             j = calc(i, n)  # noqa
 20         return
 21
```

```
 22     if __name__ == '__main__':
 23         f(5)

(Pdb) continue
i = 0
j = 0
> .../pdb_break.py(18)f()
-> print('i =', i)

(Pdb) list
 13         return j
 14
 15
 16     def f(n):
 17         for i in range(n):
 18 B->          print('i =', i)
 19             j = calc(i, n)  # noqa
 20         return
 21
 22     if __name__ == '__main__':
 23         f(5)

(Pdb) p i
1

(Pdb) enable 1
Enabled breakpoint 1 at .../pdb_break.py:8

(Pdb) continue
i = 1
> .../pdb_break.py(9)calc()
-> j = i * n

(Pdb) list
  4     # Copyright (c) 2010 Doug Hellmann.  All rights reserved.
  5     #
  6
  7
  8 B   def calc(i, n):
  9  ->     j = i * n
 10         print('j =', j)
 11         if j > 0:
 12             print('Positive!')
 13         return j
 14

(Pdb)
```

Префикс `B` в строках вывода, полученных с помощью команды `list`, указывает, где именно в программе заданы точки останова (строки 8 и 18).

Для полного сброса точки останова следует использовать команду `clear`.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8

(Pdb) break 12
Breakpoint 2 at .../pdb_break.py:12

(Pdb) break 18
Breakpoint 3 at .../pdb_break.py:18

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:8
2   breakpoint   keep yes   at .../pdb_break.py:12
3   breakpoint   keep yes   at .../pdb_break.py:18

(Pdb) clear 2
Deleted breakpoint 2 at .../pdb_break.py:12

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:8
3   breakpoint   keep yes   at .../pdb_break.py:18

(Pdb)
```

Оставшиеся точки останова сохраняют свои идентификаторы и не перенумеровываются.

16.7.3.2. Временные точки останова

Временная точка останова, созданная с помощью команды `tbreak`, автоматически сбрасывается после того, как поток выполнения программы впервые достигает ее. Использование временной точки останова упрощает быстрое достижение определенной точки в программе, как и в случае обычной точки останова. Однако, поскольку временная точка останова немедленно сбрасывается, она не будет создавать задержку, если данный участок программы станет выполняться повторно.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) tbreak 12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
```

```
i = 1
j = 5
Deleted breakpoint 1 at .../pdb_break.py:12
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb) break

(Pdb) continue
Positive!
i = 2
j = 10
Positive!
i = 3
j = 15
Positive!
i = 4
j = 20
Positive!
The program finished and will be restarted
> .../pdb_break.py(8)<module>()
-> def calc(i, n):

(Pdb)
```

После того как программа впервые достигнет строки 12, точка останова будет сброшена. Выполнение программы не станет приостанавливаться в этой точке до полного завершения программы.

16.7.3.3. Условные точки останова

К точкам останова можно применять правила, в соответствии с которыми приостановка выполнения программы будет происходить лишь при выполнении определенных условий. По сравнению с активизацией и деактивизацией точек останова вручную этот подход обеспечивает более гибкие возможности управления процессом отладки. Условные точки останова можно устанавливать двумя способами. Первый из них заключается в том, чтобы указать условие, при соблюдении которого выполнение команды `break` приведет к созданию точки останова.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 10, j>0
Breakpoint 1 at .../pdb_break.py:10

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:10
        stop only if j>0
(Pdb) continue
i = 0
j = 0
```

```
i = 1
> .../pdb_break.py(10)calc()
-> print('j =', j)

(Pdb)
```

Значения, которые используются в выражении аргумента, определяющего условие, должны быть видимыми в том фрейме стека, в котором определяется точка останова. Выполнение программы будет приостановлено в точке останова только в том случае, если результат вычисления выражения является истинным.

Альтернативный вариант заключается в применении условия к существующей точке останова с помощью команды `condition`. Аргументами этой команды служат идентификатор точки останова и выражение.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 10
Breakpoint 1 at .../pdb_break.py:10

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:10
(Pdb) condition 1 j>0
New condition set for breakpoint 1.

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:10
        stop only if j>0

(Pdb)
```

16.7.3.4. Игнорирование точек останова

Отладку программ, содержащих циклы или использующих большое количество рекурсивных вызовов одной и той же функции, часто можно упростить за счет заблаговременного отказа от наблюдения за каждым вызовом или каждой точкой останова и сквозного выполнения соответствующих участков кода. Это можно сделать с помощью команды `ignore`, сообщающей отладчику о том, что точка останова должна быть пройдена без задержки. Каждый раз, когда программа достигает точки останова, счетчик проходов команды `ignore` уменьшается на единицу. Точка останова активизируется заново, когда значение счетчика становится равным нулю.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 19
```

```
Breakpoint 1 at .../pdb_break.py:19

(Pdb) continue
i = 0
> .../pdb_break.py(19)f()
-> j = calc(i, n)  # noqa

(Pdb) next
j = 0
> .../pdb_break.py(17)f()
-> for i in range(n):

(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:19
        ignore next 2 hits
        breakpoint already hit 1 time

(Pdb) continue
i = 1
j = 5
Positive!
i = 2
j = 10
Positive!
i = 3
> .../pdb_break.py(19)f()
-> j = calc(i, n)  # noqa

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:19
        breakpoint already hit 4 times
(Pdb)
```

Явный сброс значения счетчика проходов команды `ignore` приводит к немедленной реактивизации точки останова.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 19
Breakpoint 1 at .../pdb_break.py:19

(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.

(Pdb) break
Num Type         Disp Enb   Where
```

```
1   breakpoint   keep yes   at .../pdb_break.py:19
        ignore next 2 hits

(Pdb) ignore 1 0
Will stop next time breakpoint 1 is reached.

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:19
(Pdb)
```

16.7.3.5. Запуск выполнения операций по достижении точки останова

Кроме интерактивного режима модуль pdb поддерживает выполнение простых сценариев. Команда commands позволяет выполнить последовательность команд интерпретатора, в том числе инструкции Python, при достижении определенной точки останова, номер которой задается в качестве аргумента. После выполнения команды commands отображаемое отладчиком приглашение ко вводу (командная подсказка) заменяется приглашением (com). Нужные команды сценария вводятся по одной за раз и завершаются командой end, которая сохраняет сценарий и возвращается к основному приглашению отладчика.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 10
Breakpoint 1 at .../pdb_break.py:10

(Pdb) commands 1
(com) print('debug i =', i)
(com) print('debug j =', j)
(com) print('debug n =', n)
(com) end

(Pdb) continue
i = 0
debug i = 0
debug j = 0
debug n = 5
> .../pdb_break.py(10)calc()
-> print('j =', j)

(Pdb) continue
j = 0
i = 1
debug i = 1
debug j = 5
debug n = 5
> .../pdb_break.py(10)calc()
-> print('j =', j)

(Pdb)
```

Эта возможность особенно полезна при отладке кода, содержащего большое количество структур данных или переменных. Она обеспечивает автоматический вывод всех значений, избавляя от необходимости выводить их вручную каждый раз, когда встречается указанная точка останова.

16.7.3.6. Отслеживание изменения данных

Также существует возможность следить за изменениями значений переменных в процессе выполнения программы без явного использования команд `print`. Для этого служит команда `display`.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 18
Breakpoint 1 at .../pdb_break.py:18

(Pdb) continue
> .../pdb_break.py(18)f()
-> print('i =', i)

(Pdb) display j
display j: ** raised NameError: name 'j' is not defined **

(Pdb) next
i = 0
> .../pdb_break.py(19)f()
-> j = calc(i, n)  # noqa

(Pdb) next
j = 0
> .../pdb_break.py(17)f()
-> for i in range(n):
display j: 0  [old: ** raised NameError: name 'j' is not defined
**]

(Pdb)
```

Каждый раз, когда выполнение приостанавливается во фрейме стека, вычисляется выражение. Если его значение изменилось, то оно выводится вместе с прежним значением. Команда `display`, для которой не заданы аргументы, выводит список отображаемых значений, активных для текущего фрейма.

```
(Pdb) display
Currently displaying:
j: 0

(Pdb) up
> .../pdb_break.py(23)<module>()
-> f(5)

(Pdb) display
```

```
Currently displaying:

(Pdb)
```

Для сброса выражения команды display нужно использовать команду undisplay.

```
(Pdb) display
Currently displaying:
j: 0

(Pdb) undisplay j

(Pdb) display
Currently displaying:

(Pdb)
```

16.7.4. Изменение потока управления

Команда jump изменяет поток управления программы без внесения изменений в код. Она обеспечивает пропуск выполнения некоторых участков кода или возврат к предыдущему коду для его повторного выполнения. Приведенный ниже сценарий генерирует числовой список.

Листинг 16.72. pdb_jump.py

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3 #
 4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
 5 #
 6
 7
 8 def f(n):
 9     result = []
10     j = 0
11     for i in range(n):
12         j = i * n + j
13         j += n
14         result.append(j)
15     return result
16
17 if __name__ == '__main__':
18     print(f(5))
```

Вывод, полученный при выполнении этого сценария без какого-либо вмешательства, представляет собой возрастающую последовательность чисел, которые делятся на 5.

```
$ python3 pdb_jump.py

[5, 15, 30, 50, 75]
```

16.7.4.1. Переходы вперед

Переходы вперед перемещают точку выполнения программы в прямом направлении относительно текущего местоположения без выполнения промежуточных инструкций, располагающихся между начальной и конечной точками. Пропуск строки 13 в этом примере приводит к тому, что значение переменной j не инкрементируется, и поэтому все последующие значения, которые зависят от него, оказываются немного меньшими.

```
$ python3 -m pdb pdb_jump.py

> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb) break 13
Breakpoint 1 at .../pdb_jump.py:13

(Pdb) continue
> .../pdb_jump.py(13)f()
-> j += n

(Pdb) p j
0

(Pdb) step
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
5

(Pdb) continue
> .../pdb_jump.py(13)f()
-> j += n

(Pdb) jump 14
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
10

(Pdb) disable 1
Disabled breakpoint 1 at .../pdb_jump.py:13

(Pdb) continue
[5, 10, 25, 45, 70]

The program finished and will be restarted
> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb)
```

16.7.4.2. Переходы назад

Команду jump можно также использовать для перемещения точки выполнения в обратном направлении, что позволяет повторно выполнить ранее выполненный код. В следующем примере значение j инкрементируется один лишний раз, поэтому числа в результирующей последовательности оказываются большими, чем они были бы в противном случае.

```
$ python3 -m pdb pdb_jump.py

> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb) break 14
Breakpoint 1 at .../pdb_jump.py:14

(Pdb) continue
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
5

(Pdb) jump 13
> .../pdb_jump.py(13)f()
-> j += n

(Pdb) continue
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
10

(Pdb) disable 1
Disabled breakpoint 1 at .../pdb_jump.py:14

(Pdb) continue
[10, 20, 35, 55, 80]
The program finished and will be restarted
> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb)
```

16.7.4.3. Запрещенные переходы

Переходы в тело некоторых инструкций потока выполнения или из них опасны или вызывают неопределенность. Отладчик не допускает такое поведение.

Листинг 16.73. pdb_no_jump.py

```
1 #!/usr/bin/env python3
2 # encoding: utf-8
3 #
```

```
 4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
 5 #
 6
 7
 8 def f(n):
 9     if n < 0:
10         raise ValueError('Invalid n: {}'.format(n))
11     result = []
12     j = 0
13     for i in range(n):
14         j = i * n + j
15         j += n
16         result.append(j)
17     return result
18
19
20 if __name__ == '__main__':
21     try:
22         print(f(5))
23     finally:
24         print('Always printed')
25
26     try:
27         print(f(-5))
28     except:
29         print('There was an error')
30     else:
31         print('There was no error')
32
33     print('Last statement')
```

Несмотря на то что команда `jump` может быть использована для перехода внутрь функции, нормальная работа кода маловероятна, поскольку аргумент функции не определен.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 22
Breakpoint 1 at .../pdb_no_jump.py:22

(Pdb) jump 9
> .../pdb_no_jump.py(9)<module>()
-> if n < 0:

(Pdb) p n
*** NameError: name 'n' is not defined

(Pdb) args

(Pdb)
```

С помощью команды jump нельзя входить в середину блока, такого как цикл for или инструкция try:except.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 22
Breakpoint 1 at .../pdb_no_jump.py:22

(Pdb) continue
> .../pdb_no_jump.py(22)<module>()
-> print(f(5))

(Pdb) jump 27
*** Jump failed: can't jump into the middle of a block

(Pdb)
```

Код в блоке finally должен всегда выполняться, поэтому выход за пределы блока с помощью команды jump невозможен.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 24
Breakpoint 1 at .../pdb_no_jump.py:24

(Pdb) continue
[5, 15, 30, 50, 75]
> .../pdb_no_jump.py(24)<module>()
-> print 'Always printed'

(Pdb) jump 26
*** Jump failed: can't jump into or out of a 'finally' block

(Pdb)
```

Основное ограничение заключается в том, что переходы ограничены нижним фреймом стека. Порядок выполнения программы нельзя будет изменить, если контекст отладки был изменен с помощью команды up.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 12
Breakpoint 1 at .../pdb_no_jump.py:12

(Pdb) continue
> .../pdb_no_jump.py(12)f()
-> j = 0
```

```
(Pdb) where
  .../lib/python3.5/bdb.py(
431)run()
-> exec cmd in globals, locals
  <string>(1)<module>()
  .../pdb_no_jump.py(22)<module>()
-> print(f(5))
> .../pdb_no_jump.py(12)f()
-> j = 0

(Pdb) up
> .../pdb_no_jump.py(22)<module>()
-> print(f(5))

(Pdb) jump 25
*** You can only jump within the bottom frame

(Pdb)
```

16.7.4.4. Перезапуск программы

Когда отладчик достигает конца программы, она автоматически перезапускается. Кроме того, программу можно перезапустить явным образом, не покидая отладчик и не теряя текущих точек останова и других настроек.

Листинг 16.74. pdb_run.py

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3 #
 4 # Copyright (c) 2010 Doug Hellmann. All rights reserved.
 5 #
 6
 7 import sys
 8
 9
10 def f():
11     print('Command-line args:', sys.argv)
12     return
13
14 if __name__ == '__main__':
15     f()
```

Будучи выполненной до ее завершения с помощью отладчика, как показано ниже, программа выведет только имя файла сценария, поскольку никакие другие аргументы в командной строке не были переданы.

```
$ python3 -m pdb pdb_run.py

> .../pdb_run.py(7)<module>()
-> import sys
(Pdb) continue

Command line args: ['pdb_run.py']
```

```
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

Для перезапуска программы можно воспользоваться командой `run`. Полученные командой `run` аргументы анализируются с помощью модуля `shlex` (раздел 14.6) и передаются программе, как если бы они являлись аргументами командной строки, поэтому программу можно перезапускать с другими параметрами.

```
(Pdb) run a b c "this is a long value"
Restarting pdb_run.py with arguments:
        a b c this is a long value
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb) continue
Command line args: ['pdb_run.py', 'a', 'b', 'c',
'this is a long value']
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

Команду `run` можно использовать для перезапуска программы в любом другом месте в процессе отладки.

```
$ python3 -m pdb pdb_run.py

> .../pdb_run.py(7)<module>()
-> import sys
(Pdb) break 11
Breakpoint 1 at .../pdb_run.py:11

(Pdb) continue
> .../pdb_run.py(11)f()
-> print('Command line args:', sys.argv)

(Pdb) run one two three
Restarting pdb_run.py with arguments:
        one two three
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

16.7.5. Настройка отладчика с помощью псевдонимов

Использование псевдонимов позволяет избежать утомительного повторного ввода сложных команд, сокращая их за счет использования псевдонимов. Раскрытие псевдонимов применяется к первому слову каждой команды. Тело

псевдонима может включать любую команду, ввод которой в ответ на приглашение отладчика является допустимым, в том числе другие команды отладчика и выражения Python. В определениях псевдонимов допускается использовать рекурсию, поэтому одни псевдонимы могут вызывать другие.

```
$ python3 -m pdb pdb_function_arguments.py

> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) break 11
Breakpoint 1 at .../pdb_function_arguments.py:11

(Pdb) continue
> .../pdb_function_arguments.py(11)recursive_function()
-> if n > 0:

(Pdb) pp locals().keys()
dict_keys(['output', 'n'])

(Pdb) alias pl pp locals().keys()

(Pdb) pl
dict_keys(['output', 'n'])
```

Команда `alias` без аргументов отображает список определенных псевдонимов. Если задан один аргумент, то он воспринимается как псевдоним, и выводится его определение.

```
(Pdb) alias
pl = pp locals().keys()

(Pdb) alias pl
pl = pp locals().keys()

(Pdb)
```

К аргументам псевдонимов можно обращаться с помощью ссылок вида `%n`, где *n* — номер позиции аргумента, отсчитываемый от 1. Для обращения сразу ко всем аргументам используется синтаксис `%*`.

```
$ python3 -m pdb pdb_function_arguments.py

> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) alias ph !help(%1)

(Pdb) ph locals
Help on built-in function locals in module builtins:

locals()
    Return a dictionary containing the current scope's local
    variables.

    NOTE: Whether or not updates to this dictionary will affect
```

```
    name lookups in the local scope and vice-versa is
    *implementation dependent* and not covered by any backwards
    compatibility guarantees.
```

Определение псевдонима можно удалить с помощью команды unalias.

```
(Pdb) unalias ph

(Pdb) ph locals
*** SyntaxError: invalid syntax (<stdin>, line 1)

(Pdb)
```

16.7.6. Сохранение конфигурационных параметров

Процесс отладки программы состоит из ряда повторяющихся этапов: выполнение кода, анализ результатов, исправление кода или данных и повторное выполнение кода. Модуль pdb позволяет уменьшить количество таких повторений и тем самым сосредоточить основное внимание на коде, а не на управлении процессом отладки. Для этого в модуле pdb предусмотрены средства, обеспечивающие чтение конфигурационной информации, сохраненной в текстовых файлах, которая интерпретируется при запуске отладчика.

Сначала читается файл ~/.pdbrc. Он устанавливает глобальные персональные настройки для всех сеансов отладки. Затем из текущего каталога читается файл ./.pdbrc, с помощью которого устанавливаются локальные параметры для конкретного проекта.

```
$ cat ~/.pdbrc

# Отобразить справку Python
alias ph !help(%1)
# Переопределенный псевдоним
alias redefined p 'home definition'

$ cat .pdbrc

# Точки останова
break 11
# Переопределенный псевдоним
alias redefined p 'local definition'

$ python3 -m pdb pdb_function_arguments.py

Breakpoint 1 at .../pdb_function_arguments.py:11
> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) alias
ph = !help(%1)
redefined = p 'local definition'

(Pdb) break
```

```
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_function_arguments.py:11

(Pdb)
```

Любые команды конфигурирования, которые можно вводить в ответ на приглашение, отображаемое отладчиком, могут быть сохранены в одном из таких файлов автозапуска. Точно так же могут быть сохранены некоторые команды, управляющие процессом выполнения (например, `continue` или `next`).

```
$ cat .pdbrc
break 11
continue
list

$ python3 -m pdb pdb_function_arguments.py
Breakpoint 1 at .../pdb_function_arguments.py:11
  6
  7  import pdb
  8
  9
  10  def recursive_function(n=5, output='to be printed'):
  11 B->    if n > 0:
  12            recursive_function(n - 1)
  13        else:
  14            pdb.set_trace()
  15            print(output)
  16        return
> .../pdb_function_arguments.py(11)recursive_function()
-> if n > 0:
(Pdb)
```

Особенно полезно сохранять команды `run`. Это позволяет задавать аргументы командной строки в файле `./.pdbrc`, тем самым обеспечивая их последовательное использование на протяжении нескольких запусков.

```
$ cat .pdbrc
run a b c "long argument"

$ python3 -m pdb pdb_run.py
Restarting pdb_run.py with arguments:
      a b c "long argument"
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb) continue
Command-line args: ['pdb_run.py', 'a', 'b', 'c',
'long argument']
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю pdb[18].
- readline (раздел 14.3). Библиотека средств редактирования интерактивной подсказки.
- cmd (раздел 14.5). Создание интерактивных программ.
- shlex (раздел 14.6). Синтаксический анализ командной строки.
- *Python issue 26053*[19]. Если вывод команды run не соответствует значениям, приведенным в примере, прочтите это сообщение, в котором подробно проанализирована причина различий вывода pdb в версиях Python 2.7 и 3.5.

16.8. profile и pstats: анализ производительности

Модуль profile предоставляет программные интерфейсы, предназначенные для сбора и анализа статистической информации о потреблении процессорного времени и других ресурсов кодом на языке Python.

Примечание

Приведенные в этом разделе отчеты были переформатированы для того, чтобы строки умещались по ширине страницы. Строки, заканчивающиеся символом обратной косой черты (\), продолжаются на следующей строке.

16.8.1. Запуск профилировщика

Проще всего начать работу с модулем profile, вызвав функцию run(). Эта функция получает строку инструкции в качестве аргумента и создает отчет, в котором отображается количество времени, затраченного на выполнение различных строк кода при обработке данной инструкции.

Листинг 16.75. profile_fibonacci_raw.py

```
import profile


def fib(n):
    # Код взят на сайте literateprograms.org
    # http://bit.ly/hlOQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n - 1))
```

[18] https://docs.python.org/3.5/library/pdb.html

[19] http://bugs.python.org/issue26053

```
    seq.append(fib(n))
    return seq


profile.run('print(fib_seq(20)); print()')
```

Вышеприведенная рекурсивная версия программы для вычисления последовательности чисел Фибоначчи особенно хорошо подходит для демонстрации возможностей модуля `profile`, поскольку ее производительность может быть значительно улучшена. В отчете стандартного формата сначала выводятся итоговые данные, а затем подробные сведения для каждой выполнявшейся функции.

```
$ python3 profile_fibonacci_raw.py

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 98\
7, 1597, 2584, 4181, 6765]

         57359 function calls (69 primitive calls) in 0.219 seco\
nds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu\
nction)
       21    0.000    0.000    0.000    0.000 :0(append)
        1    0.000    0.000    0.127    0.127 :0(exec)
       20    0.000    0.000    0.000    0.000 :0(extend)
        2    0.000    0.000    0.000    0.000 :0(print)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.001    0.001    0.001    0.001 <string>:1(<module\
>)
        1    0.000    0.000    0.127    0.127 profile:0(print(fi\
b_seq(20)); print())
        0    0.000             0.000          profile:0(profiler\
)
 57291/21    0.126    0.000    0.126    0.006 profile_fibonacci_\
raw.py:11(fib)
     21/1    0.000    0.000    0.127    0.127 profile_fibonacci_\
raw.py:22(fib_seq)
```

В исходной версии программы было выполнено 57359 отдельных вызовов функций за 0,127 секунды. Тот факт, что имеется всего лишь 69 *примитивных* вызовов, говорит о том, что подавляющее большинство вызовов функций из их общего количества 57359 были рекурсивными. Подробные сведения относительно того, на что именно было израсходовано время, разбиты по функциям и отображают количество вызовов (`ncalls`), общее время, затраченное на функцию (`tottime`), время, затраченное на один вызов (`tottime/ncalls`), кумулятивное время, затраченное на функцию, и отношение кумулятивного времени к количеству примитивных вызовов.

Как и следовало ожидать, большую часть времени заняли повторные вызовы функции `fib()`. Добавление декоратора кеша уменьшает количество рекурсивных вызовов, что разительно сказывается на производительности функции.

Листинг 16.76. profile_fibonacci_memoized.py

```
import functools
import profile


@functools.lru_cache(maxsize=None)
def fib(n):
    # Код взят на сайте literateprograms.org
    # http://bit.ly/hlOQ5m
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


def fib_seq(n):
    seq = []
    if n > 0:
        seq.extend(fib_seq(n - 1))
    seq.append(fib(n))
    return seq


if __name__ == '__main__':
    profile.run('print(fib_seq(20)); print()')
```

Благодаря тому, что числа Фибоначчи запоминаются на каждом уровне, удается по большей части избавиться от рекурсии и снизить количество вызовов до 89, на что ушло всего лишь 0,001 секунды. Счетчик `ncalls` для функции `fib()` указывает на то, что ее рекурсивные вызовы вообще отсутствовали.

```
$ python3 profile_fibonacci_memoized.py

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 98\
7, 1597, 2584, 4
181, 6765]

         89 function calls (69 primitive calls) in 0.016 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       21    0.000    0.000    0.000    0.000 :0(append)
        1    0.000    0.000    0.000    0.000 :0(exec)
       20    0.000    0.000    0.000    0.000 :0(extend)
        2    0.000    0.000    0.000    0.000 :0(print)
        1    0.016    0.000    0.016    0.016 :0(setprofile)
        1    0.001    0.001    0.001    0.001 <string>:1(<module\
>)
        1    0.000    0.000    0.001    0.001 profile:0(print(fi\
```

```
b_seq(20)); print())
        0    0.000              0.000          profile:0(profiler\
)
       21    0.000    0.000     0.000    0.000 profile_fibonacci_\
memoized.py:12(fib)
     21/1    0.000    0.000     0.000    0.000 profile_fibonacci_\
memoized.py:24(fib_seq)
```

16.8.2. Выполнение в контексте

Иногда, вместо того чтобы конструировать сложное выражение для функции `run()`, легче создать простое выражение и передать ему параметры через контекст, используя функцию `runctx()`.

Листинг 16.77. `profile_runctx.py`

```
import profile
from profile_fibonacci_memoized import fib, fib_seq

if __name__ == '__main__':
    profile.runctx(
        'print(fib_seq(n)); print()',
        globals(),
        {'n': 20},
    )
```

В этом примере значение `n` передается через контекст локальной переменной, а не встраивается непосредственно в инструкцию, передаваемую функции `runctx()`.

```
$ python3 profile_runctx.py

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
987, 1597, 2584, 4181, 6765]

         148 function calls (90 primitive calls) in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(\
function)
       21    0.000    0.000    0.000    0.000 :0(append)
        1    0.000    0.000    0.001    0.001 :0(exec)
       20    0.000    0.000    0.000    0.000 :0(extend)
        2    0.000    0.000    0.000    0.000 :0(print)
        1    0.001    0.001    0.001    0.001 :0(setprofile)
        1    0.000    0.000    0.001    0.001 <string>:1(<module\
>)
        1    0.000    0.000    0.002    0.002 profile:0(print(fi\
b_seq(n)); print())
        0    0.000              0.000          profile:0(profiler\
)
    59/21    0.000    0.000    0.000    0.000 profile_fibonacci_\
```

```
memoized.py:19(__call__)
       21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:27(fib)
     21/1    0.000    0.000    0.001    0.001 profile_fibonacci_\
memoized.py:39(fib_seq)
```

16.8.3. pstats: работа со статистиками

Стандартный отчет, создаваемый функциями модуля `profile`, не особенно гибок. Однако можно создать пользовательский отчет, сохранив исходные данные профилирования, полученные с помощью функций `run()` и `runctx()`, и выполнив их отдельную обработку с помощью класса `pstats.Stats`.

Ниже приведен пример выполнения нескольких итераций одного и того же теста и объединения их результатов.

Листинг 16.78. profile_stats.py

```
import cProfile as profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Создать 5 наборов статистических объектов
for i in range(5):
    filename = 'profile_stats_{}.stats'.format(i)
    profile.run('print({}, fib_seq(20))'.format(i), filename)

# Прочитать все 5 файлов статистик в один объект
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_{}.stats'.format(i))

# Удалить информацию о путях из имен файлов для отчета
stats.strip_dirs()

# Сортировать статистики по суммарному времени, затраченному на
# выполнение функции
stats.sort_stats('cumulative')

stats.print_stats()
```

Данные в выходном отчете сортируются в порядке убывания кумулятивного времени выполнения функции. Имена каталогов удаляются из имен файлов с целью экономии места при их выводе на экран.

```
$ python3 profile_stats.py

0 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
1 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
2 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
```

```
3 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
4 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \
987, 1597, 2584, 4181, 6765]
Sat Dec 31 07:46:22 2016    profile_stats_0.stats
Thu Nov 31 07:46:22 2016    profile_stats_1.stats
Thu Nov 31 07:46:22 2016    profile_stats_2.stats
Thu Nov 31 07:46:22 2016    profile_stats_3.stats
Thu Nov 31 07:46:22 2016    profile_stats_4.stats

         351 function calls (251 primitive calls) in 0.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu\
nction)
        5    0.000    0.000    0.000    0.000 {built-in method b\
uiltins.exec}
        5    0.000    0.000    0.047    0.009 <string>:1(<module>)
    105/5    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:24(fib_seq)
        5    0.000    0.009    0.000    0.000 {built-in method b\
uiltins.print}
      100    0.000    0.000    0.000    0.000 {method 'extend' o\
f 'list' objects}
       21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:12(fib)
      105    0.000    0.000    0.000    0.000 {method 'append' o\
f 'list' objects}
        5    0.000    0.000    0.000    0.000 {method 'disable' \
of '_lsprof.Profiler' objects}
```

16.8.4. Ограничение содержимого отчета

Вывод можно ограничивать данными, относящимися к определенным функциям. В приведенной ниже версии программы ограничение вывода информацией, касающейся только функций `fib()` и `fib_seq()`, достигается за счет использования регулярного выражения для поиска нужных значений *имя_файла*:*номер_строки*(*функция*).

Листинг 16.79. `profile_stats_restricted.py`

```
import profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Прочитать все 5 файлов статистик в один объект
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_{}.stats'.format(i))
stats.strip_dirs()
stats.sort_stats('cumulative')
```

```
# Ограничить вывод строками, содержащими подстроку "(fib"
stats.print_stats('\(fib')
```

Регулярное выражение включает литеральный символ открывающей круглой скобки (() для сопоставления с той частью местоположения источника данных, которая соответствует имени файла.

```
$ python3 profile_stats_restricted.py

Sat Dec 31 07:46:22 2016 profile_stats_0.stats
Sat Dec 31 07:46:22 2016 profile_stats_1.stats
Sat Dec 31 07:46:22 2016 profile_stats_2.stats
Sat Dec 31 07:46:22 2016 profile_stats_3.stats
Sat Dec 31 07:46:22 2016 profile_stats_4.stats

         351 function calls (251 primitive calls) in 0.049 seconds

   Ordered by: cumulative time
   List reduced from 8 to 2 due to restriction <'\\(fib'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu\
nction)
    105/5    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:24(fib_seq)
       21    0.000    0.000    0.000    0.000 profile_fibonacci_\
memoized.py:12(fib)
```

16.8.5. Графы вызова функций

Объекты статистики имеют методы, обеспечивающие вывод информации об объектах, вызывающих указанную функцию или вызываемых ею.

Листинг 16.80. profile_stats_callers.py

```
import cProfile as profile
import pstats
from profile_fibonacci_memoized import fib, fib_seq

# Прочитать все 5 файлов статистик в один объект
stats = pstats.Stats('profile_stats_0.stats')
for i in range(1, 5):
    stats.add('profile_stats_{}.stats'.format(i))
stats.strip_dirs()
stats.sort_stats('cumulative')

print('INCOMING CALLERS:')
stats.print_callers('\(fib')

print('OUTGOING CALLEES:')
stats.print_callees('\(fib')
```

Аргументы функций `print_callers()` и `print_callees()` работают аналогично аргументам ограничения функции `print_stats()`. В выводе отображается

информация о вызывающем и вызываемом объектах, количестве вызовов и кумулятивном времени.

```
$ python3 profile_stats_callers.py

INCOMING CALLERS:
   Ordered by: cumulative time
   List reduced from 8 to 2 due to restriction <'\\(fib'>

Function                                    was called by...
                                                ncalls  tottime  \
cumtime
profile_fibonacci_memoized.py:24(fib_seq)  <-       5    0.000  \
  0.000  <string>:1(<module>)
                                                100/5    0.000  \
  0.000  profile_fibonacci_memoized.py:24(fib_seq)
profile_fibonacci_memoized.py:12(fib)      <-      21    0.000  \
  0.000  profile_fibonacci_memoized.py:24(fib_seq)


OUTGOING CALLEES:
   Ordered by: cumulative time
   List reduced from 8 to 2 due to restriction <'\\(fib'>

Function                                    called...
                                                ncalls  tottime  \
cumtime
profile_fibonacci_memoized.py:24(fib_seq)  ->      21    0.000  \
  0.000  profile_fibonacci_memoized.py:12(fib)
                                                100/5    0.000  \
  0.000  profile_fibonacci_memoized.py:24(fib_seq)
                                                  105    0.000  \
  0.000  {method 'append' of 'list' objects}
                                                  100    0.000  \
  0.000  {method 'extend' of 'list' objects}
profile_fibonacci_memoized.py:12(fib)      ->
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `profile`[20].
- `functools.lru_cache()` (раздел 3.1.3). Декоратор кеша, используемый в приведенном примере для повышения производительности.
- Класс `Stats`[21]. Раздел документации стандартной библиотеки, посвященный классу `pstats.Stats`.
- `Gprof2Dot`[22]. Средство визуализации данных профилирования.
- `Smiley`[23]. Трассировщик приложений Python.

[20] https://docs.python.org/3.5/library/profile.html

[21] https://docs.python.org/3.5/library/profile.html#the-stats-class

[22] http://code.google.com/p/jrfonseca/wiki/Gprof2Dot

[23] https://github.com/dhellmann/smiley

16.9. timeit: замер времени выполнения небольших фрагментов кода Python

Модуль timeit предоставляет простой интерфейс для определения времени выполнения небольших фрагментов кода Python. Максимально возможная точность определения длительности промежутков времени обеспечивается за счет использования зависящей от платформы функции time(), а снижение влияния накладных расходов, связанных с запуском и завершением выполняемого кода, достигается за счет его многократного выполнения.

16.9.1. Содержимое модуля

Модуль timeit определяет единственный общедоступный класс: Timer. Конструктор класса Timer получает в качестве аргументов инструкцию, подлежащую хронометрированию, и инструкцию "настройки" (используемую для инициализации переменных). Инструкции Python должны быть строками и могут включать встроенные символы новой строки.

Метод timeit() выполняет инструкцию настройки один раз, а затем многократно выполняет основную инструкцию и возвращает количество истекшего времени. Аргумент number метода timeit() позволяет указать, сколько раз необходимо выполнить инструкцию. Значение по умолчанию — 1000000.

16.9.2. Базовый пример

В качестве иллюстрации использования различных аргументов конструктора Timer ниже приведен простой пример, в котором выводятся значения, идентифицирующие каждую выполняющуюся инструкцию.

Листинг 16.81. timeit_example.py

```
import timeit

t = timeit.Timer("print('main statement')", "print('setup')")

print('TIMEIT:')
print(t.timeit(2))

print('REPEAT:')
print(t.repeat(3, 2))
```

В выводе отображаются результаты для многократных вызовов функции print().

```
$ python3 timeit_example.py

TIMEIT:
setup
main statement
main statement
3.7070130929350853e-06
REPEAT:
setup
```

```
main statement
main statement
setup
main statement
main statement
setup
main statement
main statement
[1.4499528333544731e-06, 1.1939555406570435e-06,
1.1870870366692543e-06]
```

Функция `timeit()` выполняет инструкцию настройки один раз, после чего вызывает основную инструкцию `count` раз. Она возвращает значение в виде числа с плавающей точкой, представляющее кумулятивное значение количества времени, затраченного на выполнение основной инструкции.

Также предусмотрена вспомогательная функция `repeat()`, вызывающая функцию `timeit()` несколько раз (в данном случае — три раза). Все результаты возвращаются в виде списка.

16.9.3. Сохранение значений в словаре

В следующем более сложном примере выполняются сравнительные расчеты количества времени, необходимого для заполнения словаря большим количеством значений с использованием различных методов. Прежде всего необходимо сконфигурировать объект `Timer` с помощью нескольких констант. Переменная `setup_statement` инициализирует список кортежей, которые содержат строки и целые числа, используемые основными инструкциями для создания словарей; строки являются ключами словаря, а целые числа — ассоциируемыми с ними значениями.

```
# Константы
range_size = 1000
count = 1000
setup_statement = ';'.join([
    "l = [(str(x), x) for x in range(1000)]",
    "d = {}",
])
```

Функция `show_results()` выводит результаты в удобном формате. Метод `timeit()` возвращает количество времени, которое потребовалось для многократного выполнения инструкции. Вывод функции `show_results()` преобразует это значение в количество времени, приходящегося на одну итерацию, а затем дополнительно уменьшает это значение до среднего количества времени, которое требуется для сохранения одного элемента в словаре.

```
def show_results(result):
    "Выводит число микросекунд на проход и на каждый элемент."
    global count, range_size
    per_pass = 1000000 * (result / count)
    print('{:6.2f} usec/pass'.format(per_pass), end=' ')
    per_item = per_pass / range_size
```

```
    print('{:6.2f} usec/item'.format(per_item))


print("{} items".format(range_size))
print("{} iterations".format(count))
print()
```

В качестве эталона будет выступать первая из тестируемых конфигураций, в которой используется метод `__setitem__()`. Во всех остальных конфигурациях значения, уже находящиеся в словаре, не перезаписываются, поэтому данная простая версия должна быть самой быстрой.

Первым аргументом конструктора `Timer` является многострочная строка, сохранение в которой пробелов гарантирует ее корректный анализ в процессе выполнения. Второй аргумент – это константа, введенная для инициализации списка значений и словаря.

```
# Использование метода __setitem__ без предварительной проверки
# существующих значений
print('__setitem__:', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
            d[s] = i
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

В следующей версии кода для гарантии того, что существующие в словаре значения не будут перезаписываться, используется метод `setdefault()`.

```
# Использование метода setdefault()
print('setdefault :', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
            d.setdefault(s, i)
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

Этот метод добавляет значение только в том случае, если при поиске существующего значения возбуждается исключение `KeyError`.

```
# Использование исключений
print('KeyError :', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
```

```
            try:
                existing = d[s]
            except KeyError:
                d[s] = i
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

Последний метод использует оператор `in` для проверки того, что определенный ключ уже содержится в словаре.

```
# Использование оператора "in"
print('"not in" :', end=' ')
t = timeit.Timer(
    textwrap.dedent(
        """
        for s, i in l:
            if s not in d:
                d[s] = i
        """),
    setup_statement,
)
show_results(t.timeit(number=count))
```

Этот сценарий дает следующий вывод.

```
$ python3 timeit_dictionary.py

1000 items
1000 iterations

__setitem__:  91.79 usec/pass   0.09 usec/item
setdefault : 182.85 usec/pass   0.18 usec/item
KeyError   :  80.87 usec/pass   0.08 usec/item
"not in"   :  66.77 usec/pass   0.07 usec/item
```

Приведенные результаты были получены на компьютере MacMini. Разумеется, результаты будут различными для разных систем, в зависимости от используемого оборудования и программного обеспечения. Экспериментируйте, изменяя значения переменных `range_size` и `count`, поскольку их различные комбинации будут приводить к разным результатам.

16.9.4. Тестирование из командной строки

Кроме программного интерфейса модуль `timeit` предоставляет интерфейс командной строки для тестирования модулей без использования средств замера производительности.

Чтобы выполнить модуль, следует использовать опцию `-m` интерпретатора Python для поиска модуля и его запуска в качестве основной программы.

```
$ python3 -m timeit
```

Например, для получения справки можно воспользоваться следующей командой.

```
$ python3 -m timeit -h

Tool for measuring execution time of small code snippets.

This module avoids a number of common traps for measuring
execution times. See also Tim Peters' introduction to the
Algorithms chapter in the Python Cookbook, published by
O'Reilly.

...
```

Аргумент `statement` работает в командной строке несколько иначе по сравнению с аналогичным аргументом конструктора `Timer`. Вместо того чтобы использовать одну длинную строку, следует передавать каждую строку инструкций в качестве отдельного аргумента командной строки. Для создания отступов строк (например, в теле цикла) необходимо вставлять в строки пробелы, заключая их в кавычки.

```
$ python3 -m timeit -s \

"d={}" \
"for i in range(1000):" \
"  d[str(i)] = i"

1000 loops, best of 3: 306 usec per loop
```

Также существует возможность определить функцию с более сложным кодом, а затем вызвать ее из командной строки.

Листинг 16.82. `timeit_setitem.py`

```
def test_setitem(range_size=1000):
    l = [(str(x), x) for x in range(range_size)]
    d = {}
    for s, i in l:
        d[s] = i
```

Чтобы выполнить тест, следует передать код, импортирующий модули, и запустить тестовую функцию.

```
$ python3 -m timeit \
"import timeit_setitem; timeit_setitem.test_setitem()"

1000 loops, best of 3: 401 usec per loop
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `timeit`[24].

[24] https://docs.python.org/3.5/library/timeit.html

- `profile` (раздел 16.8). Модуль `profile` также может использоваться для анализа производительности.
- Функция `monotonic()` (раздел 4.1.3). Обсуждение содержащейся в модуле `time` функции, обеспечивающей строго монотонное возрастание результатов измерения временных интервалов.

16.10. tabnanny: проверка отступов

Согласованное использование отступов играет важную роль в языке Python, в котором пробелы имеют существенное значение. Модуль `tabnanny` предоставляет средство просмотра кода, которое выводит отчет о любых случаях неоднозначного использования пробелов.

16.10.1. Запуск из командной строки

Простейший способ использования модуля `tabnanny` — вызвать его из командной строки, передав ему имена проверяемых файлов. Если передаются каталоги, в них выполняется рекурсивный поиск *.py*-файлов, подлежащих проверке.

Вызов модуля `tabnanny` по отношению к исходному коду в каталоге `PyMOTW` позволило выявить один старый модуль, в котором вместо пробелов использовались символы табуляции.

```
$ python3 -m tabnanny .
./source/queue/fetch_podcasts.py 65 "    \t\tparsed_url = \
urlparse(enclosure['url'])\n"
```

Строка 65 в файле `fetch_podcasts.py` включала два символа табуляции вместо восьми пробелов. Символы табулятора было не так-то просто заметить в текстовом редакторе, в котором ширина табуляции составляла четыре пробела, так что визуально никакой разницы между двумя символами табуляции и восемью пробелами не было заметно.

```
for enclosure in entry.get('enclosures', []):
            parsed_url = urlparse(enclosure['url'])
            message('queuing {}'.format(
                parsed_url.path.rpartition('/')[-1]))
            enclosure_queue.put(enclosure['url'])
```

Исправление строки 65 и повторное выполнение модуля `tabnanny` позволило обнаружить другую ошибку в строке 66. Еще одна, последняя, проблема проявилась в строке 67.

Если вы хотите только просканировать файлы, не просматривая ошибки, используйте опцию `-q` для подавления вывода всей информации, за исключением имен файлов.

```
$ python3 -m tabnanny -q .
./source/queue/fetch_podcasts.py
```

Для просмотра дополнительной информации о проверяемых файлах следует использовать опцию `-v`.

```
'source/queue/': listing directory
'source/queue/fetch_podcasts.py': *** Line 65: trouble in tab
city! ***
offending line: "    \t\tparsed_url = urlparse(enclosure['url'])
\n"
indent not greater e.g. at tab sizes 1, 2
'source/queue/queue_fifo.py': Clean bill of health.
'source/queue/queue_lifo.py': Clean bill of health.
'source/queue/queue_priority.py': Clean bill of health.
```

Примечание

Выполнение этих примеров с использованием файлов с исходным кодом, находящихся в каталоге `PyMOTW`, не обнаружит описанных выше ошибок, поскольку эти проблемы к настоящему времени устранены.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `tabnanny`[25].
- `tokenize`. Лексический анализатор исходного кода на языке Python.
- `flake8`[26]. Модульное средство проверки исходного кода.
- `pycodestyle`[27]. Средство проверки стиля оформления исходного кода на языке Python.
- `pylint`[28]. Средство статической проверки кода на языке Python.

16.11. `compileall`: файлы скомпилированного байт-кода

Модуль `compileall` находит файлы с исходным кодом на языке Python и компилирует их в представление байт-кода, сохраняя результаты в файлах с расширением *.pyc*.

16.11.1. Компиляция одного каталога

Функция `compile_dir()` используется для рекурсивного просмотра каталога и компиляции всех находящихся в нем файлов с исходным кодом в байт-код.

Листинг 16.83. `compileall_compile_dir.py`

```
import compileall
import glob


def show(title):
    print(title)
```

[25] https://docs.python.org/3.5/library/tabnanny.html

[26] https://pypi.python.org/pypi/flake8

[27] https://pycodestyle.readthedocs.io/en/latest/

[28] https://pypi.python.org/pypi/pylint

```
    for filename in glob.glob('examples/**',
                              recursive=True):
        print('  {}'.format(filename))
    print()


show('Before')

compileall.compile_dir('examples')

show('\nAfter')
```

По умолчанию просматриваются все подкаталоги до глубины вложения 10. Выходные файлы записываются в каталог `__pycache__` с присваиванием им имен на основе версии интерпретатора Python.

```
$ python3 compileall_compile_dir.py

Before
  examples/
  examples/README
  examples/a.py
  examples/subdir
  examples/subdir/b.py

Listing 'examples'...
Compiling 'examples/a.py'...
Listing 'examples/subdir'...
Compiling 'examples/subdir/b.py'...

After
  examples/
  examples/README
  examples/__pycache__
  examples/__pycache__/a.cpython-35.pyc
  examples/a.py
  examples/subdir
  examples/subdir/__pycache__
  examples/subdir/__pycache__/b.cpython-35.pyc
  examples/subdir/b.py
```

16.11.2. Игнорирование файлов

Если необходимо отфильтровать каталоги, следует использовать аргумент `rx` для предоставления регулярного выражения, осуществляющего поиск исключаемых имен.

Листинг 16.84. `compileall_exclude_dirs.py`

```
import compileall
import re

compileall.compile_dir(
```

```
    'examples',
    rx=re.compile(r'/subdir'),
)
```

В следующем примере исключаются файлы из подкаталога subdirectory.

```
$ python3 compileall_exclude_dirs.py

Listing 'examples'...
Compiling 'examples/a.py'...
Listing 'examples/subdir'...
```

Аргумент maxlevels задает глубину рекурсии. Например, чтобы полностью избежать рекурсии, в качестве значения этого аргумента следует передать 0.

Листинг 16.85. compileall_recursion_depth.py

```
import compileall
import re

compileall.compile_dir(
    'examples',
    maxlevels=0,
)
```

В данном случае компилируются только файлы, которые находятся в каталоге, переданном функции compile_dir().

```
$ python3 compileall_recursion_depth.py

Listing 'examples'...
Compiling 'examples/a.py'...
```

16.11.3. Компиляция sys.path

Все файлы исходного кода на языке Python, находящиеся в папках, которые указаны в списке путей sys.path, можно скомпилировать одним вызовом функции compile_path().

Листинг 16.86. compileall_path.py

```
import compileall
import sys

sys.path[:] = ['examples', 'notthere']
print('sys.path =', sys.path)
compileall.compile_path()
```

В данном примере заданное по умолчанию содержимое sys.path изменяется, чтобы избежать ошибок, связанных с возможным отсутствием прав доступа к файлам, во время выполнения сценария, но это не мешает проиллюстрировать поведение по умолчанию. Заметьте, что значение аргумента maxlevels по умолчанию равно 0.

```
$ python3 compileall_path.py

sys.path = ['examples', 'notthere']
Listing 'examples'...
Compiling 'examples/a.py'...
Листинг 'notthere'...
Can't list 'notthere'
```

16.11.4. Компиляция отдельных файлов

Для компиляции одиночного файла, а не всего каталога, следует использовать функцию `compile_file()`.

Листинг 16.87. `compileall_compile_file.py`

```
import compileall
import glob


def show(title):
    print(title)
    for filename in glob.glob('examples/**',
                              recursive=True):
        print('  {}'.format(filename))
    print()


show('Before')

compileall.compile_file('examples/a.py')

show('\nAfter')
```

Первым аргументом должно быть имя файла, заданное в виде полного или относительного пути.

```
$ python3 compileall_compile_file.py

Before
  examples/
  examples/README
  examples/a.py
  examples/subdir
  examples/subdir/b.py

Compiling 'examples/a.py'...

After
  examples/
  examples/README
  examples/__pycache__
  examples/__pycache__/a.cpython-35.pyc
```

```
  examples/a.py
  examples/subdir
  examples/subdir/b.py
```

16.11.5. Компиляция из командной строки

Функцию `compileall` можно вызвать также из командной строки, поэтому она может интегрироваться с системой с помощью средства Makefile. Соответствующий пример приведен ниже.

```
$ python3 -m compileall -h

usage: compileall.py [-h] [-l] [-r RECURSION] [-f] [-q] [-b] [-d
DESTDIR]
                     [-x REGEXP] [-i FILE] [-j WORKERS]
                     [FILE|DIR [FILE|DIR ...]]

Utilities to support installing Python libraries.

positional arguments:
  FILE|DIR              zero or more file and directory names to
compile; if
                        no arguments given, defaults to the
equivalent of -l
                        sys.path

optional arguments:
  -h, --help            show this help message and exit
  -l                    don't recurse into subdirectories
  -r RECURSION          control the maximum recursion level. if
'-l' and '-r'
                        options are specified, then '-r' takes
precedence.
  -f                    force rebuild even if timestamps are up
to date
  -q                    output only error messages; -qq will
suppress the
                        error messages as well.
  -b                    use legacy (pre-PEP3147) compiled file
locations
  -d DESTDIR            directory to prepend to file paths for
use in compile-
                        time tracebacks and in runtime
tracebacks in cases
                        where the source file is unavailable
  -x REGEXP             skip files matching the regular
expression; the regexp
                        is searched for in the full path of each
file
                        considered for compilation
  -i FILE               add all the files and directories listed
in FILE to
```

```
                        the list considered for compilation; if
"-", names are
                        read from stdin
  -j WORKERS, --workers WORKERS
                        Run compileall concurrently
```

Предыдущий пример, в котором пропускался подкаталог subdir, воспроизводится с помощью следующей команды.

```
$ python3 -m compileall -x '/subdir' examples

Listing 'examples'...
Compiling 'examples/a.py'...
Listing 'examples/subdir'...
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю compileall[29].

16.12. pyclbr: обозреватель классов

Модуль pyclbr позволяет находить в исходных файлах на языке Python классы и автономные функции. Информация об именах классов, методов и функций, а также номерах строк собирается с использованием модуля tokenize, не требуя импорта кода.

В приведенных в этом разделе примерах используется следующий исходный файл.

Листинг 16.88. pyclbr_example.py

```
"""Пример источника для pyclbr.
"""


class Base:
    """Это базовый класс.
    """

    def method1(self):
        return


class Sub1(Base):
    """Это первый подкласс.
    """


class Sub2(Base):
    """Это второй подкласс.
    """
```

[29] https://docs.python.org/3.5/library/compileall.html

```
class Mixin:
    """Примесный класс.
    """

    def method2(self):
        return


class MixinUser(Sub2, Mixin):
    """Переопределяет method1 и method2.
    """

    def method1(self):
        return

    def method2(self):
        return

    def method3(self):
        return


def my_function():
    """Отдельная функция.
    """
    return
```

16.12.1. Поиск классов

Модуль `pyclbr` предоставляет две общедоступные функции. Первая из них, `readmodule()`, получает имя модуля в качестве аргумента и возвращает привязку имен класса к объектам `Class`, содержащим метаданные для исходного кода класса.

Листинг 16.89. `pyclbr_readmodule.py`

```
import pyclbr
import os
from operator import itemgetter


def show_class(name, class_data):
    print('Class:', name)
    filename = os.path.basename(class_data.file)
    print('  File: {0} [{1}]'.format(
        filename, class_data.lineno))
    show_super_classes(name, class_data)
    show_methods(name, class_data)
    print()


def show_methods(class_name, class_data):
    for name, lineno in sorted(class_data.methods.items(),
                               key=itemgetter(1)):
```

```
        print('  Method: {0} [{1}]'.format(name, lineno))


def show_super_classes(name, class_data):
    super_class_names = []
    for super_class in class_data.super:
        if super_class == 'object':
            continue
        if isinstance(super_class, str):
            super_class_names.append(super_class)
        else:
            super_class_names.append(super_class.name)
    if super_class_names:
        print('  Super classes:', super_class_names)


example_data = pyclbr.readmodule('pyclbr_example')

for name, class_data in sorted(example_data.items(),
                               key=lambda x: x[1].lineno):
    show_class(name, class_data)
```

Метаданные класса включают имя файла и номер строки, в которой он определен, а также имена суперклассов. Методы класса сохраняются в виде привязки имен методов к номерам строк. В выводе отображаются классы и методы, перечисленные в порядке следования строк их определений в исходном файле.

```
$ python3 pyclbr_readmodule.py

Class: Base
  File: pyclbr_example.py [11]
  Method: method1 [15]

Class: Sub1
  File: pyclbr_example.py [19]
  Super classes: ['Base']

Class: Sub2
  File: pyclbr_example.py [24]
  Super classes: ['Base']

Class: Mixin
  File: pyclbr_example.py [29]
  Method: method2 [33]

Class: MixinUser
  File: pyclbr_example.py [37]
  Super classes: ['Sub2', 'Mixin']
  Method: method1 [41]
  Method: method2 [44]
  Method: method3 [47]
```

16.12.2. Поиск функций

Другая общедоступная функция в модуле pyclbr — readmodule_ex(). Она делает то же самое, что и функция readmodule(), но добавляет функции в результирующий набор.

Листинг 16.90. pyclbr_readmodule_ex.py

```
import pyclbr
import os
from operator import itemgetter

example_data = pyclbr.readmodule_ex('pyclbr_example')

for name, data in sorted(example_data.items(),
                         key=lambda x: x[1].lineno):
    if isinstance(data, pyclbr.Function):
        print('Function: {0} [{1}]'.format(name, data.lineno))
```

Как и объекты Class, каждый объект Function имеет свойства.

```
$ python3 pyclbr_readmodule_ex.py

Function: my_function [51]
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю pyclbr[30].
- inspect (раздел 18.4). Модуль inspect может обнаруживать дополнительные метаданные о классах и функциях, но требует импортировать код.
- tokenize. Модуль tokenize обеспечивает разбор исходного кода на языке Python на лексемы.

16.13. venv: создание виртуальных окружений

Настройка виртуальных окружений (сред) Python, управляемых модулем venv, обеспечивает установку пакетов и выполнение программ таким образом, что они изолируются от других пакетов, установленных в остальной части системы. Поскольку каждое окружение имеет собственный интерпретатор и каталог для установки пакетов, не составляет труда создать несколько окружений, сконфигурированных с использованием различных комбинаций Python и версий пакетов на одном и том же компьютере.

16.13.1. Создание окружения

В основном интерфейсе командной строки модуля venv используется возможность запуска основной функции модуля посредством указания параметра -m.

```
$ python3 -m venv /tmp/demoenv
```

[30] https://docs.python.org/3.5/library/pyclbr.html

В зависимости от способа сборки и формирования пакетов Python существует возможность установки отдельного приложения командной строки `pyvenv`, предназначенного для этих же целей. Приведенная ниже команда дает тот же результат, что и предыдущая.

```
$ pyvenv /tmp/demoenv
```

Команда `-m venv` более предпочтительна, поскольку позволяет указать версию интерпретатора Python. Такой подход гарантирует отсутствие путаницы между используемыми версиями или путями импорта, ассоциируемыми с виртуальным окружением.

16.13.2. Содержимое виртуального окружения

Каждое виртуальное окружение содержит каталог `bin`, в котором установлены локальный интерпретатор и исполняемые сценарии, каталог `include` для файлов, связанных со сборкой C-расширений, и каталог `lib` с отдельным расположением `site-packages` для установки пакетов.

```
$ ls -F /tmp/demoenv

bin/
include/
lib/
pyvenv.cfg
```

Каталог `bin`, заданный по умолчанию, содержит сценарии активизации для нескольких вариантов оболочки Unix. Их можно использовать для установки виртуального окружения в каталоге, включенном в путь поиска командной оболочки, гарантируя тем самым, что оболочка будет выбирать программы, установленные в данном окружении. Несмотря на то что активизация окружения для использования установленных в нем программ не является обязательной, этот прием может быть весьма полезным.

```
$ ls -F /tmp/demoenv/bin

activate
activate.csh
activate.fish
easy_install*
easy_install-3.5*
pip*
pip3*
pip3.5*
python@
python3@
```

Вместо копирования исполняемых файлов наподобие интерпретатора используются символические ссылки, если они поддерживаются платформой. В таком окружении программа `pip` устанавливается в качестве локальной копии, однако интерпретатор является символической ссылкой.

Наконец, окружение включает файл `pyvenv.cfg` с параметрами, описывающими, какими должны быть конфигурация и поведение окружения. Переменная `home` задает расположение интерпретатора Python, в котором модуль `venv` запускался для создания окружения. Переменная `include-systemsite-packages` — это булева переменная, указывающая, должны ли быть видимыми в виртуальном окружении пакеты, установленные вне его на уровне системы. Переменная `version` содержит номер версии Python, используемой для создания окружения.

Листинг 16.91. `pyvenv.cfg`

```
home = /Library/Frameworks/Python.framework/Versions/3.5/bin
include-system-site-packages = false
version = 3.5.2
```

Виртуальное окружение полезнее использовать в сочетании с такими инструментами, как `pip` или `setuptools`, доступными для установки других пакетов, поэтому программа `pyvenv` устанавливает их по умолчанию. Чтобы создать окружение, не включающее эти инструменты, следует использовать в командной строке параметр `--without-pip`.

16.13.3. Использование виртуальных окружений

Как правило, виртуальные окружения применяются для выполнения различных версий программ или тестирования конкретной версии программы с различными версиями ее зависимостей. Например, прежде чем переходить от одной версии Sphinx к другой, целесообразно протестировать входные файлы документации с использованием старой и новой версий. Для этого прежде всего необходимо создать два виртуальных окружения.

```
$ python3 -m venv /tmp/sphinx1
$ python3 -m venv /tmp/sphinx2
```

Затем устанавливаются версии тестируемых инструментов.

```
$ /tmp/sphinx1/bin/pip install Sphinx==1.3.6

Collecting Sphinx==1.3.6
  Using cached Sphinx-1.3.6-py2.py3-none-any.whl
Collecting Jinja2>=2.3 (from Sphinx==1.3.6)
  Using cached Jinja2-2.8-py2.py3-none-any.whl
Collecting Pygments>=2.0 (from Sphinx==1.3.6)
  Using cached Pygments-2.1.3-py2.py3-none-any.whl
Collecting babel!=2.0,>=1.3 (from Sphinx==1.3.6)
  Using cached Babel-2.3.4-py2.py3-none-any.whl
Collecting snowballstemmer>=1.1 (from Sphinx==1.3.6)
  Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
Collecting alabaster<0.8,>=0.7 (from Sphinx==1.3.6)
  Using cached alabaster-0.7.9-py2.py3-none-any.whl
Collecting six>=1.4 (from Sphinx==1.3.6)
  Using cached six-1.10.0-py2.py3-none-any.whl
Collecting sphinx-rtd-theme<2.0,>=0.1 (from Sphinx==1.3.6)
  Using cached sphinx_rtd_theme-0.1.9-py3-none-any.whl
Collecting docutils>=0.11 (from Sphinx==1.3.6)
```

```
  Using cached docutils-0.13.1-py3-none-any.whl
Collecting MarkupSafe (from Jinja2>=2.3->Sphinx==1.3.6)
Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.3.6)
  Using cached pytz-2016.10-py2.py3-none-any.whl
Installing collected packages: MarkupSafe, Jinja2, Pygments,
pytz, babel, snowballstemmer, alabaster, six, sphinx-rtd-theme,
docutils, Sphinx
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Pygments-2.1.3
Sphinx-1.3.6 alabaster-0.7.9 babel-2.3.4 docutils-0.13.1
pytz-2016.10 six-1.10.0 snowballstemmer-1.2.1 sphinx-rtdtheme-
0.1.9

$ /tmp/sphinx2/bin/pip install Sphinx==1.4.4

Collecting Sphinx==1.4.4
  Using cached Sphinx-1.4.4-py2.py3-none-any.whl
Collecting Jinja2>=2.3 (from Sphinx==1.4.4)
  Using cached Jinja2-2.8-py2.py3-none-any.whl
Collecting imagesize (from Sphinx==1.4.4)
  Using cached imagesize-0.7.1-py2.py3-none-any.whl
Collecting Pygments>=2.0 (from Sphinx==1.4.4)
  Using cached Pygments-2.1.3-py2.py3-none-any.whl
Collecting babel!=2.0,>=1.3 (from Sphinx==1.4.4)
  Using cached Babel-2.3.4-py2.py3-none-any.whl
Collecting snowballstemmer>=1.1 (from Sphinx==1.4.4)
  Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
Collecting alabaster<0.8,>=0.7 (from Sphinx==1.4.4)
  Using cached alabaster-0.7.9-py2.py3-none-any.whl
Collecting six>=1.4 (from Sphinx==1.4.4)
  Using cached six-1.10.0-py2.py3-none-any.whl
Collecting docutils>=0.11 (from Sphinx==1.4.4)
  Using cached docutils-0.13.1-py3-none-any.whl
Collecting MarkupSafe (from Jinja2>=2.3->Sphinx==1.4.4)
Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.4.4)
  Using cached pytz-2016.10-py2.py3-none-any.whl
Installing collected packages: MarkupSafe, Jinja2, imagesize,
Pygments, pytz, babel, snowballstemmer, alabaster, six,
docutils, Sphinx
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Pygments-2.1.3
Sphinx-1.4.4 alabaster-0.7.9 babel-2.3.4 docutils-0.13.1
imagesize-0.7.1 pytz-2016.10 six-1.10.0 snowballstemmer-1.2.1
```

Теперь различные версии Sphinx можно запускать из виртуальных окружений по отдельности для их тестирования с использованием одних и тех же входных файлов.

```
$ /tmp/sphinx1/bin/sphinx-build --version

Sphinx (sphinx-build) 1.3.6

$ /tmp/sphinx2/bin/sphinx-build --version

Sphinx (sphinx-build) 1.4.4
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `venv`[31].
- **PEP 405**[32]. *Python Virtual Environments.*
- `virtualenv`[33]. Видоизмененные виртуальные окружения Python, работающие с версиями Python 2 и 3.
- `virtualenvwrapper`[34]. Инструментарий, упрощающий управление большим количеством виртуальных окружений.
- Sphinx[35]. Инструмент для преобразования файлов reStructuredText в файлы HTML, LaTeX, а также файлы других форматов.

16.14. `ensurepip`: программа-установщик пакетов Python

Язык программирования Python создавался по принципу "батарейки входят в комплект" и поставляется с широким набором модулей в составе стандартной библиотеки, но еще большее количество библиотек, фреймворков и инструментов доступно для установки из каталога PyPI (сокр. от *Python Package Index*)[36]. Для установки этих пакетов разработчику требуется программа-установщик `pip`. Установка инструмента, предназначенного для установки других инструментов, уже сама по себе представляет интересную проблему начальной загрузки, которую решает модуль `ensurepip`.

16.14.1. Установка `pip`

В этом примере используется виртуальное окружение, сконфигурированное без использования программы `pip`.

```
$ python3 -m venv --without-pip /tmp/demoenv
$ ls -F /tmp/demoenv/bin

activate
activate.csh
activate.fish
python@
python3@
```

Вызовите модуль `ensurepip` из командной строки с использованием опции `-m` интерпретатора Python. По умолчанию устанавливается экземпляр `pip`, включенный в состав стандартной библиотеки. Впоследствии эту версию можно использовать для установки обновленной версии `pip`. Чтобы сразу же установить

[31] https://docs.python.org/3.5/library/venv.html
[32] www.python.org/dev/peps/pep-0405
[33] https://pypi.python.org/pypi/virtualenv
[34] https://pypi.python.org/pypi/virtualenvwrapper
[35] www.sphinx-doc.org/en/stable/
[36] https://pypi.python.org/pypi

последнюю версию pip, следует использовать параметр --upgrade для модуля ensurepip.

```
$ /tmp/demoenv/bin/python3 -m ensurepip --upgrade

Ignoring indexes: https://pypi.python.org/simple
Collecting setuptools
Collecting pip
Installing collected packages: setuptools, pip
Successfully installed pip-8.1.1 setuptools-20.10.1
```

Эта команда устанавливает pip3 и pip3.5 как отдельные программы в виртуальном окружении вместе с зависимостью setuptools, необходимой для их поддержки.

```
$ ls -F /tmp/demoenv/bin

activate
activate.csh
activate.fish
easy_install-3.5*
pip3*
pip3.5*
python@
python3@
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю ensurepip[37].
- venv (раздел 16.13). Виртуальные окружения.
- **PEP 453**[38]. *Explicit bootstrapping of pip in Python installations.*
- *Installing Python Modules*[39]. Инструкции по установке дополнительных пакетов для использования в Python.
- Python Package Index[40]. Хостинговый сайт, на котором программисты могут размещать написанные ими модули расширений для Python.
- pip[41]. Программа для установки пакетов Python.

[37] https://docs.python.org/3.5/library/ensurepip.html
[38] www.python.org/dev/peps/pep-0453
[39] https://docs.python.org/3.5/installing/index.html#installing-index
[40] https://pypi.python.org/pypi
[41] https://pypi.python.org/pypi/pip

Глава 17

Инструменты среды времени выполнения

В этой главе рассмотрены средства стандартной библиотеки, используя которые программа может взаимодействовать с интерпретатором или окружением, в котором она выполняется.

В процессе своего запуска интерпретатор загружает модуль `site` (раздел 17.1) для настройки параметров конфигурации, специфических для данной установки. Пути поиска импортируемых модулей (пути импорта) конструируются на основе объединения информации из переменных среды, параметров сборки интерпретатора и конфигурационных файлов.

Модуль `sys` (раздел 17.2) — один из самых крупных в стандартной библиотеке. Он содержит функции, обеспечивающие доступ к многочисленным конфигурационным параметрам интерпретатора и системы, включая параметры и ограничения сборки интерпретатора, аргументы командной строки и коды завершения программ, обработку исключений, отладку и управление потоками, механизм импорта модулей, трассировку потока управления во время выполнения программ, а также стандартные потоки ввода-вывода, используемые процессом.

В то время как модуль `sys` ориентирован на управление параметрами интерпретатора, модуль `os` (раздел 17.3) предоставляет доступ к информации об операционной системе. Его можно использовать для создания переносимых интерфейсов к системным вызовам, возвращающим подробную информацию о текущем процессе, включая данные о владельце процесса и переменных среды. Кроме того, модуль `os` включает функции, предназначенные для работы с файловой системой и управлением процессами.

Python часто используют в качестве кроссплатформенного языка программирования для создания переносимых программ. Даже если программа предназначена для выполнения на любой платформе, иногда требуется получение сведений, касающихся операционной системы или архитектуры оборудования текущей платформы. Модуль `platform` (раздел 17.4) предоставляет функции для извлечения соответствующих параметров.

Модуль `resource` (раздел 17.5) позволяет получать информацию о таких параметрах системных ресурсов, как максимальный размер стека процесса или количество одновременно открытых файлов, и изменять их, если в этом возникнет необходимость. Он также позволяет получать отчеты о потреблении системных ресурсов, благодаря чему процесс имеет возможность осуществлять мониторинг их возможной утечки.

Модуль `gc` (раздел 17.6) предоставляет доступ к информации о внутреннем состоянии системы сборки мусора Python. Эти данные могут быть полезными для обнаружения и устранения циклических ссылок, включения и отключения сборщика мусора и настройки его автоматического запуска.

Модуль sysconfig (раздел 17.7) хранит значения переменных времени компиляции из сценариев сборки. Он также может использоваться инструментами сборки и создания пакетов для динамической генерации путей поиска и других параметров.

17.1. site: конфигурирование сайта

Модуль site обрабатывает специфические для сайта конфигурационные параметры, такие как пути импорта модулей.

17.1.1. Пути импорта модулей

Модуль site автоматически импортируется при каждом запуске интерпретатора. Он добавляет в список sys.path специфические для данного сайта имена каталогов, конструируемые путем объединения значений префиксов sys.prefix и sys.exec_prefix с некоторыми суффиксами. Используемые префиксные значения сохраняются в переменной PREFIXES уровня модуля для последующих обращений. В Windows суффиксами служат пустая строка и строка lib/site-packages. В Unix-подобных системах роль этих значений играют строки lib/python$version/site-packages (где вместо $version используются номера старшей и младшей версий интерпретатора, например 3.5) и lib/site-python.

Листинг 17.1. site_import_path.py

```
import sys
import os
import site

if 'Windows' in sys.platform:
    SUFFIXES = [
        '',
        'lib/site-packages',
    ]
else:
    SUFFIXES = [
        'lib/python{}/site-packages'.format(sys.version[:3]),
        'lib/site-python',
    ]

print('Path prefixes:')
for p in site.PREFIXES:
    print('  ', p)

for prefix in sorted(set(site.PREFIXES)):
    print()
    print(prefix)
    for suffix in SUFFIXES:
        print()
        print(' ', suffix)
        path = os.path.join(prefix, suffix).rstrip(os.sep)
        print('   exists :', os.path.exists(path))
        print('   in path:', path in sys.path)
```

Тестируется каждый из путей, получаемых составлением указанных комбинаций, и если результирующий путь существует, то он добавляется в переменную sys.path. Ниже представлен вывод, полученный для версии фреймворка Python, установленной в системе Mac OS X.

```
$ python3 site_import_path.py

Path prefixes:
   /Library/Frameworks/Python.framework/Versions/3.5
   /Library/Frameworks/Python.framework/Versions/3.5

/Library/Frameworks/Python.framework/Versions/3.5

  lib/python3.5/site-packages
   exists : True
   in path: True

  lib/site-python
   exists : False
   in path: False
```

17.1.2. Пользовательские каталоги

В дополнение к глобальным путям, ведущим к каталогу site-packages, модуль site отвечает за добавление в путь поиска модулей местоположений, выделенных для конкретных пользователей. Все пользовательские пути базируются на каталоге USER_BASE, который обычно располагается в части файловой системы, принадлежащей (и предоставляющей право записи) текущему пользователю. Каталог USER_BASE содержит каталог site-packages, путь к которому доступен через переменную USER_SITE.

Листинг 17.2. site_user_base.py

```
import site

print('Base:', site.USER_BASE)
print('Site:', site.USER_SITE)
```

Путь, определенный в переменной USER_SITE, создается с использованием описанных ранее платформозависимых значений суффиксов.

```
$ python3 site_user_base.py

Base: /Users/dhellmann/.local
Site: /Users/dhellmann/.local/lib/python3.5/site-packages
```

Базовый каталог пользователя может устанавливаться посредством переменной среды PYTHONUSERBASE и имеет платформозависимые значения по умолчанию (~/Python$version/site-packages для Windows и ~/.local для систем, отличных от Windows).

```
$ PYTHONUSERBASE=/tmp/$USER python3 site_user_base.py

Base: /tmp/dhellmann
Site: /tmp/dhellmann/lib/python3.5/site-packages
```

При некоторых обстоятельствах (например, если процесс выполняется с использованием идентификатора другого пользователя, отличающегося от идентификатора фактического пользователя, запустившего процесс) пользовательский каталог отключается. Приложение может определить, так ли это, проверив значение переменной `ENABLE_USER_SITE`.

Листинг 17.3. `site_enable_user_site.py`

```
import siteimport site

status = {
    None: 'Disabled for security',
    True: 'Enabled',
    False: 'Disabled by command-line option',
}

print('Flag   :', site.ENABLE_USER_SITE)
print('Meaning:', status[site.ENABLE_USER_SITE])
```

17.1.3. Конфигурационные файлы путей

Для добавления путей в список путей импорта можно также использовать *конфигурационные файлы путей*. Конфигурационный файл пути — это обычный текстовый файл с расширением *.pth*. Каждая строка такого файла может содержать одну из следующих четырех форм:

- полный или относительный путь к другому местоположению, которое должно быть добавлено в список путей импорта;
- исполняемая инструкция Python; все подобные строки должны начинаться с инструкции `import`;
- пустая строка, которая игнорируется;
- строка, начинающаяся с символа `#`, который трактуется как комментарий и игнорируется.

Конфигурационные файлы путей могут использоваться для расширения путей импорта, включая в них расположения, не добавляемые автоматически. Например, пакет `setuptools` добавляет путь в файл `easy-install.pth` во время установки пакета в режиме разработки с помощью команды `python setup.py develop`.

Функция, выполняющая расширение списка путей поиска модулей в переменной `sys.path`, является общедоступной, и поэтому может быть использована в примерах программ для того, чтобы показать, как работают конфигурационные файлы путей. Предположим, что в каталоге `with_modules` содержится файл `mymodule.py` с инструкциями `print`, отображающими информацию о том, как импортировался модуль.

Листинг 17.4. `with_modules/mymodule.py`

```
import os
print('Loaded {} from {}'.format(
    __name__, __file__[len(os.getcwd()) + 1:])
)
```

Следующий сценарий демонстрирует, как функция `addsitedir()` расширяет список путей импорта, чтобы интерпретатор мог найти нужный модуль.

Листинг 17.5. `site_addsitedir.py`

```
import site
import os
import sys

script_directory = os.path.dirname(__file__)
module_directory = os.path.join(script_directory, sys.argv[1])

try:
    import mymodule
except ImportError as err:
    print('Could not import mymodule:', err)

print()
before_len = len(sys.path)
site.addsitedir(module_directory)
print('New paths:')
for p in sys.path[before_len:]:
    print(p.replace(os.getcwd(), '.'))  # использовать сокращенное
                                        # имя каталога

print()
import mymodule
```

После добавления каталога, содержащего нужный модуль, в переменную `sys.path` сценарий может без проблем импортировать модуль `mymodule`.

```
$ python3 site_addsitedir.py with_modules

Could not import mymodule: No module named 'mymodule'

New paths:
./with_modules

Loaded mymodule from with_modules/mymodule.py
```

Изменения путей, вносимые функцией `addsitedir()`, не ограничиваются простым добавлением аргумента в `sys.path`. Если предоставленный ей каталог включает файлы с расширением *.pth*, то они загружаются в качестве конфигурационных файлов импорта модулей. Предположим, каталог имеет структуру

```
with_pth
 pymotw.pth
```

```
 subdir
    mymodule.py
```

Файл `with_pth/pymotw.pth` содержит следующие строки

```
# Добавить в путь один подкаталог
./subdir
```

Тогда импорт модуля `with_pth/subdir/mymodule.py` может быть обеспечен путем добавления каталога `with_pth` в качестве каталога сайта, несмотря на то что данный модуль не содержится в этом каталоге, поскольку как `with_pth`, так и `with_pth/subdir` добавляются в список путей импорта.

```
$ python3 site_addsitedir.py with_pth

Could not import mymodule: No module named 'mymodule'

New paths:
./with_pth
./with_pth/subdir

Loaded mymodule from with_pth/subdir/mymodule.py
```

Если каталог сайта содержит несколько *.pth*-файлов, то они обрабатываются в алфавитном порядке.

```
$ ls -F multiple_pth

a.pth
b.pth
from_a/
from_b/

$ cat multiple_pth/a.pth

./from_a

$ cat multiple_pth/b.pth

./from_b
```

В данном случае выбирается модуль, находящийся в каталоге `multiple_pth/from_a`, поскольку файл *a.pth* читается раньше файла *b.pth*.

```
$ python3 site_addsitedir.py multiple_pth

Could not import mymodule: No module named 'mymodule'

New paths:
./multiple_pth
./multiple_pth/from_a
./multiple_pth/from_b
```

Загружаемым модулем является `multiple_pth/from_a/mymodule.py`.

17.1.4. Настройка конфигурации сайта

Модуль site отвечает за загрузку параметров всего сайта, определенных владельцем локального сайта в модуле sitecustomize. В частности, модуль sitecustomize можно использовать для расширения списка путей импорта и активизации инструментов, предназначенных для измерения степени покрытия кода, профилирования кода, и многих других инструментов разработки.

Например, в приведенном ниже листинге сценарий sitecustomize.py добавляет в список путей импорта каталог, зависящий от текущей платформы. Специфический для платформы путь в /opt/python добавляется в список путей импорта, поэтому любые установленные в нем пакеты могут быть импортированы. Система такого типа удобна для организации совместного использования пакетов, содержащих модули скомпилированных расширений, всеми хостами сети посредством общей файловой системы. Тогда на каждом хосте достаточно установить только сценарий sitecustomize.py, а доступ к остальным пакетам обеспечит файловый сервер.

Листинг 17.6. with_sitecustomize/sitecustomize.py

```
print('Loading sitecustomize.py')

import site
import platform
import os
import sys

path = os.path.join('/opt',
                    'python',
                    sys.version[:3],
                    platform.platform(),
                    )
print('Adding new path', path)

site.addsitedir(path)
```

Продемонстрировать тот факт, что модуль sitecustomize.py импортируется до того, как Python начнет выполнять код программы, можно с помощью следующего простого сценария:

Листинг 17.7. with_sitecustomize/site_sitecustomize.py

```
import sys

print('Running main program from\n{}'.format(sys.argv[0]))

print('End of path:', sys.path[-1])
```

Поскольку модуль sitecustomize предназначен для конфигурирования всей системы, он должен устанавливаться в одном из каталогов, заданных по умолчанию (обычно в каталоге site-packages). В данном примере гарантией того, что этот модуль будет успешно найден, служит явная установка переменной среды PYTHONPATH.

```
$ PYTHONPATH=with_sitecustomize python3 with_sitecustomize/sit\
e_sitecustomize.py

Loading sitecustomize.py
Adding new path /opt/python/3.5/Darwin-15.6.0-x86_64-i386-64bit
Running main program from
with_sitecustomize/site_sitecustomize.py
End of path: /opt/python/3.5/Darwin-15.6.0-x86_64-i386-64bit
```

17.1.5. Настройка пользовательской конфигурации

По аналогии с модулем `sitecustomize` модуль `usercustomize` может быть использован для задания специфических для пользователя конфигурационных параметров при каждом запуске интерпретатора. Модуль `usercustomize` загружается после модуля `sitecustomize`, что позволяет переопределять параметры, установленные для сайта в целом.

В случае окружений, в которых домашний каталог пользователя совместно используется несколькими серверами, работающими под управлением разных операционных систем или их версий, стандартный механизм пользовательских каталогов для конфигурирования специфических для пользователя вариантов установки пакетов может не сработать. В подобных случаях вместо него можно использовать дерево каталогов, специфических для платформы.

Листинг 17.8. `with_usercustomize/usercustomize.py`

```
print('Loading usercustomize.py')

import site
import platform
import os
import sys

path = os.path.expanduser(os.path.join('~',
                                       'python',
                                       sys.version[:3],
                                       platform.platform(),
                                       ))
print('Adding new path', path)

site.addsitedir(path)
```

Продемонстрировать тот факт, что модуль `usercustomize.py` импортируется до того, как Python начнет выполнять код программы, можно с помощью простого сценария, аналогичного тому, который использовался ранее для тех же целей применительно к модулю `sitecustomize.py`.

Листинг 17.9. `with_usercustomize/site_usercustomize.py`

```
import sys

print('Running main program from\n{}'.format(sys.argv[0]))

print('End of path:', sys.path[-1])
```

Поскольку модуль usercustomize предназначен для конфигурирования параметров, специфических для пользователя, он должен устанавливаться в одном из каталогов, заданных по умолчанию для конкретного пользователя, а не в каталогах, заданных по умолчанию для сайта в целом. Неплохим кандидатом на эту роль является каталог по умолчанию, заданный переменной USER_BASE. В данном примере гарантией того, что этот модуль будет успешно найден, служит явная установка переменной среды PYTHONPATH.

```
$ PYTHONPATH=with_usercustomize python3 with_usercustomize/site\
_usercustomize.py

Loading usercustomize.py
Adding new path /Users/dhellmann/python/3.5/Darwin-15.5.0-x86_64\
-i386-64bit
Running main program from
with_usercustomize/site_usercustomize.py
End of path: /Users/dhellmann/python/3.5/Darwin-15.5.0-x86_64\
-i386-64bit
```

Если это средство отключено, то модуль usercustomize не будет импортироваться, независимо от того, установлен он в пользовательском каталоге сайта или не установлен.

```
$ PYTHONPATH=with_usercustomize python3 -s with_usercustomize/s\
ite_usercustomize.py

Running main program from
with_usercustomize/site_usercustomize.py
End of path: /Users/dhellmann/Envs/pymotw35/lib/python3.5/sitepackages
```

17.1.6. Отключение модуля site

В целях поддержки обратной совместимости с версиями Python, предшествующими добавлению функциональности автоматического импорта, интерпретатор принимает параметр -S.

```
$ python3 -S site_import_path.py

Path prefixes:
   /Users/dhellmann/Envs/pymotw35/bin/..
   /Users/dhellmann/Envs/pymotw35/bin/..

/Users/dhellmann/Envs/pymotw35/bin/..

  lib/python3.5/site-packages
   exists : True
   in path: False

  lib/site-python
   exists : False
   in path: False
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `site`[1].
- Раздел 17.2.6. Описание того, как работают пути поиска модулей, определенные в модуле `sys`.
- `setuptools`[2]. Описание библиотеки средств для создания пакетов и инструмента установки пакетов `easy_install`.
- *Running code at Python startup* (Ned Batchelder)[3]. Статья в блоге, в которой обсуждаются способы выполнения пользовательского кода инициализации до того, как интерпретатор Python запустит основную программу.

17.2. `sys`: настройка конфигурационных параметров, специфических для системы

Модуль `sys` содержит средства, обеспечивающие определение или изменение конфигурации интерпретатора во время выполнения и взаимодействие текущей программы с операционным окружением.

17.2.1. Параметры интерпретатора

Модуль `sys` содержит атрибуты и функции, обеспечивающие доступ к конфигурационным параметрам интерпретатора времени компиляции и времени выполнения.

17.2.1.1. Информация о версии интерпретатора

Номер версии интерпретатора доступен в нескольких формах. Атрибут `sys.version` — это строка, предназначенная для чтения человеком, которая обычно включает полный номер версии интерпретатора и дополнительную информацию о дате сборки, компиляторе и платформе. Поскольку атрибут `sys.hexversion` содержит обычное целое число, его проще использовать для проверки версии интерпретатора. Если отформатировать это число с помощью функции `hex()`, то сразу станет ясно, что части его значения берутся из информации о версии, представленной в более удобочитаемой форме в атрибуте `sys.version_info` (именованный кортеж из пяти элементов, представляющий номер версии). Отдельный номер версии C API, используемой данным интерпретатором, хранится в атрибуте `sys.api_version`.

Листинг 17.10. `sys_version_values.py`

```
import sys

print('Version info:')
print()
print('sys.version      =', repr(sys.version))
print('sys.version_info =', sys.version_info)
```

[1] https://docs.python.org/3.5/library/site.html

[2] https://setuptools.readthedocs.io/en/latest/index.html

[3] http://nedbatchelder.com/blog/201001/running_code_at_python_startup.html

```
print('sys.hexversion   =', hex(sys.hexversion))
print('sys.api_version  =', sys.api_version)
```

Все значения зависят от фактической версии интерпретатора, используемой для выполнения примера программы.

```
$ python3 sys_version_values.py

Version info:

sys.version      = '3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016,
10:47:25) \n[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]'
sys.version_info = sys.version_info(major=3, minor=5, micro=2,
releaselevel='final', serial=0)
sys.hexversion   = 0x30502f0
sys.api_version  = 1013
```

Информация о платформе операционной системы, использованной для сборки интерпретатора, хранится в атрибуте `sys.platform`.

Листинг 17.11. `sys_platform.py`

```
import sys

print('This interpreter was built for:', sys.platform)
```

Для большинства систем Unix это значение создается путем объединения результата выполнения команды `uname -s` с первой частью версии, содержащейся в результате выполнения команды `uname -r`. В случае других операционных систем используется таблица значений, заданная в коде.

```
$ python3 sys_platform.py

This interpreter was built for: darwin
```

Дополнительные ссылки

- Идентификаторы платформы[4]. Жестко запрограммированные значения атрибута `sys.platform` для систем, не имеющих утилиты `uname`.

17.2.1.2. Реализация интерпретатора

Интерпретатор CPython — это одна из нескольких реализаций языка Python. Информация о текущей реализации, необходимая тем библиотекам, которым приходится учитывать различия в интерпретаторах, предоставляется объектом `sys.implementation`.

Листинг 17.12. `sys_implementation.py`

```
import sys

print('Name:', sys.implementation.name)
```

[4] https://docs.python.org/3/library/sys.html#sys.platform

```
print('Version:', sys.implementation.version)
print('Cache tag:', sys.implementation.cache_tag)
```

Значение `sys.implementation.version` совпадает со значением `sys.version_info` для CPython, но отличается от него в случае других интерпретаторов.

```
$ python3 sys_implementation.py

Name: cpython
Version: sys.version_info(major=3, minor=5, micro=2, releaseleve
l='final', serial=0)
Cache tag: cpython-35
```

Дополнительные ссылки

- **PEP 421**[5]. Добавление атрибута `sys.implementation`.

17.2.1.3. Параметры командной строки

Интерпретатор CPython поддерживает несколько необязательных параметров командной строки, управляющих его поведением (табл. 17.1). Некоторые из этих параметров доступны для программ через объект `sys.flags`.

Таблица 17.1. Флаги командной строки CPython

Флаг	Значение
`-B`	Не сохранять *.py[co]*-файлы компилированного байт-кода при импорте модулей
`-b`	Выводить предупреждающие сообщения о преобразовании байтов в строки без декодирования и сравнении байтов со строками
`-bb`	Выдавать сообщения об ошибке вместо предупреждений при манипулировании байтовыми объектами
`-d`	Выполнить отладку вывода синтаксического анализатора
`-E`	Игнорировать переменные среды `PYTHON*` (такие, как `PYTHONPATH`)
`-i`	Запустить интерактивный сеанс после выполнения сценария
`-O`	Оптимизировать байт-код
`-OO`	Удалить строки документации байт-кода наряду с его оптимизацией, задаваемой с помощью опции `-O`
`-s`	Не добавлять пользовательский каталог `site` в список путей `sys.path`
`-S`	Не выполнять инструкцию `"import site"` во время инициализации
`-t`	Выводить предупреждающие сообщения о непоследовательном использовании табуляции
`-tt`	Выводить сообщения об ошибках при непоследовательном использовании табуляции
`-v`	Вывести подробную информацию

Листинг 17.13. `sys_flags.py`

```
import sys

if sys.flags.bytes_warning:
```

[5] www.python.org/dev/peps/pep-0421

```
    print('Warning on bytes/str errors')
if sys.flags.debug:
    print('Debuging')
if sys.flags.inspect:
    print('Will enter interactive mode after running')
if sys.flags.optimize:
    print('Optimizing byte-code')
if sys.flags.dont_write_bytecode:
    print('Not writing byte-code files')
if sys.flags.no_site:
    print('Not importing "site"')
if sys.flags.ignore_environment:
    print('Ignoring environment')
if sys.flags.verbose:
    print('Verbose mode')
```

Поэкспериментируйте со сценарием `sys_flags.py`, чтобы больше узнать о том, как параметры командной строки связаны со значениями флагов.

```
$ python3 -S -E -b sys_flags.py

Warning on bytes/str errors
Not importing "site"
Ignoring environment
```

17.2.1.4. Использование кодировки Unicode по умолчанию

Чтобы получить имя кодировки Unicode, которая по умолчанию используется интерпретатором, следует вызвать функцию `getdefaultencoding()`. Это значение устанавливается во время запуска интерпретатора и не может быть изменено на протяжении интерактивного сеанса.

В случае некоторых операционных систем внутренняя кодировка, используемая по умолчанию, и кодировка файловой системы могут отличаться, поэтому для получения соответствующего параметра файловой системы следует использовать другой способ. Функция `getfilesystemencoding()` возвращает значение, специфическое для ОС (но не для файловой системы).

Листинг 17.14. `sys_unicode.py`

```
import sys

print('Default encoding :', sys.getdefaultencoding())
print('File system encoding :', sys.getfilesystemencoding())
```

Вместо того чтобы полагаться на глобальные настройки кодировки по умолчанию, большинство экспертов в области Unicode рекомендуют явно задавать кодировку в приложении. Такой подход обеспечивает два преимущества: более чистую обработку различных кодировок Unicode для различных источников данных и снижение количества предположений об используемых в программном коде кодировок.

```
$ python3 sys_unicode.py

Default encoding : utf-8
File system encoding : utf-8
```

17.2.1.5. Интерактивная командная подсказка

Интерактивный интерпретатор использует две отдельные разновидности приглашений к вводу (командных подсказок) для начального уровня ввода (`ps1`) и продолжения ввода многострочных инструкций (`ps2`). Эти значения используются лишь интерактивным интерпретатором.

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>>
```

Любое из этих приглашений можно заменить другой строкой.

```
>>> sys.ps1 = '::: '
::: sys.ps2 = '~~~ '
::: for i in range(3):
~~~ print i
~~~
0
1
2
:::
```

В качестве приглашения можно использовать любой объект, преобразуемый в строку (посредством вызова метода `__str__`).

Листинг 17.15. `sys_ps1.py`

```
import sys


class LineCounter:

    def __init__(self):
        self.count = 0

    def __str__(self):
        self.count += 1
        return '({:3d})> '.format(self.count)
```

Объект `LineCounter` отслеживает, сколько раз использовалось приглашение, поэтому номер приглашения каждый раз увеличивается на единицу.

```
$ python

Python 3.4.2 (v3.4.2:ab2c023a9432, Oct 5 2014, 20:42:22)
```

```
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> from sys_ps1 import LineCounter
>>> import sys
>>> sys.ps1 = LineCounter()
( 1)>
( 2)>
( 3)>
```

17.2.1.6. Перехват вывода

Функция `sys.displayhook()` вызывается интерактивным интерпретатором всякий раз, когда пользователь вводит выражение. Результат вычисления выражения передастся этой функции в качестве единственного аргумента.

Листинг 17.16. `sys_displayhook.py`

```
import sys


class ExpressionCounter:

    def __init__(self):
        self.count = 0
        self.previous_value = self

    def __call__(self, value):
        print()
        print('  Previous:', self.previous_value)
        print('  New     :', value)
        print()
        if value != self.previous_value:
            self.count += 1
            sys.ps1 = '({:3d})> '.format(self.count)
        self.previous_value = value
        sys.__displayhook__(value)


print('installing')
sys.displayhook = ExpressionCounter()
```

Значение по умолчанию (сохраненное в `sys.__displayhook__`) выводится в качестве результата в стандартный поток `stdout` и сохраняется в переменной `_`, на которую впоследствии можно легко ссылаться.

```
$ python3
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct 5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> import sys_displayhook
installing
```

```
>>> 1 + 2

  Previous: <sys_displayhook.ExpressionCounter
  object at 0x1021035f8>
  New     : 3

3
(  1)> 'abc'

  Previous: 3
  New     : abc

'abc'
(  2)> 'abc'

  Previous: abc
  New     : abc

'abc'
(  2)> 'abc' * 3

  Previous: abc
  New     : abcabcabc

'abcabcabc'
(  3)>
```

17.2.1.7. Каталог установки интерпретатора

Путь к фактическому каталогу установки интерпретатора доступен через атрибут `sys.executable` во всех системах, для которых понятие пути к интерпретатору имеет смысл. Эта информация полезна тем, что позволяет убедиться в корректности используемой версии интерпретатора и получить представление о тех путях, которые могли быть установлены на основании данного пути к расположению интерпретатора.

Атрибут `sys.prefix` ссылается на родительский каталог установки интерпретатора. Как правило, он включает каталоги `bin` и `lib`, предназначенные для размещения исполняемых файлов и модулей соответственно.

Листинг 17.17. `sys_locations.py`

```
import sys

print('Interpreter executable:')
print(sys.executable)
print('\nInstallation prefix:')
print(sys.prefix)
```

Приведенный ниже вывод был получен на компьютере Mac, где выполняется фреймворк Python, установленный с помощью пакета, загруженного с сайта `python.org`.

```
$ python3 sys_locations.py

Interpreter executable:
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3

Installation prefix:
/Library/Frameworks/Python.framework/Versions/3.5
```

17.2.2. Среда времени выполнения

Модуль sys предоставляет низкоуровневые программные интерфейсы для взаимодействия с системой вне приложения за счет обработки аргументов командной строки, получения доступа к пользовательскому вводу и передачи сообщений и информации о состоянии пользователю.

17.2.2.1. Аргументы командной строки

Аргументы, предназначенные для интерпретатора, в нем же и обрабатываются — они не передаются в выполняющуюся программу. Остальные параметры, в том числе имя самого сценария, сохраняются в атрибуте sys.argv на тот случай, если они действительно потребуются программе.

Листинг 17.18. sys_argv.py

```
import sys

print('Arguments:', sys.argv)
```

В третьем из представленных ниже примеров параметр -u распознается интерпретатором и не передается программе.

```
$ python3 sys_argv.py

Arguments: ['sys_argv.py']

$ python3 sys_argv.py -v foo blah

Arguments: ['sys_argv.py', '-v', 'foo', 'blah']

$ python3 -u sys_argv.py

Arguments: ['sys_argv.py']
```

Дополнительные ссылки

- argparse (раздел 14.1). Модуль для синтаксического анализа аргументов командной строки.

17.2.2.2. Потоки ввода-вывода

Следуя парадигме Unix, программы на языке Python могут обращаться к трем дескрипторам файлов, заданным по умолчанию.

Листинг 17.19. sys_stdio.py

```
import sys

print('STATUS: Reading from stdin', file=sys.stderr)

data = sys.stdin.read()

print('STATUS: Writing data to stdout', file=sys.stderr)

sys.stdout.write(data)
sys.stdout.flush()

print('STATUS: Done', file=sys.stderr)
```

stdin — стандартный поток ввода, используемый для чтения входных данных, который обычно ассоциирован с консолью, но также может поступать из других программ посредством конвейера. stdout — стандартный поток вывода, обеспечивающий вывод данных для пользователя (на консоль) или передачу данных следующей команде конвейера. stderr — стандартный поток ошибок, предназначенный для вывода предупреждений и сообщений об ошибках.

```
$ cat sys_stdio.py | python3 -u sys_stdio.py

STATUS: Reading from stdin
STATUS: Writing data to stdout
#!/usr/bin/env python3

#end_pymotw_header
import sys

print('STATUS: Reading from stdin', file=sys.stderr)

data = sys.stdin.read()

print('STATUS: Writing data to stdout', file=sys.stderr)

sys.stdout.write(data)
sys.stdout.flush()

print('STATUS: Done', file=sys.stderr)
STATUS: Done
```

Дополнительные ссылки

- subprocess (см. раздел 10.1) и pipes. Оба модуля содержат средства конвейеризации программ.

17.2.2.3. Коды завершения

Чтобы вернуть код завершения программы, следует передать целое число функции sys.exit().

Листинг 17.20. sys_exit.py

```
import sys

exit_code = int(sys.argv[1])
sys.exit(exit_code)
```

Ненулевое значение означает, что программа завершилась с ошибкой.

```
$ python3 sys_exit.py 0 ; echo "Exited $?"

Exited 0

$ python3 sys_exit.py 1 ; echo "Exited $?"

Exited 1
```

17.2.3. Управление памятью и ограничения

Модуль sys включает несколько функций, предназначенных для управления использованием памяти и получения соответствующей информации.

17.2.3.1. Счетчики ссылок

В базовой реализации Python (CPython) для управления памятью используются *счетчики ссылок* и механизм *сборки мусора*. Объект автоматически помечается к удалению, если значение его счетчика ссылок уменьшается до нуля. Для проверки счетчика ссылок существующего объекта используется функция getrefcount().

Листинг 17.21. sys_getrefcount.py

```
import sys

one = []
print('At start :', sys.getrefcount(one))

two = one

print('Second reference :', sys.getrefcount(one))

del two

print('After del :', sys.getrefcount(one))
```

Выводимое значение счетчика на единицу превышает ожидаемое, поскольку имеется временная ссылка на объект, которая хранится в самой функции getrefcount().

```
$ python3 sys_getrefcount.py

At start         : 2
Second reference : 3
After del        : 2
```

Дополнительные ссылки

- gc (раздел 17.6). Управление сборщиком мусора посредством функций, предоставляемых модулем gc.

17.2.3.2. Размер объекта

Знание количества ссылок на объект может облегчить разработчику обнаружение циклических ссылок или идентификацию источника утечки памяти, но этой информации недостаточно для того, чтобы определить, какие объекты потребляют наибольший объем памяти. Для этого необходимо знать размеры объектов.

Листинг 17.22. sys_getsizeof.py

```
import sys


class MyClass:
    pass

objects = [
    [], (), {}, 'c', 'string', b'bytes', 1, 2.3,
    MyClass, MyClass(),
]

for obj in objects:
    print('{:>10} : {}'.format(type(obj).__name__,
                               sys.getsizeof(obj)))
```

Функция getsizeof() выводит отчет о размере объекта в байтах.

```
$ python3 sys_getsizeof.py
      list : 64
     tuple : 48
      dict : 288
       str : 50
       str : 55
     bytes : 38
       int : 28
     float : 24
      type : 1016
   MyClass : 56
```

Выводимое значение размера пользовательского класса не включает размер значений атрибутов.

Листинг 17.23. sys_getsizeof_object.py

```
import sys


class WithoutAttributes:
    pass


class WithAttributes:
```

```
    def __init__(self):
        self.a = 'a'
        self.b = 'b'
        return


without_attrs = WithoutAttributes()
print('WithoutAttributes:', sys.getsizeof(without_attrs))

with_attrs = WithAttributes()
print('WithAttributes:', sys.getsizeof(with_attrs))
```

Такой подход может создавать ложную картину того, какой фактический объем памяти занимает данный объект.

```
$ python3 sys_getsizeof_object.py

WithoutAttributes: 56
WithAttributes: 56
```

Более точную оценку объема памяти, используемого классом, обеспечивает метод `__sizeof__()`, вычисляющий эту величину путем суммирования размеров всевозможных атрибутов объекта.

Листинг 17.24. sys_getsizeof_custom.py

```
import sys


class WithAttributes:
    def __init__(self):
        self.a = 'a'
        self.b = 'b'
        return

    def __sizeof__(self):
        return object.__sizeof__(self) + \
            sum(sys.getsizeof(v) for v in self.__dict__.values())


my_inst = WithAttributes()
print(sys.getsizeof(my_inst))
```

При таком варианте базовый размер объекта складывается с суммарным размером всех атрибутов, сохраняющихся во внутреннем словаре `__dict__`.

```
$ python3 sys_getsizeof_custom.py

156
```

17.2.3.3. Рекурсия

Разрешение выполнения бесконечных рекурсий в приложении Python может приводить к переполнению стека самого интерпретатора и аварийному

завершению программы. Чтобы избежать возникновения подобных ситуаций, интерпретатор предоставляет возможность управлять максимальной глубиной рекурсии посредством использования функций `setrecursionlimit()` и `getrecursionlimit()`.

Листинг 17.25. `sys_recursionlimit.py`

```
import sys

print('Initial limit:', sys.getrecursionlimit())

sys.setrecursionlimit(10)

print('Modified limit:', sys.getrecursionlimit())


def generate_recursion_error(i):
    print('generate_recursion_error({})'.format(i))
    generate_recursion_error(i + 1)


try:
    generate_recursion_error(1)
except RuntimeError as err:
    print('Caught exception:', err)
```

Как только размер стека достигает предельной глубины рекурсии, интерпретатор возбуждает исключение `RuntimeError`, которое программа может обработать для разрешения конфликтной ситуации.

```
$ python3 sys_recursionlimit.py

Initial limit: 1000
Modified limit: 10
generate_recursion_error(1)
generate_recursion_error(2)
generate_recursion_error(3)
generate_recursion_error(4)
generate_recursion_error(5)
generate_recursion_error(6)
generate_recursion_error(7)
generate_recursion_error(8)
Caught exception: maximum recursion depth exceeded while calling
a Python object
```

17.2.3.4. Максимальные значения

Наряду с конфигурируемыми параметрами времени выполнения модуль `sys` включает переменные, определяющие максимальные значения для типов данных, которые могут меняться от системы к системе.

Листинг 17.26. sys_maximums.py

```
import sys

print('maxsize   :', sys.maxsize)
print('maxunicode:', sys.maxunicode)
```

Атрибут `maxsize` — это максимально возможный размер списка, словаря, строки или другой структуры данных, диктуемый типом `size_t` языка C. Атрибут `maxunicode` — это наибольшее целое число, определяющее кодовую точку Unicode, которая поддерживается интерпретатором в его текущей конфигурации.

```
$ python3 sys_maximums.py

maxsize : 9223372036854775807
maxunicode: 1114111
```

17.2.3.5. Значения с плавающей точкой

Структура `float_info` содержит информацию о представлении типа чисел с плавающей точкой, поддерживаемом интерпретатором, которое основано на реализации этого типа в базовой системе.

Листинг 17.27. sys_float_info.py

```
import sys

print('Smallest difference (epsilon):', sys.float_info.epsilon)
print()
print('Digits (dig)              :', sys.float_info.dig)
print('Mantissa digits (mant_dig):', sys.float_info.mant_dig)
print()
print('Maximum (max):', sys.float_info.max)
print('Minimum (min):', sys.float_info.min)
print()
print('Radix of exponents (radix):', sys.float_info.radix)
print()
print('Maximum exponent for radix (max_exp):',
      sys.float_info.max_exp)
print('Minimum exponent for radix (min_exp):',
      sys.float_info.min_exp)
print()
print('Max. exponent power of 10 (max_10_exp):',
      sys.float_info.max_10_exp)
print('Min. exponent power of 10 (min_10_exp):',
      sys.float_info.min_10_exp)
print()
print('Rounding for addition (rounds):', sys.float_info.rounds)
```

Эти значения зависят от компилятора и базовой системы. Приведенный ниже вывод соответствует системе OS X 10.9.5 с процессором Intel Core i7.

```
$ python3 sys_float_info.py
Smallest difference (epsilon): 2.220446049250313e-16
```

```
Digits (dig)              : 15
Mantissa digits (mant_dig): 53

Maximum (max): 1.7976931348623157e+308
Minimum (min): 2.2250738585072014e-308

Radix of exponents (radix): 2

Maximum exponent for radix (max_exp): 1024
Minimum exponent for radix (min_exp): -1021

Max. exponent power of 10 (max_10_exp): 308
Min. exponent power of 10 (min_10_exp): -307

Rounding for addition (rounds): 1
```

Дополнительные ссылки

- Более подробная информация об этих параметрах содержится в заголовочном файле C *float.h* локального компилятора.

17.2.3.6. Целочисленные значения

Структура `int_info` содержит информацию о внутреннем представлении целых чисел, используемом интерпретатором.

Листинг 17.28. `sys_int_info.py`

```
import sys

print('Number of bits used to hold each digit:',
      sys.int_info.bits_per_digit)
print('Size in bytes of C type used to hold each digit:',
      sys.int_info.sizeof_digit)
```

Приведенный ниже вывод соответствует системе OS X 10.9.5 с процессором Intel Core i7.

```
$ python3 sys_int_info.py

Number of bits used to hold each digit: 30
Size in bytes of C type used to hold each digit: 4
```

Тип C, внутренне используемый для хранения целых чисел, определяется при сборке интерпретатора. 64-разрядные архитектуры используют 30-битовые целые числа по умолчанию, но такие целые числа можно использовать и в случае 32-разрядных архитектур, установив флаг конфигурации `--enable-big-digits`.

Дополнительные ссылки

- Статья *Build and C API Changes*[6] из раздела *What's New in Python 3.1* документации Python.

[6] https://docs.python.org/3.1/whatsnew/3.1.html#build-and-c-api-changes

17.2.3.7. Порядок байтов

Для атрибута byteorder устанавливается значение, соответствующее машинному порядку следования байтов.

Листинг 17.29. sys_byteorder.py

```
import sys

print(sys.byteorder)
```

Обратному порядку байтов соответствует значение big (big-endian), прямому — значение little (little-endian).

```
$ python3 sys_byteorder.py

little
```

Дополнительные ссылки

- Википедия: *Порядок байтов*[7]. Описание систем с прямым и обратным порядком следования байтов.
- array (раздел 2.3) и struct (раздел 2.7). Другие модули, зависящие от порядка следования байтов.
- *float.h*. Заголовочный файл C локального компилятора, содержащий более подробную информацию об этих настройках.

17.2.4. Обработка исключений

Модуль sys включает средства, предназначенные для перехвата и обработки исключений.

17.2.4.1. Необработанные исключения

Структура многих приложений включает основной цикл, который обертывает выполняемый код глобальным обработчиком исключений, перехватывающим те ошибки, что не были обработаны на более низком уровне. Можно получить тот же результат, установив в качестве перехватчика sys.excepthook функцию, получающую три аргумента (тип ошибки, значение ошибки и объект трассировки), и поручив ей обработку ошибок, оставшихся необработанными.

Листинг 17.30. sys_excepthook.py

```
import sys


def my_excepthook(type, value, traceback):
    print('Unhandled error:', type, value)


sys.excepthook = my_excepthook
```

[7] https://ru.wikipedia.org/wiki/Порядок_байтов

```
print('Before exception')

raise RuntimeError('This is the error message')

print('After exception')
```

Поскольку строка, в которой возникло исключение, не находится в блоке try:except, следующий за ней вызов функции print() не выполняется, хотя и установлен перехватчик исключений.

```
$ python3 sys_excepthook.py

Before exception
Unhandled error: <class 'RuntimeError'> This is the error
message
```

17.2.4.2. Текущее исключение

В ряде случаев использование явного обработчика исключений оказывается более предпочтительным, поскольку это позволяет либо сделать код более понятным, либо избежать конфликтов с библиотеками, пытающимися установить собственные перехватчики исключений. В подобных ситуациях программист может создать собственную функцию-обработчик, не нуждающуюся в явной передаче ей объекта исключения, путем вызова функции exc_info() для извлечения текущего исключения, возникшего в данном потоке.

Возвращаемым значением функции exc_info() является кортеж из трех элементов, содержащий класс исключения, экземпляр исключения и объект трассировки. Использование функции exc_info() более предпочтительно по сравнению с ранее использованной формой (получающей аргументы exc_type, exc_value и exc_traceback), поскольку это обеспечивает потоковую безопасность.

Листинг 17.31. sys_exc_info.py

```
import sys
import threading
import time


def do_something_with_exception():
    exc_type, exc_value = sys.exc_info()[:2]
    print('Handling {} exception with message "{}" in {}'.format(
        exc_type.__name__, exc_value,
        threading.current_thread().name))


def cause_exception(delay):
    time.sleep(delay)
    raise RuntimeError('This is the error message')


def thread_target(delay):
    try:
```

```
        cause_exception(delay)
    except:
        do_something_with_exception()


threads = [
    threading.Thread(target=thread_target, args=(0.3,)),
    threading.Thread(target=thread_target, args=(0.1,)),
]

for t in threads:
    t.start()
for t in threads:
    t.join()
```

В данном примере возможность образования циклической ссылки между объектом трассировки и локальной переменной в текущем фрейме предотвращается за счет игнорирования этой части значения, возвращаемого функцией `exc_info()`. Если требуется объект трассировки (например, для создания соответствующей записи в журнале), необходимо явно удалить локальную переменную (с помощью команды `del`), чтобы избежать образования циклических ссылок.

```
$ python3 sys_exc_info.py

Handling RuntimeError exception with message "This is the error
message" in Thread-2
Handling RuntimeError exception with message "This is the error
message" in Thread-1
```

17.2.4.3. Предыдущее интерактивное исключение

Интерактивный интерпретатор включает только один поток взаимодействия. Необработанные исключения в этом потоке сохраняются в трех переменных модуля `sys` (`last_type`, `last_value` и `last_traceback`), тем самым упрощая их извлечение для отладочных целей. Использование поставарийного отладчика модуля `pdb` (раздел 16.7) избавляет от необходимости непосредственно использовать эти значения.

```
$ python3

Python 3.4.2 (v3.4.2:ab2c023a9432, Oct 5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def cause_exception():
...     raise RuntimeError('This is the error message')
...
>>> cause_exception()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cause_exception
RuntimeError: This is the error message
>>> import pdb
```

```
>>> pdb.pm()
> <stdin>(2)cause_exception()
(Pdb) where
  <stdin>(1)<module>()
> <stdin>(2)cause_exception()
(Pdb)
```

Дополнительные ссылки

- `exceptions`. Встроенные ошибки.
- `pdb` (раздел 16.7). Отладчик Python.
- `traceback` (раздел 16.5). Модуль для работы с трассировочной информацией.

17.2.5. Низкоуровневая поддержка потоков

Модуль `sys` включает низкоуровневые функции, предназначенные для управления поведением потоков и их отладки.

17.2.5.1. Интервалы переключения потоков

Python 3 использует глобальную блокировку интерпретатора, позволяющую избежать повреждения состояния интерпретатора отдельными потоками. По прошествии определенного интервала времени, величину которого можно регулировать, выполнение байт-кода приостанавливается, и интерпретатор проверяет, необходимо ли выполнить какой-либо из обработчиков сигналов. Во время этой проверки *глобальная блокировка интерпретатора* (global interpreter lock — GIL) освобождается текущим потоком, а затем заново захватывается, причем другие потоки имеют приоритет по отношению к потоку, только что освободившему блокировку.

По умолчанию интервал переключения составляет 5 миллисекунд, и его текущее значение в любой момент можно получить с помощью функции `sys.getswitchinterval()`. Изменение величины интервала с помощью функции `sys.setswitchinterval()` может влиять на производительность приложения, причем степень этого влияния зависит от выполняемых операций.

Листинг 17.32. `sys_switchinterval.py`

```
import sys
import threading
from queue import Queue


def show_thread(q):
    for i in range(5):
        for j in range(1000000):
            pass
        q.put(threading.current_thread().name)
    return


def run_threads():
    interval = sys.getswitchinterval()
```

```
    print('interval = {:0.3f}'.format(interval))
    q = Queue()
    threads = [
        threading.Thread(target=show_thread,
                         name='T{}'.format(i),
                         args=(q,))
        for i in range(3)
    ]
    for t in threads:
        t.setDaemon(True)
        t.start()
    for t in threads:
        t.join()
    while not q.empty():
        print(q.get(), end=' ')
    print()
    return


for interval in [0.001, 0.1]:
    sys.setswitchinterval(interval)
    run_threads()
    print()
```

Если интервал переключения меньше промежутка времени, который требуется потоку для завершения выполнения, то интерпретатор передает управление другому потоку, который будет выполняться в течение некоторого времени. Это поведение иллюстрируется первым набором приведенных ниже результатов, в котором указанный интервал установлен равным 1 миллисекунде.

При более длительных интервалах переключения активный поток сможет выполнить больше работы, прежде чем будет вынужден уступить управление. Этот случай иллюстрируется измененным порядком следования имен потоков в очереди во втором наборе результатов, который был получен при интервале переключения, равном 10 миллисекундам.

```
$ python3 sys_switchinterval.py

interval = 0.001
T0 T1 T2 T1 T0 T2 T0 T1 T2 T1 T0 T2 T1 T0 T2

interval = 0.100
T0 T0 T0 T0 T0 T1 T1 T1 T1 T1 T2 T2 T2 T2 T2
```

Кроме величины указанного интервала на поведение переключения контекста потоков Python оказывают влияние другие факторы. Например, если поток выполняет операции ввода-вывода, то он освобождает GIL, тем самым предоставляя другому потоку возможность перехватить управление.

17.2.5.2. Отладка

Одним из наиболее трудных аспектов работы с потоками является предотвращение взаимоблокировок. Здесь на выручку может прийти функция `sys`.

_current_frames(), способная точно указать, где именно застопорилось выполнение потока.

Листинг 17.33. sys_current_frames.py

```
import sys
import threading
import time

io_lock = threading.Lock()
blocker = threading.Lock()


def block(i):
    t = threading.current_thread()
    with io_lock:
        print('{} with ident {} going to sleep'.format(
            t.name, t.ident))
    if i:
        blocker.acquire()  # блокировка приобретается, но
                           # никогда не освобождается
        time.sleep(0.2)
    with io_lock:
        print(t.name, 'finishing')
    return


# Создать и запустить несколько потоков, которые блокируются
threads = [
    threading.Thread(target=block, args=(i,))
    for i in range(3)
]
for t in threads:
    t.setDaemon(True)
    t.start()

# Создать для потоков отображение идентификаторов
# на объекты потоков
threads_by_ident = dict((t.ident, t) for t in threads)

# Показать, где именно блокируется каждый поток
time.sleep(0.01)
with io_lock:
    for ident, frame in sys._current_frames().items():
        t = threads_by_ident.get(ident)
        if not t:
            # Основной поток
            continue
        print('{} stopped in {} at line {} of {}'.format(
            t.name, frame.f_code.co_name,
            frame.f_lineno, frame.f_code.co_filename))
```

В словаре, возвращаемом функцией `sys._current_frames()`, ключами служат идентификаторы потоков, а не их имена. Для обратного преобразования идентификаторов в объекты потоков требуется выполнить небольшую работу.

Поскольку поток `Thread-1` не спит, он успевает завершить выполнение до проверки его состояния. В силу того, что этот поток уже не является активным, он не появляется в выводе. Поток `Thread-2` получает блокировку, а затем засыпает на короткое время. В это время поток `Thread-3` пытается получить блокировку, но не может этого сделать, поскольку ее уже захватил поток `Thread-2`.

```
$ python3 sys_current_frames.py

Thread-1 with ident 123145307557888 going to sleep
Thread-1 finishing
Thread-2 with ident 123145307557888 going to sleep
Thread-3 with ident 123145312813056 going to sleep
Thread-3 stopped in block at line 18 of sys_current_frames.py
Thread-2 stopped in block at line 19 of sys_current_frames.py
```

Дополнительные ссылки

- `threading` (раздел 10.3). Модуль `threading` включает классы, предназначенные для создания потоков Python.
- `Queue`. Модуль `Queue` предоставляет потокобезопасную реализацию структуры данных FIFO.
- *Reworking the GIL* (Antoine Pitrou)[8]. Письмо из списка рассылки **python-dev**, в котором описаны изменения в реализации GIL, связанные с введением интервалов переключения.

17.2.6. Модули и операции импорта

Большинство программ на языке Python состоит из нескольких модулей и импортирующего их основного приложения. Независимо от того, что именно используется — средства стандартной библиотеки или же пользовательский код, организованный в виде отдельных файлов для упрощения его сопровождения, — понимание зависимостей программы и управления ими является важным аспектом разработки. Модуль `sys` включает информацию о модулях, доступных приложению в качестве встроенных или импортируемых объектов. Он также определяет функции-перехватчики, позволяющие в случае необходимости переопределять стандартное поведение операции импорта.

17.2.6.1. Импорт модулей

`sys.modules` — это словарь, который связывает имена импортируемых объектов с объектами модулей, содержащими их код.

Листинг 17.34. `sys_modules.py`

```
import sys
import textwrap
```

[8] https://mail.python.org/pipermail/python-dev/2009-October/093321.html

```
names = sorted(sys.modules.keys())
name_text = ', '.join(names)

print(textwrap.fill(name_text, width=64))
```

По мере импортирования новых модулей содержимое `sys.modules` изменяется.

```
$ python3 sys_modules.py

__main__, _bootlocale, _codecs, _collections_abc,
_frozen_importlib, _frozen_importlib_external, _imp, _io,
_locale, _signal, _sre, _stat, _thread, _warnings, _weakref,
_weakrefset, abc, builtins, codecs, copyreg, encodings,
encodings.aliases, encodings.latin_1, encodings.utf_8, errno,
genericpath, io, marshal, os, os.path, posix, posixpath, re,
site, sre_compile, sre_constants, sre_parse, stat, sys,
textwrap, zipimport
```

17.2.6.2. Встроенные модули

Интерпретатор Python может компилироваться вместе со встроенными модулями на языке C, чтобы избежать необходимости распространять их в виде отдельных разделяемых библиотек. Эти модули не появляются в списке импортируемых модулей `sys.modules`, поскольку технически они не импортируются. Единственным способом поиска встроенных модулей является использование списка `sys.builtin_module_names`.

Листинг 17.35. `sys_builtins.py`

```
import sys
import textwrap

name_text = ', '.join(sorted(sys.builtin_module_names))

print(textwrap.fill(name_text, width=64))
```

Результаты работы этого сценария будут отличаться для разных систем, особенно в случае нестандартных сборок интерпретатора. Приведенные ниже результаты были получены с использованием экземпляра интерпретатора, инсталлированного с помощью стандартного установщика для OS X, который предлагается на сайте `python.org`.

```
$ python3 sys_builtins.py

_ast, _codecs, _collections, _functools, _imp, _io, _locale,
_operator, _signal, _sre, _stat, _string, _symtable, _thread,
_tracemalloc, _warnings, _weakref, atexit, builtins, errno,
faulthandler, gc, itertools, marshal, posix, pwd, sys, time,
xxsubtype, zipimport
```

Дополнительные ссылки

- *Build Instructions*[9]. Инструкции по сборке Python из файла README, распространяемого вместе с исходным кодом.

17.2.6.3. Пути импорта

Путем поиска модулей можно управлять как списком Python, сохраняемым в атрибуте `sys.path`. По умолчанию этот список включает каталог, в котором находится сценарий, использованный для запуска приложения, и текущий рабочий каталог.

Листинг 17.36. `sys_path_show.py`

```
import sys

for d in sys.path:
    print(d)
```

Первым из каталогов среди путей поиска является домашний каталог самого сценария примера. За ним следует ряд платформозависимых путей, указывающих, где могут находиться установленные модули расширения (написанные на языке C). Последним указан глобальный каталог `site-packages`.

```
$ python3 sys_path_show.py

/Users/dhellmann/Documents/PyMOTW/pymotw-3/source/sys
.../python35.zip
.../lib/python3.5
.../lib/python3.5/plat-darwin
.../python3.5/lib-dynload
.../lib/python3.5/site-packages
```

Список путей поиска импортируемых модулей можно изменить до запуска интерпретатора, задав в переменной `PYTHONPATH` командной оболочки каталоги, разделенные двоеточиями.

```
$ PYTHONPATH=/my/private/site-packages:/my/shared/site-packages \
> python3 sys_path_show.py

/Users/dhellmann/Documents/PyMOTW/pymotw-3/source/sys
/my/private/site-packages
/my/shared/site-packages
.../python35.zip
.../lib/python3.5
.../lib/python3.5/plat-darwin
.../python3.5/lib-dynload
.../lib/python3.5/site-packages
```

Кроме того, программа может изменить путь поиска, добавив элементы непосредственно в список `sys.path`.

[9] https://hg.python.org/cpython/file/tip/README

Листинг 17.37. `sys_path_modify.py`

```
import imp
import os
import sys

base_dir = os.path.dirname(__file__) or '.'
print('Base directory:', base_dir)

# Вставить каталог package_dir_a в начало списка
# путей поиска модулей
package_dir_a = os.path.join(base_dir, 'package_dir_a')
sys.path.insert(0, package_dir_a)

# Импортировать модуль примера
import example
print('Imported example from:', example.__file__)
print(' ', example.DATA)

# Поставить каталог package_dir_b первым в списке путей поиска
package_dir_b = os.path.join(base_dir, 'package_dir_b')
sys.path.insert(0, package_dir_b)

# Перезагрузить модуль для получения другой версии
imp.reload(example)
print('Reloaded example from:', example.__file__)
print(' ', example.DATA)
```

При перезагрузке импортированного модуля файл импортируется заново, а это означает, что изменение пути в промежуток времени между первоначальной операцией импорта и вызовом функции `reload()` позволяет загрузить одноименный модуль из другого каталога.

```
$ python3 sys_path_modify.py

Base directory: .
Imported example from: ./package_dir_a/example.py
    This is example A
Reloaded example from: ./package_dir_b/example.py
    This is example B
```

17.2.6.4. Нестандартный импорт

Изменяя пути импорта, программист может выбирать расположения, из которых будут загружаться стандартные модули Python. А что если программе необходимо импортировать код из какого-либо другого источника, отличного от обычных мест расположения *.py*- или *.pyc*-файлов в файловой системе? В документе **PEP 302**[10] эта проблема решается за счет введения функций, перехватывающих попытки найти модуль в каталогах путей поиска при его импорте и предпринимающих альтернативные действия для импортирования кода из других мест или его предварительной обработки.

[10] www.python.org/dev/peps/pep-0302

Пользовательский *импорт* реализуется в два этапа. *Искатель* (finder) отвечает за определение местонахождения модуля и предоставление *загрузчика* (loader) для управления фактической операцией импорта. Чтобы добавить искатель, следует присоединить фабрику объектов к списку `sys.path_hooks`. В процессе импорта искателю предоставляется каждая часть пути, пока одна из них не подтвердит наличие поддержки (за счет того, что не сгенерирует исключение `ImportError`). Этот искатель отвечает за дальнейший поиск хранилища данных, представленного собственным путем поиска модулей.

Листинг 17.38. `sys_path_hooks_noisy.py`

```
import sys


class NoisyImportFinder:

    PATH_TRIGGER = 'NoisyImportFinder_PATH_TRIGGER'

    def __init__(self, path_entry):
        print('Checking {}:'.format(path_entry), end=' ')
        if path_entry != self.PATH_TRIGGER:
            print('wrong finder')
            raise ImportError()
        else:
            print('works')
        return

    def find_module(self, fullname, path=None):
        print('Looking for {!r}'.format(fullname))
        return None


sys.path_hooks.append(NoisyImportFinder)

for hook in sys.path_hooks:
    print('Path hook: {}'.format(hook))

sys.path.insert(0, NoisyImportFinder.PATH_TRIGGER)

try:
    print('importing target_module')
    import target_module
except Exception as e:
    print('Import failed:', e)
```

Приведенный пример демонстрирует процесс инстанциализации и опроса искателей. Класс `NoisyImportFinder` возбуждает исключение `ImportError`, если он инстанциализируется с использованием пути, не совпадающего со специальным триггерным значением, которое явно не представляет реальный путь в файловой системе. Благодаря такой проверке класс `NoisyImportFinder` не создает помех для импорта реальных модулей.

```
$ python3 sys_path_hooks_noisy.py

Path hook: <class 'zipimport.zipimporter'>
Path hook: <function
FileFinder.path_hook.<locals>.path_hook_for_FileFinder at
0x100734950>
Path hook: <class '__main__.NoisyImportFinder'>
importing target_module
Checking NoisyImportFinder_PATH_TRIGGER: works
Looking for 'target_module'
Import failed: No module named 'target_module'
```

17.2.6.5. Импорт из хранилищ модуля `shelve`

Если искателю удается найти модуль, он отвечает за возврат *загрузчика*, способного импортировать данный модуль. Приведенный в этом разделе пример иллюстрирует создание пользовательского импортера, сохраняющего содержимое своего модуля в базе данных, созданной с помощью модуля `shelve` (см. раздел 7.2).

Сначала с помощью приведенного ниже сценария в хранилище сохраняется пакет, содержащий подмодуль и подпакет.

Листинг 17.39. `sys_shelve_importer_create.py`

```
import shelve
import os

filename = '/tmp/pymotw_import_example.shelve'
if os.path.exists(filename + '.db'):
    os.unlink(filename + '.db')
with shelve.open(filename) as db:
    db['data:README'] = b"""
==============
package README
==============

This is the README for "package".
"""
    db['package.__init__'] = b"""
print('package imported')
message = 'This message is in package.__init__'
"""
    db['package.module1'] = b"""
print('package.module1 imported')
message = 'This message is in package.module1'
"""
    db['package.subpackage.__init__'] = b"""
print('package.subpackage imported')
message = 'This message is in package.subpackage.__init__'
"""
    db['package.subpackage.module2'] = b"""
print('package.subpackage.module2 imported')
message = 'This message is in package.subpackage.module2'
```

```
"""
    db['package.with_error'] = b"""
print('package.with_error being imported')
raise ValueError('raising exception to break import')
"""
    print('Created {} with:'.format(filename))
    for key in sorted(db.keys()):
        print('  ', key)
```

В реальном сценарии создания пакета содержимое будет читаться из файловой системы, но в данном простом примере достаточно использовать жестко закодированные значения.

```
$ python3 sys_shelve_importer_create.py

Created /tmp/pymotw_import_example.shelve with:
   data:README
   package.__init__
   package.module1
   package.subpackage.__init__
   package.subpackage.module2
   package.with_error
```

Пользовательский импортер должен предоставить классы для искателя и загрузчика, которым известно, каким образом должен осуществляться поиск модуля или пакета в хранилище.

Листинг 17.40. sys_shelve_importer.py

```
import imp
import os
import shelve
import sys


def _mk_init_name(fullname):
    """Возвращает имя модуля __init__
    для заданного имени пакета.
    """
    if fullname.endswith('.__init__'):
        return fullname
    return fullname + '.__init__'


def _get_key_name(fullname, db):
    """Поискать в хранилище shelve имя fullname или
    fullname.__init__ и вернуть найденное имя.
    """
    if fullname in db:
        return fullname
    init_name = _mk_init_name(fullname)
    if init_name in db:
        return init_name
```

```
    return None


class ShelveFinder:
    """Найти модули, собранные в хранилище shelve."""

    _maybe_recursing = False

    def __init__(self, path_entry):
        # При загрузке хранилища shelve создается рекурсивный цикл
        # импорта, если импортируется dbm, и нам известно, что в
        # этом случае мы не будем загружать импортируемый модуль.
        # Таким образом, если обнаруживается, что возможна
        # рекурсия, следует просто игнорировать запрос и
        # использовать другой искатель.
        if ShelveFinder._maybe_recursing:
            raise ImportError
        try:
            # Проверить, что path_entry действительно является
            # хранилищем shelve
            try:
                ShelveFinder._maybe_recursing = True
                with shelve.open(path_entry, 'r'):
                    pass
            finally:
                ShelveFinder._maybe_recursing = False
        except Exception as e:
            print('shelf could not import from {}: {}'.format(
                path_entry, e))
            raise
        else:
            print('shelf added to import path:', path_entry)
            self.path_entry = path_entry
        return

    def __str__(self):
        return '<{} for {!r}>'.format(self.__class__.__name__,
                                      self.path_entry)

    def find_module(self, fullname, path=None):
        path = path or self.path_entry
        print('\nlooking for {!r}\n  in {}'.format(
            fullname, path))
        with shelve.open(self.path_entry, 'r') as db:
            key_name = _get_key_name(fullname, db)
            if key_name:
                print('  found it as {}'.format(key_name))
                return ShelveLoader(path)
        print('  not found')
        return None


class ShelveLoader:
```

```
    """Загрузить исходные коды модулей из хранилища shelve."""

    def __init__(self, path_entry):
        self.path_entry = path_entry
        return

    def _get_filename(self, fullname):
        # Создать фиктивное имя файла, начинающееся с данного
        # пути, чтобы обеспечить корректную работу
        # функции pkgutil.get_data()
        return os.path.join(self.path_entry, fullname)

    def get_source(self, fullname):
        print('loading source for {!r} from shelf'.format(
            fullname))
        try:
            with shelve.open(self.path_entry, 'r') as db:
                key_name = _get_key_name(fullname, db)
                if key_name:
                    return db[key_name]
                raise ImportError(
                    'could not find source for {}'.format(
                        fullname)
                )
        except Exception as e:
            print('could not load source:', e)
            raise ImportError(str(e))

    def get_code(self, fullname):
        source = self.get_source(fullname)
        print('compiling code for {!r}'.format(fullname))
        return compile(source, self._get_filename(fullname),
                       'exec', dont_inherit=True)

    def get_data(self, path):
        print('looking for data\n  in {}\n  for {!r}'.format(
            self.path_entry, path))
        if not path.startswith(self.path_entry):
            raise IOError
        path = path[len(self.path_entry) + 1:]
        key_name = 'data:' + path
        try:
            with shelve.open(self.path_entry, 'r') as db:
                return db[key_name]
        except Exception:
            # Преобразовывать все ошибки в тип IOError
            raise IOError()

    def is_package(self, fullname):
        init_name = _mk_init_name(fullname)
        with shelve.open(self.path_entry, 'r') as db:
            return init_name in db
```

```
    def load_module(self, fullname):
        source = self.get_source(fullname)

        if fullname in sys.modules:
            print('reusing module from import of {!r}'.format(
                fullname))
            mod = sys.modules[fullname]
        else:
            print('creating a new module object for {!r}'.format(
                fullname))
            mod = sys.modules.setdefault(
                fullname,
                imp.new_module(fullname)
            )

        # Задать свойства, требуемые документом PEP 302
        mod.__file__ = self._get_filename(fullname)
        mod.__name__ = fullname
        mod.__path__ = self.path_entry
        mod.__loader__ = self
        # PEP-366 определяет, что пакеты устанавливают в
        # качестве значения атрибута __package__ свое имя,
        # а для модулей им является имя их родительского
        # пакета (если таковой имеется)
        if self.is_package(fullname):
            mod.__package__ = fullname
        else:
            mod.__package__ = '.'.join(fullname.split('.')[:-1])

        if self.is_package(fullname):
            print('adding path for package')
            # Задать атрибут __path__ для пакетов, чтобы
            # можно было найти подмодули
            mod.__path__ = [self.path_entry]
        else:
            print('imported as regular module')

        print('execing source...')
        exec(source, mod.__dict__)
        print('done')
        return mod
```

Теперь классы `ShelveFinder` и `ShelveLoader` можно использовать для импортирования кода из хранилища. Например, созданный пакет `package` можно импортировать с помощью следующего кода.

Листинг 17.41. `sys_shelve_importer_package.py`

```
import sys
import sys_shelve_importer


def show_module_details(module):
    print('  message    :', module.message)
```

```
    print('  __name__    :', module.__name__)
    print('  __package__:', module.__package__)
    print('  __file__    :', module.__file__)
    print('  __path__    :', module.__path__)
    print('  __loader__  :', module.__loader__)


filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print('Import of "package":')
import package

print()
print('Examine package details:')
show_module_details(package)

print()
print('Global settings:')
print('sys.modules entry:')
print(sys.modules['package'])
```

Хранилище добавляется в путь импорта при первой же операции импорта, осуществляемой после изменения пути. Искатель распознает хранилище и возвращает загрузчик, который используется для всех операций импорта из данного хранилища. Первоначальное импортирование на уровне пакета создает новый объект модуля, а затем использует функцию exec для выполнения исходного кода, загруженного из хранилища. Новый модуль используется в качестве пространства имен, поэтому имена, определенные в исходном коде, сохраняются в виде атрибутов на уровне модуля.

```
$ python3 sys_shelve_importer_package.py

Import of "package":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done

Examine package details:
  message     : This message is in package.__init__
  __name__    : package
  __package__: package
  __file__    : /tmp/pymotw_import_example.shelve/package
```

```
  __path__    : ['/tmp/pymotw_import_example.shelve']
  __loader__  : <sys_shelve_importer.ShelveLoader object at
0x101467860>

Global settings:
sys.modules entry:
<module 'package' (<sys_shelve_importer.ShelveLoader object at
0x101467860>)>
```

17.2.6.6. Импорт пользовательских пакетов

Загрузка других модулей и подпакетов осуществляется аналогичным образом.

Листинг 17.42. sys_shelve_importer_module.py

```
import sys
import sys_shelve_importer


def show_module_details(module):
    print('  message    :', module.message)
    print('  __name__   :', module.__name__)
    print('  __package__:', module.__package__)
    print('  __file__   :', module.__file__)
    print('  __path__   :', module.__path__)
    print('  __loader__ :', module.__loader__)


filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print('Import of "package.module1":')
import package.module1

print()
print('Examine package.module1 details:')
show_module_details(package.module1)

print()
print('Import of "package.subpackage.module2":')
import package.subpackage.module2

print()
print('Examine package.subpackage.module2 details:')
show_module_details(package.subpackage.module2)
```

Искатель получает уточненное с помощью точечной нотации имя загружаемого модуля и возвращает объект `ShelveLoader`, настроенный для загрузки модулей в соответствии с путем, ведущим к файлу хранилища. Полностью уточненное имя модуля передается методу `load_module()` загрузчика, который конструирует и возвращает экземпляр модуля.

```
$ python3 sys_shelve_importer_module.py

Import of "package.module1":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done

looking for 'package.module1'
  in /tmp/pymotw_import_example.shelve
  found it as package.module1
loading source for 'package.module1' from shelf
creating a new module object for 'package.module1'
imported as regular module
execing source...
package.module1 imported
done

Examine package.module1 details:
  message : This message is in package.module1
  __name__ : package.module1
  __package__: package
  __file__ : /tmp/pymotw_import_example.shelve/package.module1
  __path__ : /tmp/pymotw_import_example.shelve
  __loader__ : <sys_shelve_importer.ShelveLoader object at
0x101376e10>

Import of "package.subpackage.module2":

looking for 'package.subpackage'
  in /tmp/pymotw_import_example.shelve
  found it as package.subpackage.__init__
loading source for 'package.subpackage' from shelf
creating a new module object for 'package.subpackage'
adding path for package
execing source...
package.subpackage imported
done

looking for 'package.subpackage.module2'
  in /tmp/pymotw_import_example.shelve
  found it as package.subpackage.module2
loading source for 'package.subpackage.module2' from shelf
creating a new module object for 'package.subpackage.module2'
imported as regular module
execing source...
```

```
package.subpackage.module2 imported
done

Examine package.subpackage.module2 details:
  message : This message is in package.subpackage.module2
  __name__ : package.subpackage.module2
  __package__: package.subpackage
  __file__ :
/tmp/pymotw_import_example.shelve/package.subpackage.module2
  __path__ : /tmp/pymotw_import_example.shelve
  __loader__ : <sys_shelve_importer.ShelveLoader object at
0x1013a6c88>
```

17.2.6.7. Перезагрузка модулей с помощью пользовательского импортера

Перезагрузка модулей осуществляется несколько иначе. Вместо того чтобы создавать новый объект модуля, повторно используется существующий объект.

Листинг 17.43. sys_shelve_importer_reload.py

```
import importlib
import sys
import sys_shelve_importer

filename = '/tmp/pymotw_import_example.shelve'

sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print('First import of "package":')
import package

print()
print('Reloading "package":')
importlib.reload(package)
```

Благодаря повторному использованию того же объекта существующие ссылки на модуль сохраняются, даже если определения класса или функции изменяются на стадии загрузки.

```
$ python3 sys_shelve_importer_reload.py

First import of "package":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
```

```
done

Reloading "package":

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
reusing module from import of 'package'
adding path for package
execing source...
package imported
done
```

17.2.6.8. Обработка ошибок импорта

Если ни одному из искателей не удается найти модуль, основной импортирующий код возбуждает исключение `ImportError`.

Листинг 17.44. sys_shelve_importer_missing.py

```
import sys
import sys_shelve_importer

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

try:
    import package.module3
except ImportError as e:
    print('Failed to import:', e)
```

Другим ошибкам, возникающим в процессе импорта, разрешено распространяться.

```
$ python3 sys_shelve_importer_missing.py

shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done

looking for 'package.module3'
  in /tmp/pymotw_import_example.shelve
  not found
Failed to import: No module named 'package.module3'
```

17.2.6.9. Данные пакета

Кроме определения API для загрузки исполняемого кода Python документ **PEP 302** определяет дополнительный API, предназначенный для извлечения данных пакета, которые могут использоваться при распространении файлов данных, документации и других необходимых пакету ресурсов, не являющихся кодом. Реализуя метод `get_data()`, загрузчик может предоставить вызывающим приложениям возможность поддерживать извлечение данных, ассоциированных с пакетом, без учета того, каким образом фактически устанавливается пакет (в частности, не предполагая, что пакет сохраняется в виде файлов в файловой системе).

Листинг 17.45. `sys_shelve_importer_get_data.py`

```
import sys
import sys_shelve_importer
import os
import pkgutil

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

import package

readme_path = os.path.join(package.__path__[0], 'README')

readme = pkgutil.get_data('package', 'README')
# Эквивалентно следующей инструкции:
# readme = package.__loader__.get_data(readme_path)
print(readme.decode('utf-8'))

foo_path = os.path.join(package.__path__[0], 'foo')
try:
    foo = pkgutil.get_data('package', 'foo')
    # Эквивалентно следующей инструкции:
    # foo = package.__loader__.get_data(foo_path)
except IOError as err:
    print('ERROR: Could not load "foo"', err)
else:
    print(foo)
```

Функция `get_data()` принимает в качестве аргумента путь, базирующийся на модуле или пакете, которому принадлежат данные. Она либо возвращает содержимое ресурсного файла в виде байтовой строки, либо возбуждает исключение `IOError`, если такого ресурса не существует.

```
$ python3 sys_shelve_importer_get_data.py

shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
```

```
creating a new module object for 'package'
adding path for package
execing source...
package imported
done
looking for data
  in /tmp/pymotw_import_example.shelve
  for '/tmp/pymotw_import_example.shelve/README'

==============
package README
==============

This is the README for "package".

looking for data
  in /tmp/pymotw_import_example.shelve
  for '/tmp/pymotw_import_example.shelve/foo'
ERROR: Could not load "foo"
```

Дополнительные ссылки

- `pkgutil` (раздел 19.2). Включает функцию `get_data()`, предназначенную для извлечения данных из пакета.

17.2.6.10. Кеш объекта импорта

Выполнение процедуры поиска по всем перехватчикам всякий раз, когда импортируется модуль, может занимать много времени. В целях экономии времени поддерживается словарь `sys.path_importer_cache`, устанавливающий соответствие между значениями пути и именами загрузчиков, которые могут использовать эти значения для поиска модулей.

Листинг 17.46. `sys_path_importer_cache.py`

```
import os
import sys

prefix = os.path.abspath(sys.prefix)

print('PATH:')
for name in sys.path:
    name = name.replace(prefix, '...')
    print(' ', name)

print()
print('IMPORTERS:')
for name, cache_value in sys.path_importer_cache.items():
    if '..' in name:
        name = os.path.abspath(name)
    name = name.replace(prefix, '...')
    print('  {}: {!r}'.format(name, cache_value))
```

Для идентификации путей к каталогам файловой системы используются искатели (класс FileFinder). Если каталог не поддерживается никаким искателем, то ему соответствует значение None, поскольку его нельзя использовать для импортирования модулей. В приведенном ниже выводе, отображающем импортирование из кеша, результаты усечены форматированием.

```
$ python3 sys_path_importer_cache.py

PATH:
  /Users/dhellmann/Documents/PyMOTW/Python3/pymotw-3/source/sys
  .../lib/python35.zip
  .../lib/python3.5
  .../lib/python3.5/plat-darwin
  .../lib/python3.5/lib-dynload
  .../lib/python3.5/site-packages
IMPORTERS:
  sys_path_importer_cache.py: None
  .../lib/python3.5/encodings: FileFinder(
  '.../lib/python3.5/encodings')
  .../lib/python3.5/lib-dynload: FileFinder(
  '.../lib/python3.5/lib-dynload')
  .../lib/python3.5/lib-dynload: FileFinder(
  '.../lib/python3.5/lib-dynload')
  .../lib/python3.5/site-packages: FileFinder(
  '.../lib/python3.5/site-packages')
  .../lib/python3.5: FileFinder(
  '.../lib/python3.5/')
  .../lib/python3.5/plat-darwin: FileFinder(
  '.../lib/python3.5/plat-darwin')
  .../lib/python3.5: FileFinder(
  '.../lib/python3.5')
  .../lib/python35.zip: None
  .../lib/python3.5/plat-darwin: FileFinder(
  '.../lib/python3.5/plat-darwin')
```

17.2.6.11. Метапуть

Список sys.meta_path дополнительно расширяет состав потенциальных источников импорта, поскольку искатель просматривает его *до* просмотра обычного списка путей sys.path. API искателя для метапути тот же, что и для обычного пути, за исключением того, что метаискатель не ограничивается одной записью в sys.path — он может выполнять поиск везде.

Листинг 17.47. sys_meta_path.py

```
import sys
import imp


class NoisyMetaImportFinder:

    def __init__(self, prefix):
```

```
        print('Creating NoisyMetaImportFinder for {}'.format(
            prefix))
        self.prefix = prefix
        return

    def find_module(self, fullname, path=None):
        print('looking for {!r} with path {!r}'.format(
            fullname, path))
        name_parts = fullname.split('.')
        if name_parts and name_parts[0] == self.prefix:
            print(' ... found prefix, returning loader')
            return NoisyMetaImportLoader(path)
        else:
            print(' ... not the right prefix, cannot load')
        return None


class NoisyMetaImportLoader:

    def __init__(self, path_entry):
        self.path_entry = path_entry
        return

    def load_module(self, fullname):
        print('loading {}'.format(fullname))
        if fullname in sys.modules:
            mod = sys.modules[fullname]
        else:
            mod = sys.modules.setdefault(
                fullname,
                imp.new_module(fullname))

        # Задать свойства, требуемые документом PEP 302
        mod.__file__ = fullname
        mod.__name__ = fullname
        # Всегда выглядит как пакет
        mod.__path__ = ['path-entry-goes-here']
        mod.__loader__ = self
        mod.__package__ = '.'.join(fullname.split('.')[:-1])

        return mod


# Установить искатель для метапути
sys.meta_path.append(NoisyMetaImportFinder('foo'))

# Импортировать модули, найденные искателем для метапути
print()
import foo

print()
import foo.bar
```

```
# Импортировать модуль, который не был найден
print()
try:
    import bar
except ImportError as e:
    pass
```

Каждый искатель метапути опрашивается до выполнения поиска в списке `sys.path`, поэтому центральный импортер всегда может загрузить модули без явного изменения списка `sys.path`. Как только модуль найден, API загрузчика работает точно так же, как и в случае обычных загрузчиков (хотя вывод этого примера для простоты усечен).

```
$ python3 sys_meta_path.py

Creating NoisyMetaImportFinder for foo

looking for 'foo' with path None
 ... found prefix, returning loader
loading foo

looking for 'foo.bar' with path ['path-entry-goes-here']
 ... found prefix, returning loader
loading foo.bar

looking for 'bar' with path None
 ... not the right prefix, cannot load
```

Дополнительные ссылки

- `importlib` (раздел 19.1). Базовые классы и другие инструменты, предназначенные для создания пользовательских импортеров.
- `zipimport` (раздел 19.3). Реализует импортирование модулей Python из ZIP-архивов.
- *The Internal Structure of Python Eggs*[11]. Документация пакета `setuptools` в части, касающейся формата `egg`.
- `Wheel`[12]. Документация формата архивирования `wheel` для устанавливаемого кода Python.
- **PEP 302**[13]. *New Import Hooks*.
- **PEP 366**[14]. *Main module explicit relative imports*.
- **PEP 427**[15]. *The Wheel Binary Package Format 1.0*.
- *Import this, that, and the other thing: custom importers* (Brett Cannon)[16]. Презентация на конференции PyCon 2010.

[11] http://setuptools.readthedocs.io/en/latest/formats.html?highlight=egg
[12] http://wheel.readthedocs.org/en/latest/
[13] www.python.org/dev/peps/pep-0302
[14] www.python.org/dev/peps/pep-0366
[15] www.python.org/dev/peps/pep-0427
[16] http://pyvideo.org/pycon-us-2010/pycon-2010--import-this--that--and-the-other-thin.html

17.2.7. Трассировка выполняющихся программ

Существуют два способа внедрения кода, отслеживающего выполнение программы: *трассировка* и *профилирование*. Эти техники сходны, но предназначены для разных целей и поэтому имеют разные ограничения. Однако наиболее простым, хотя и наименее эффективным, является мониторинг выполнения программы с помощью перехватчика трассировки, который можно использовать для написания отладчика, мониторинга покрытия кода и архивирования сведений о многих других событиях.

Перехватчик трассировки можно видоизменять, передавая функцию обратного вызова в качестве аргумента функции `sys.settrace()`. Функция обратного вызова принимает три аргумента: фрейм текущего стека, строку с именем типа уведомления о событии и аргумент, зависящий от типа события. В табл. 17.2 приведены семь типов событий, свойственных процессу выполнения программы.

Таблица 17.2. Перехватчики событий для функции `settrace()`

Событие	Когда генерируется	Значение аргумента
`call`	Перед выполнением строки	`None`
`line`	Перед выполнением строки	`None`
`return`	Перед возвратом из функции	Возвращаемое значение
`exception`	После возникновения исключения	Кортеж (`exception`, `value`, `traceback`)
`c_call`	Перед вызовом функции C	Объект функции C
`c_return`	После возврата из функции C	`None`
`c_exception`	После возбуждения ошибки в функции C	`None`

17.2.7.1. Трассировка вызовов функций

Событие `call` генерируется при каждом вызове функции. Переданный функции обратного вызова фрейм стека позволяет определить, какая функция вызывается и откуда именно.

Листинг 17.48. `sys_settrace_call.py`

```
#!/usr/bin/env python3
# encoding: utf-8

import sys


def trace_calls(frame, event, arg):
    if event != 'call':
        return
    co = frame.f_code
    func_name = co.co_name
    if func_name == 'write':
        # Не выводить вызовы write()
        return
    func_line_no = frame.f_lineno
    func_filename = co.co_filename
```

```
    caller = frame.f_back
    caller_line_no = caller.f_lineno
    caller_filename = caller.f_code.co_filename
    print('* Call to', func_name)
    print('* on line {} of {}'.format(
        func_line_no, func_filename))
    print('* from line {} of {}'.format(
        caller_line_no, caller_filename))
    return


def b():
    print('inside b()\n')


def a():
    print('inside a()\n')
    b()

sys.settrace(trace_calls)
    a()
```

Вызовы функции `write()`, используемые для записи в стандартный поток `sys.stdout` при выводе информации на печать, в этом примере игнорируются.

```
$ python3 sys_settrace_call.py

* Call to a
* on line 32 of sys_settrace_call.py
* from line 37 of sys_settrace_call.py
inside a()

* Call to b
* on line 28 of sys_settrace_call.py
* from line 34 of sys_settrace_call.py
inside b()
```

17.2.7.2. Трассировка внутри функций

Перехватчик трассировки может вернуть другой перехватчик, предназначенный для использования в другой области видимости (*локальную* функцию трассировки). Например, это позволяет организовать построчную трассировку, ограниченную определенной функцией или модулем.

Листинг 17.49. `sys_settrace_line.py`

```
#!/usr/bin/env python3
# encoding: utf-8

import functools
import sys

```

```
 8 def trace_lines(frame, event, arg):
 9     if event != 'line':
10         return
11     co = frame.f_code
12     func_name = co.co_name
13     line_no = frame.f_lineno
14     print('*  {} line {}'.format(func_name, line_no))
15
16
17 def trace_calls(frame, event, arg, to_be_traced):
18     if event != 'call':
19         return
20     co = frame.f_code
21     func_name = co.co_name
22     if func_name == 'write':
23         # Игнорировать вызовы write() при выводе на экран
24         return
25     line_no = frame.f_lineno
26     filename = co.co_filename
27     print('* Call to {} on line {} of {}'.format(
28         func_name, line_no, filename))
29     if func_name in to_be_traced:
30         # Выполнять трассировку внутри этой функции
31         return trace_lines
32     return
33
34
35 def c(input):
36     print('input =', input)
37     print('Leaving c()')
38
39
40 def b(arg):
41     val = arg * 5
42     c(val)
43     print('Leaving b()')
44
45
46 def a():
47     b(2)
48     print('Leaving a()')
49
50
51 tracer = functools.partial(trace_calls, to_be_traced=['b'])
52 sys.settrace(tracer)
53 a()
```

В этом примере список функций, подлежащих трассировке, содержится в переменной `to_be_traced`. Таким образом, когда выполняется функция `race_calls()`, она может вернуть функцию `trace_lines()`, активизирующую трассировку внутри функции `b()`.

```
$ python3 sys_settrace_line.py

* Call to a on line 46 of sys_settrace_line.py
* Call to b on line 40 of sys_settrace_line.py
*  b line 41
*  b line 42
* Call to c on line 35 of sys_settrace_line.py
input = 10
Leaving c()
*  b line 43
Leaving b()
Leaving a()
```

17.2.7.3. Наблюдение за стеком

Другим полезным применением перехватчиков является отслеживание вызываемых функций и возвращаемых ими значений. Для мониторинга возвращаемых значений предназначено событие `return`.

Листинг 17.50. `sys_settrace_return.py`

```
#!/usr/bin/env python3
# encoding: utf-8

import sys


def trace_calls_and_returns(frame, event, arg):
    co = frame.f_code
    func_name = co.co_name
    if func_name == 'write':
        # Игнорировать вызовы write() при выводе на экран
        return
    line_no = frame.f_lineno
    filename = co.co_filename
    if event == 'call':
        print('* Call to {} on line {} of {}'.format(
            func_name, line_no, filename))
        return trace_calls_and_returns
    elif event == 'return':
        print('* {} => {}'.format(func_name, arg))
    return


def b():
    print('inside b()')
    return 'response_from_b '


def a():
    print('inside a()')
    val = b()
    return val * 2
```

```
33
34
35 sys.settrace(trace_calls_and_returns)
36 a()
```

Для мониторинга событий возврата из функций используется локальная функция трассировки. Чтобы отслеживание возвращаемого значения было возможным, функция `trace_calls_and_returns()` должна возвращать ссылку на саму себя, если обрабатывается событие `call`.

```
$ python3 sys_settrace_return.py

* Call to a on line 29 of sys_settrace_return.py
inside a()
* Call to b on line 24 of sys_settrace_return.py
inside b()
* b => response_from_b
* a => response_from_b response_from_b
```

17.2.7.4. Распространение исключений

Для мониторинга исключений необходимо отслеживать события `exception` в локальной функции трассировки. Когда возникает исключение, вызывается перехватчик трассировки с кортежем, содержащим тип исключения, объект исключения и объект трассировки.

Листинг 17.51. `sys_settrace_exception.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4 import sys
 5
 6
 7 def trace_exceptions(frame, event, arg):
 8     if event != 'exception':
 9         return
10     co = frame.f_code
11     func_name = co.co_name
12     line_no = frame.f_lineno
13     exc_type, exc_value, exc_traceback = arg
14     print(('* Tracing exception:\n'
15            '* {} "{}"\n'
16            '* on line {} of {}\n').format(
17                exc_type.__name__, exc_value, line_no,
18                func_name))
19
20
21 def trace_calls(frame, event, arg):
22     if event != 'call':
23         return
24     co = frame.f_code
25     func_name = co.co_name
```

```
26      if func_name in TRACE_INTO:
27          return trace_exceptions
28
29
30  def c():
31      raise RuntimeError('generating exception in c()')
32
33
34  def b():
35      c()
36      print('Leaving b()')
37
38
39  def a():
40      b()
41      print('Leaving a()')
42
43
44  TRACE_INTO = ['a', 'b', 'c']
45
46  sys.settrace(trace_calls)
47  try:
48      a()
49  except Exception as e:
50      print('Exception handler:', e)
```

Необходимо позаботиться о наложении ограничений на применение локальной функции, поскольку некоторые внутренние механизмы форматирования сообщений об ошибках генерируют — и игнорируют — собственные исключения. Перехватчик трассировки реагирует на каждое исключение, даже если вызывающий код перехватывает и игнорирует его.

```
$ python3 sys_settrace_exception.py

* Tracing exception:
* RuntimeError "generating exception in c()"
* on line 31 of c

* Tracing exception:
* RuntimeError "generating exception in c()"
* on line 35 of b

* Tracing exception:
* RuntimeError "generating exception in c()"
* on line 40 of a

Exception handler: generating exception in c()
```

Дополнительные ссылки

- `profile` (раздел 16.8). В документации модуля `profile` показано, как использовать готовый профилировщик.

- `trace` (раздел 16.4). Модуль `trace` реализует несколько средств, предназначенных для анализа кода.
- *Types and members*[17]. Описание объектов `frame` и `code` и их атрибутов.
- *Tracing Python code*[18]. Дополнительное руководство по использованию функции `settrace()`.
- *Wicked hack: Python bytecode tracing* (Ned Batchelder)[19]. Статья, содержащая описание экспериментов по трассировке кода с использованием более мелкой гранулярности, чем уровень строк исходного кода.
- `smiley`[20]. Средство трассировки приложений Python.

Дополнительные ссылки, касающиеся модуля `sys`

- Раздел документации стандартной библиотеки, посвященный модулю `sys`[21].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `sys` (раздел A.6.44).

17.3. os: портируемый доступ к средствам, специфическим для операционных систем

Модуль `os` предоставляет обертки для таких платформозависимых модулей, как `posix`, `nt` и `mac`. API для работы с функциями, доступными на всех платформах, должен быть одним и тем же, поэтому модуль `os` обеспечивает определенную портируемость программ. Однако не все функции доступны на всех платформах. Следует подчеркнуть, что многие из функций управления процессами, описанные в этой книге, недоступны на платформах Windows.

В документации Python описанию модуля `os` посвящен подраздел "Miscellaneous Operating System Interfaces". Данный модуль состоит главным образом из функций, предназначенных для управления процессами и содержимым файловой системы (файлами и каталогами), и сравнительно небольшого количества функций, обеспечивающих другие виды функциональности.

17.3.1. Исследование содержимого файловой системы

Чтобы подготовить список содержимого каталога файловой системы, следует использовать функцию `listdir()`.

Листинг 17.52. `os_listdir.py`

```
import os
import sys

print(os.listdir(sys.argv[1]))
```

[17] https://docs.python.org/3/library/inspect.html#types-and-members

[18] www.dalkescientific.com/writings/diary/archive/2005/04/20/tracing_python_code.html

[19] http://nedbatchelder.com/blog/200804/wicked_hack_python_bytecode_tracing.html

[20] https://pypi.python.org/pypi/smiley

[21] https://docs.python.org/3.5/library/sys.html

Функция `listdir()` возвращает список всех именованных элементов каталога, предоставленного в качестве аргумента. Никаких различий между файлами, подкаталогами и символическими ссылками не проводится.

```
$ python3 os_listdir.py .

['index.rst', 'os_access.py', 'os_cwd_example.py',
'os_directories.py', 'os_environ_example.py',
'os_exec_example.py', 'os_fork_example.py',
'os_kill_example.py', 'os_listdir.py', 'os_listdir.py~',
'os_process_id_example.py', 'os_process_user_example.py',
'os_rename_replace.py', 'os_rename_replace.py~',
'os_scandir.py', 'os_scandir.py~', 'os_spawn_example.py',
'os_stat.py', 'os_stat_chmod.py', 'os_stat_chmod_example.txt',
'os_strerror.py', 'os_strerror.py~', 'os_symlinks.py',
'os_system_background.py', 'os_system_example.py',
'os_system_shell.py', 'os_wait_example.py',
'os_waitpid_example.py', 'os_walk.py']
```

Функция `walk()` совершает рекурсивный обход каталога. Она генерирует для каждого подкаталога кортеж, содержащий путь к каталогу, все промежуточные каталоги, находящиеся вдоль этого пути, и список, включающий имена всех файлов в данном каталоге.

Листинг 17.53. `os_walk.py`

```
import os
import sys

# Если не предоставлен путь к списку, использовать /tmp
if len(sys.argv) == 1:
    root = '/tmp'
else:
    root = sys.argv[1]

for dir_name, sub_dirs, files in os.walk(root):
    print(dir_name)
    # Заканчивать имена каталогов символом /
    sub_dirs = [n + '/' for n in sub_dirs]
    # Создать общий список подкаталогов и файлов
    contents = sub_dirs + files
    contents.sort()
    # Отобразить содержимое
    for c in contents:
        print('  {}'.format(c))
    print()
```

В этом примере отображается следующий рекурсивный список каталогов.

```
$ python3 os_walk.py ../zipimport

../zipimport
  __init__.py
```

```
  example_package/
  index.rst
  zipimport_example.zip
  zipimport_find_module.py
  zipimport_get_code.py
  zipimport_get_data.py
  zipimport_get_data_nozip.py
  zipimport_get_data_zip.py
  zipimport_get_source.py
  zipimport_is_package.py
  zipimport_load_module.py
  zipimport_make_example.py

../zipimport/example_package
  README.txt
  __init__.py
```

В тех случаях, когда требуется более подробная информация, а не просто имена файлов, функция `scandir()` работает эффективнее, чем `listdir()`: при сканировании каталога один системный вызов позволяет получить больше информации.

Листинг 17.54. `os_scandir.py`

```
import os
import sys

for entry in os.scandir(sys.argv[1]):
    if entry.is_dir():
        typ = 'dir'
    elif entry.is_file():
        typ = 'file'
    elif entry.is_symlink():
        typ = 'link'
    else:
        typ = 'unknown'
    print('{name} {typ}'.format(
        name=entry.name,
        typ=typ,
    ))
```

Функция `scandir()` возвращает последовательность экземпляров `DirEntry` для элементов каталога. Этот объект имеет несколько атрибутов и методов, обеспечивающих доступ к метаданным файла.

```
$ python3 os_scandir.py .

index.rst file
os_access.py file
os_cwd_example.py file
os_directories.py file
os_environ_example.py file
os_exec_example.py file
```

```
os_fork_example.py file
os_kill_example.py file
os_listdir.py file
os_listdir.py~ file
os_process_id_example.py file
os_process_user_example.py file
os_rename_replace.py file
os_rename_replace.py~ file
os_scandir.py file
os_scandir.py~ file
os_spawn_example.py file
os_stat.py file
os_stat_chmod.py file
os_stat_chmod_example.txt file
os_strerror.py file
os_strerror.py~ file
os_symlinks.py file
os_system_background.py file
os_system_example.py file
os_system_shell.py file
os_wait_example.py file
os_waitpid_example.py file
os_walk.py file
```

17.3.2. Управление правами доступа к файловой системе

Более подробную информацию о файле можно получить с помощью функций `stat()` или `lstat()` (аналогична функции `stat()`, но не следует по символическим ссылкам).

Листинг 17.55. `os_stat.py`

```
import os
import sys
import time

if len(sys.argv) == 1:
    filename = __file__
else:
    filename = sys.argv[1]

stat_info = os.stat(filename)

print('os.stat({}):'.format(filename))
print('  Size:', stat_info.st_size)
print('  Permissions:', oct(stat_info.st_mode))
print('  Owner:', stat_info.st_uid)
print('  Device:', stat_info.st_dev)
print('  Created      :', time.ctime(stat_info.st_ctime))
print('  Last modified:', time.ctime(stat_info.st_mtime))
print('  Last accessed:', time.ctime(stat_info.st_atime))
```

Вывод может быть различным в зависимости от способа установки примера. Поэкспериментируйте с этой функцией, передавая различные имена файлов сценарию `os_stat.py`.

```
$ python3 os_stat.py

os.stat(os_stat.py):
  Size: 593
  Permissions: 0o100644
  Owner: 527
  Device: 16777218
  Created : Sat Dec 17 12:09:51 2016
  Last modified: Sat Dec 17 12:09:51 2016
  Last accessed: Sat Dec 31 12:33:19 2016

$ python3 os_stat.py index.rst

os.stat(index.rst):
  Size: 26878
  Permissions: 0o100644
  Owner: 527
  Device: 16777218
  Created : Sat Dec 31 12:33:10 2016
  Last modified: Sat Dec 31 12:33:10 2016
  Last accessed: Sat Dec 31 12:33:19 2016
```

В случае Unix-подобных систем права доступа можно изменять с помощью функции `chmod()`, передавая ей режим доступа в виде целого числа. Значения режима доступа можно формировать с помощью констант, определенных в модуле `stat`. В следующем примере переключается бит разрешения на выполнение файла.

Листинг 17.56. `os_stat_chmod.py`

```
import os
import stat

filename = 'os_stat_chmod_example.txt'
if os.path.exists(filename):
    os.unlink(filename)
with open(filename, 'wt') as f:
    f.write('contents')

# Определить, какие разрешения уже заданы, используя модуль stat
existing_permissions = stat.S_IMODE(os.stat(filename).st_mode)

if not os.access(filename, os.X_OK):
    print('Adding execute permission')
    new_permissions = existing_permissions | stat.S_IXUSR
else:
    print('Removing execute permission')
    # Использовать операцию XOR для отмены разрешения на выполнение
    new_permissions = existing_permissions ^ stat.S_IXUSR

os.chmod(filename, new_permissions)
```

В этом сценарии предполагается, что во время его выполнения он имеет полномочия, необходимые для изменения режима доступа к файлу.

```
$ python3 os_stat_chmod.py

Adding execute permission
```

Для проверки прав доступа процесса к файлу следует использовать функцию `access()`.

Листинг 17.57. `os_access.py`

```
import os

print('Testing:', __file__)
print('Exists:', os.access(__file__, os.F_OK))
print('Readable:', os.access(__file__, os.R_OK))
print('Writable:', os.access(__file__, os.W_OK))
print('Executable:', os.access(__file__, os.X_OK))
```

Результаты будут меняться в зависимости от способа установки кода примера, но в целом вывод будет аналогичен приведенному ниже.

```
$ python3 os_access.py

Testing: os_access.py
Exists: True
Readable: True
Writable: True
Executable: False
```

Библиотечная документация для функции `access()` содержит два предупреждения. Во-первых, нецелесообразно вызывать функцию `access()` для того, чтобы проверить возможность открытия файла до того, как фактически вызывать функцию `open()` для этого файла. Между этими двумя вызовами существует хотя и небольшой, но вполне ощутимый временной промежуток, в течение которого права доступа к файлу могут быть изменены. Второе предупреждение относится главным образом к сетевым файловым системам, расширяющим семантику разрешений POSIX. Некоторые файловые системы могут сначала ответить на вызов POSIX подтверждением полномочий процесса на доступ к файлу, а затем вывести сообщение об ошибке при попытке использования функции `open()` по причинам, которые не тестируются вызовами POSIX. Лучше всего вызывать функцию `open()` с требуемым режимом доступа и перехватывать исключение `IOError` в случае возникновения каких-либо проблем.

17.3.3. Создание и удаление каталогов

Для работы с каталогами файловой системы предназначено несколько функций, в том числе функции для создания каталогов, перечисления их содержимого и удаления.

Листинг 17.58. os_directories.py

```
import os

dir_name = 'os_directories_example'

print('Creating', dir_name)
os.makedirs(dir_name)

file_name = os.path.join(dir_name, 'example.txt')
print('Creating', file_name)
with open(file_name, 'wt') as f:
    f.write('example file')

print('Cleaning up')
os.unlink(file_name)
os.rmdir(dir_name)
```

Для создания и удаления каталогов предназначены два набора функций. При создании нового каталога с помощью функции `mkdir()` все родительские каталоги уже должны существовать. При удалении каталога с помощью функции `rmdir()` фактически удаляется лишь каталог, соответствующий последней части пути. В отличие от этого функции `makedirs()` и `removedirs()` оперируют в отношении всех узлов, принадлежащих данному пути. Функция `makedirs()` создает все части пути, которые до этого не существовали, а функция `removedirs()` удаляет все родительские каталоги при условии, что они пусты.

```
$ python3 os_directories.py

Creating os_directories_example
Creating os_directories_example/example.txt
Cleaning up
```

17.3.4. Работа с символическими ссылками

Для работы с символическими ссылками доступны функции, которые можно использовать на платформах, поддерживающих такую возможность.

Листинг 17.59. os_symlinks.py

```
import os

link_name = '/tmp/' + os.path.basename(__file__)

print('Creating link {} -> {}'.format(link_name, __file__))
os.symlink(__file__, link_name)

stat_info = os.lstat(link_name)
print('Permissions:', oct(stat_info.st_mode))

print('Points to:', os.readlink(link_name))

# Очистка.
os.unlink(link_name)
```

Функция `symlink()` используется для создания символической ссылки, а функция `readlink()` — для чтения ссылки и определения исходного файла, на который указывает ссылка. Функция `lstat()` аналогична функции `stat()`, но не следует по символическим ссылкам.

```
$ python3 os_symlinks.py

Creating link /tmp/os_symlinks.py -> os_symlinks.py
Permissions: 0o120755
Points to: os_symlinks.py
```

17.3.5. Безопасная замена существующего файла

Операция замены или переименования существующего файла не является идемпотентной и может приводить к переходу приложения в состояние гонки. Функции `rename()` и `replace()` реализуют безопасные алгоритмы этих действий, по возможности используя атомарные операции в случае POSIX-совместимых систем.

Листинг 17.60. `os_rename_replace.py`

```
import glob
import os


with open('rename_start.txt', 'w') as f:
    f.write('starting as rename_start.txt')

print('Starting:', glob.glob('rename*.txt'))

os.rename('rename_start.txt', 'rename_finish.txt')

print('After rename:', glob.glob('rename*.txt'))

with open('rename_finish.txt', 'r') as f:
    print('Contents:', repr(f.read()))

with open('rename_new_contents.txt', 'w') as f:
    f.write('ending with contents of rename_new_contents.txt')

os.replace('rename_new_contents.txt', 'rename_finish.txt')

with open('rename_finish.txt', 'r') as f:
    print('After replace:', repr(f.read()))

for name in glob.glob('rename*.txt'):
    os.unlink(name)
```

В большинстве случаев функции `rename()` и `replace()` способны работать с использованием различных файловых систем. Переименование файла может оказаться неуспешным, если он перемещается в другую файловую систему или конечный файл уже существует.

```
$ python3 os_rename_replace.py

Starting: ['rename_start.txt']
After rename: ['rename_finish.txt']
Contents: 'starting as rename_start.txt'
After replace: 'ending with contents of rename_new_contents.txt'
```

17.3.6. Определение и изменение владельца процесса

Рассматриваемый далее набор функций, предоставляемых модулем os, используется для определения и изменения идентификаторов владельцев процессов. Эти функции чаще всего применяются авторами демонов или специальных системных программ, для которых желательно изменить права доступа, а не выполняться от имени суперпользователя root. В данном разделе не делается никаких попыток объяснения сложных вопросов, касающихся системы безопасности Unix, а также владения процессами и организации работы с ними. Для получения более подробной информации по этому вопросу следует обратиться к дополнительным ссылкам, приведенным в конце раздела.

В следующем примере сначала отображается информация о реальных и эффективных пользователе и группе для процесса, а затем эффективные значения изменяются. Эти действия аналогичны тем, которые выполнял бы демон, запущенный от имени пользователя root во время первоначальной загрузки системы, для понижения уровня своих привилегий и выполнения от имени другого пользователя.

Примечание

Прежде чем выполнять этот пример, измените значения констант TEST_GID и TEST_UID таким образом, чтобы они соответствовали реальному пользователю, определенному в системе.

Листинг 17.61. os_process_user_example.py

```
import os

TEST_GID = 502
TEST_UID = 502


def show_user_info():
    print('User (actual/effective)  : {} / {}'.format(
        os.getuid(), os.geteuid()))
    print('Group (actual/effective) : {} / {}'.format(
        os.getgid(), os.getegid()))
    print('Actual Groups   :', os.getgroups())


print('BEFORE CHANGE:')
show_user_info()
print()
```

```
try:
    os.setegid(TEST_GID)
except OSError:
    print('ERROR: Could not change effective group. '
          'Rerun as root.')
else:
    print('CHANGE GROUP:')
    show_user_info()
    print()

try:
    os.seteuid(TEST_UID)
except OSError:
    print('ERROR: Could not change effective user. '
          'Rerun as root.')
else:
    print('CHANGE USER:')
    show_user_info()
    print()
```

При выполнении этого сценария на компьютере Mac, работающем под управлением OS X, с использованием значения 502 для идентификаторов пользователя и группы были получены следующие результаты.

```
$ python3 os_process_user_example.py

BEFORE CHANGE:
User (actual/effective) : 527 / 527
Group (actual/effective) : 501 / 501
Actual Groups : [501, 701, 402, 702, 500, 12, 61, 80, 98, 398,
399, 33, 100, 204, 395]

ERROR: Could not change effective group. Rerun as root.
ERROR: Could not change effective user. Rerun as root.
```

Значения идентификаторов не удалось изменить, поскольку процесс, выполняющийся не от имени пользователя `root`, не может изменить идентификатор своего эффективного владельца. Любая попытка установить для идентификатора эффективного пользователя или группы значение, отличное от идентификатора текущего пользователя, приводит к возбуждению исключения `OSError`. Другое дело — выполнение этого же сценария с помощью команды `sudo`, запускающей его с привилегиями `root`.

```
$ sudo python3 os_process_user_example.py

BEFORE CHANGE:

User (actual/effective)  : 0 / 0
Group (actual/effective) : 0 / 0
Actual Groups : [0, 1, 2, 3, 4, 5, 8, 9, 12, 20, 29, 61, 80,
702, 33, 98, 100, 204, 395, 398, 399, 701]
```

```
CHANGE GROUP:
User (actual/effective)  : 0 / 0
Group (actual/effective) : 0 / 502
Actual Groups   : [0, 1, 2, 3, 4, 5, 8, 9, 12, 20, 29, 61, 80,
702, 33, 98, 100, 204, 395, 398, 399, 701]

CHANGE USER:
User (actual/effective)  : 0 / 502
Group (actual/effective) : 0 / 502
Actual Groups   : [0, 1, 2, 3, 4, 5, 8, 9, 12, 20, 29, 61, 80,
702, 33, 98, 100, 204, 395, 398, 399, 701]
```

В этом случае, поскольку сценарий запускается от имени суперпользователя, он может изменить идентификатор эффективного пользователя или группы для процесса. Коль скоро идентификатор UID эффективного пользователя изменен, полномочия доступа для процесса ограничиваются разрешениями, установленными для данного пользователя. Поскольку непривилегированный пользователь не может изменить свою принадлежность к группе, программа должна изменить сначала идентификатор своей группы и только после этого — идентификатор пользователя.

17.3.7. Управление окружением процесса

Еще одним средством операционной системы, которое модуль `os` предоставляет программам, является *окружение* (также — *среда*). Переменные, установленные в окружении, доступны в виде строк, которые можно читать с помощью объекта `os.environ` или функции `getenv()`. Обычно переменные среды используются в качестве конфигурационных параметров, таких как пути поиска, расположения файлов и флаги отладки. В следующем примере показано, как можно получить переменную среды и передать через нее значение дочернему процессу.

Листинг 17.62. `os_environ_example.py`

```
import os

print('Initial value:', os.environ.get('TESTVAR', None))
print('Child process:')
os.system('echo $TESTVAR')

os.environ['TESTVAR'] = 'THIS VALUE WAS CHANGED'

print()
print('Changed value:', os.environ['TESTVAR'])
print('Child process:')
os.system('echo $TESTVAR')

del os.environ['TESTVAR']

print()
print('Removed value:', os.environ.get('TESTVAR', None))
print('Child process:')
os.system('echo $TESTVAR')
```

Объект `os.environ` следует принятым в Python стандартам API отображений для получения и установки значений. Изменения в объекте `os.environ` экспортируются в дочерние процессы.

```
$ python3 -u os_environ_example.py

Initial value: None
Child process:


Changed value: THIS VALUE WAS CHANGED
Child process:
THIS VALUE WAS CHANGED

Removed value: None
Child process:
```

17.3.8. Управление рабочим каталогом процесса

В операционных системах с иерархическими файловыми системами существует понятие *текущего рабочего каталога* — каталога файловой системы, используемого процессом в качестве начального при обращении к файлам с использованием относительных путей. Текущий рабочий каталог можно получить с помощью функции `getcwd()` и изменить с помощью функции `chdir()`.

Листинг 17.63. `os_cwd_example.py`

```
import os

print('Starting:', os.getcwd())

print('Moving up one:', os.pardir)
os.chdir(os.pardir)

print('After move:', os.getcwd())
```

Строки `os.curdir` и `os.pardir` обеспечивают портируемый способ обращения к текущему и родительскому каталогам соответственно.

```
$ python3 os_cwd_example.py

Starting: .../pymotw-3/source/os
Moving up one: ..
After move: .../pymotw-3/source
```

17.3.9. Выполнение внешних команд

Предупреждение

Многие из функций для работы с процессами имеют ограниченную портируемость. Если нужен более последовательный подход, обеспечивающий работу с процессами платформонезависимым способом, воспользуйтесь возможностями модуля `subprocess` (см. раздел 10.1).

Самым простым способом выполнения независимой команды, вообще не требующим никакого взаимодействия с ней, является использование функции `system()`. Эта функция получает единственный строковый аргумент, определяющий командную строку, которая должна быть выполнена подпроцессом, запускающим командную оболочку.

Листинг 17.64. `os_system_example.py`

```
import os

# Простая команда
os.system('pwd')
```

Функция `system()` возвращает значение кода выхода из оболочки, в которой выполняется программа, упакованное в 16-битовое число. Старший байт этого числа представляет код завершения программы, а младший — номер сигнала, прервавшего выполнение процесса, или нуль.

```
$ python3 -u os_system_example.py

.../pymotw-3/source/os
```

Поскольку команда передается для обработки непосредственно командной оболочке, она может включать любой допустимый для оболочки синтаксис, в том числе универсализацию имен файлов и переменные среды.

Листинг 17.65. `os_system_shell.py`

```
import os

# Команда с расширением оболочки
os.system('echo $TMPDIR')
```

Переменная среды `$TMPDIR` в этой строке расширяется во время выполнения оболочкой командной строки.

```
$ python3 -u os_system_shell.py

/var/folders/5q/8gk0wq888xlggz008k8dr7180000hg/T/
```

Если команда не запущена явным образом для выполнения в фоновом режиме, то вызов функции `system()` блокируется до ее полного завершения. Стандартные потоки `input`, `output` и `error` дочернего процесса по умолчанию связываются с соответствующими потоками, принадлежащими вызывающему коду, но их можно перенаправить, используя синтаксис командной оболочки.

Листинг 17.66. `os_system_background.py`

```
import os
import time

print('Calling...')
os.system('date; (sleep 3; date) &')
```

```
print('Sleeping...')
time.sleep(5)
```

Здесь мы вторгаемся в тонкости работы командной оболочки, однако для того, чтобы сделать то же самое, существуют более удобные способы.

```
$ python3 -u os_system_background.py

Calling...
Sat Dec 31 12:33:20 EST 2016
Sleeping...
Sat Dec 31 12:33:23 EST 2016
```

17.3.10. Создание процессов с помощью вызова `os.fork()`

Функции `fork()` и `exec()` из POSIX (доступны на платформах Mac OS X, Linux и других Unix-подобных платформах) предоставляются модулем `os`. Обсуждению надежных способов использования этих функций посвящены целые книги, поэтому для получения более подробных сведений, чем те, которые представлены в этом разделе, поищите соответствующую книгу.

Для создания нового процесса как клона текущего процесса нужно использовать функцию `fork()`.

Листинг 17.67. `os_fork_example.py`

```
import os

pid = os.fork()

if pid:
    print('Child process id:', pid)
else:
    print('I am the child')
```

Вывод будет меняться в зависимости от состояния системы во время выполнения примера, но в целом он будет подобен приведенному ниже.

```
$ python3 -u os_fork_example.py

Child process id: 29190
I am the child
```

После ответвления нового процесса существуют уже два процесса, выполняющих один и тот же код. Чтобы определить, какой процесс выполняется — родительский или дочерний, программа должна проверить значение, возвращаемое функцией `fork()`. Если оно равно `0`, значит, текущий процесс является дочерним. В противном случае программа выполняется в родительском процессе, а возвращенное функцией `fork()` значение является идентификатором дочернего процесса.

Листинг 17.68. os_kill_example.py

```
import os
import signal
import time


def signal_usr1(signum, frame):
    "Функция обратного вызова, срабатывающая при получении сигнала"
    pid = os.getpid()
    print('Received USR1 in process {}'.format(pid))


print('Forking...')
child_pid = os.fork()
if child_pid:
    print('PARENT: Pausing before sending signal...')
    time.sleep(1)
    print('PARENT: Signaling {}'.format(child_pid))
    os.kill(child_pid, signal.SIGUSR1)
else:
    print('CHILD: Setting up signal handler')
    signal.signal(signal.SIGUSR1, signal_usr1)
    print('CHILD: Pausing to wait for signal')
    time.sleep(5)
```

Родительский процесс может посылать сигналы дочернему, используя функцию `kill()` и модуль `signal` (см. раздел 10.2). Прежде всего следует определить обработчик, который будет вызываться при получении сигнала, затем вызвать функцию `fork()` и сделать короткую паузу в родительском процессе перед отправкой сигнала `USR1` с помощью функции `kill()`. В данном примере короткая пауза используется для того, чтобы предоставить дочернему процессу возможность установить обработчик сигнала. Реальному приложению вызов функции `sleep()` не понадобится. В дочернем процессе следует установить обработчик сигнала и сделать небольшую паузу, чтобы предоставить родительскому процессу достаточно времени для отправки сигнала.

```
$ python3 -u os_kill_example.py

Forking...
PARENT: Pausing before sending signal...
CHILD: Setting up signal handler
CHILD: Pausing to wait for signal
PARENT: Signaling 29193
Received USR1 in process 29193
```

Простой способ обработки отдельного поведения в дочернем процессе заключается в проверке возвращаемого значения функции `fork()` и использовании оператора ветвления. Для более сложного поведения может потребоваться более высокая степень разделения кода, чем та, которую может обеспечить простое ветвление. В других случаях может потребоваться обертка для программы. В обеих этих ситуациях для выполнения другой программы можно воспользоваться одной из функций семейства `exec*()`.

Листинг 17.69. os_exec_example.py

```
import os

child_pid = os.fork()
if child_pid:
    os.waitpid(child_pid, 0)
else:
    os.execlp('pwd', 'pwd', '-P')
```

Если программа выполняется функцией `exec()`, код этой программы заменяет код, выполняющийся в существующем процессе.

```
$ python3 os_exec_example.py

.../pymotw-3/source/os
```

В зависимости от того, в какой форме доступны аргументы, должны ли путь и окружение родительского процесса копироваться в дочерний, а также от ряда других факторов, могут применяться различные разновидности функции `exec()`. Во всех этих случаях первым аргументом является путь или имя файла, а остальные аргументы управляют тем, как выполняется программа. Аргументы передаются либо через командную строку, либо посредством изменения окружения процесса (см. описание объекта `os.environ` и функции `os.getenv()`). За более подробной информацией следует обратиться к документации библиотеки.

17.3.11. Ожидание завершения дочерних процессов

Для обхода ограничений Python, связанных с выполнением потоков и глобальной блокировкой интерпретатора, многие интенсивные в вычислительном отношении программы используют несколько процессов. Когда для выполнения нескольких задач запускается несколько процессов, основной процесс должен выжидать, пока один или несколько процессов не завершатся, прежде чем запустить новые, во избежание перегрузки сервера. Для этой цели можно использовать функцию `wait()` и родственные ей функции, в зависимости от обстоятельств.

Если не имеет значения, какой из дочерних процессов завершится первым, следует использовать функцию `wait()`. Она возвращает управление, как только завершится любой из дочерних процессов.

Листинг 17.70. os_wait_example.py

```
import osimport os
import sys
import time

for i in range(2):
    print('PARENT {}: Forking {}'.format(os.getpid(), i))
    worker_pid = os.fork()
    if not worker_pid:
        print('WORKER {}: Starting'.format(i))
        time.sleep(2 + i)
        print('WORKER {}: Finishing'.format(i))
```

```
        sys.exit(i)

for i in range(2):
    print('PARENT: Waiting for {}'.format(i))
    done = os.wait()
    print('PARENT: Child done:', done)
```

Функция `wait()` возвращает идентификатор процесса и код завершения, упакованный в 16-битовое значение. Младший байт представляет номер сигнала, прекратившего выполнение процесса, а старший — код состояния, возвращенный процессом по его завершении.

```
$ python3 -u os_wait_example.py

PARENT 29202: Forking 0
PARENT 29202: Forking 1
PARENT: Waiting for 0
WORKER 0: Starting
WORKER 1: Starting
WORKER 0: Finishing
PARENT: Child done: (29203, 0)
PARENT: Waiting for 1
WORKER 1: Finishing
PARENT: Child done: (29204, 256)
```

Для ожидания завершения конкретного процесса следует использовать функцию `waitpid()`.

Листинг 17.71. `os_waitpid_example.py`

```
import os
import sys
import time

workers = []
for i in range(2):
    print('PARENT {}: Forking {}'.format(os.getpid(), i))
    worker_pid = os.fork()
    if not worker_pid:
        print('WORKER {}: Starting'.format(i))
        time.sleep(2 + i)
        print('WORKER {}: Finishing'.format(i))
        sys.exit(i)
    workers.append(worker_pid)

for pid in workers:
    print('PARENT: Waiting for {}'.format(pid))
    done = os.waitpid(pid, 0)
    print('PARENT: Child done:', done)
```

Функция `waitpid()`, которой передан идентификатор целевого процесса, будет блокироваться до тех пор, пока процесс не завершится.

```
$ python3 -u os_waitpid_example.py

PARENT 29211: Forking 0
PARENT 29211: Forking 1
PARENT: Waiting for 29212
WORKER 0: Starting
WORKER 1: Starting
WORKER 0: Finishing
PARENT: Child done: (29212, 0)
PARENT: Waiting for 29213
WORKER 1: Finishing
PARENT: Child done: (29213, 256)
```

Функции `wait3()` и `wait4()` работают аналогичным образом, однако возвращают более подробную информацию о дочернем процессе, включающую идентификатор процесса, код завершения и информацию о потреблении ресурсов.

17.3.12. Создание новых процессов

Семейство функций `spawn()` обеспечивает более удобную обработку вызовов функций `fork()` и `exec()` в рамках одной инструкции.

Листинг 17.72. os_spawn_example.py

```
import os

os.spawnlp(os.P_WAIT, 'pwd', 'pwd', '-P')
```

Первый аргумент этой функции — `mode` — указывает, должна ли функция дожидаться завершения процесса, прежде чем вернуть управление. В этом примере выбран режим ожидания. Чтобы позволить другим процессам начать выполняться, а затем вернуть управление текущему процессу, необходимо указать режим `P_NOWAIT`.

```
$ python3 os_spawn_example.py

.../pymotw-3/source/os
```

17.3.13. Коды ошибок операционной системы

Коды ошибок, определенные операционной системой и контролируемые модулем `errno`, можно транслировать в строки сообщений, используя функцию `strerror()`.

Листинг 17.73. os_strerror.py

```
import errno
import os

for num in [errno.ENOENT, errno.EINTR, errno.EBUSY]:
    name = errno.errorcode[num]
    print('[{num:>2}] {name:<6}: {msg}'.format(
        name=name, num=num, msg=os.strerror(num)))
```

Приведенный ниже вывод отображает сообщения, связанные с кодами часто встречающихся ошибок.

```
$ python3 os_strerror.py

[ 2] ENOENT: No such file or directory
[ 4] EINTR : Interrupted system call
[16] EBUSY : Resource busy
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `os`[22].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `os` (раздел A.6.29).
- `signal` (раздел 10.2). Раздел, посвященный модулю `signal`, в котором принципы обработки сигналов рассмотрены более подробно.
- `subprocess` (раздел 10.1). Модуль `subprocess` заменяет функцию `os.popen()`.
- `multiprocessing` (раздел 10.4). Модуль `multiprocessing` упрощает работу с дополнительными процессами.
- `tempfile` (раздел 6.6). Модуль `tempfile` предназначен для работы с временными файлами.
- Раздел 6.7.3. Описание функций модуля `shutil` (раздел 6.7), предназначенных для работы с деревьями каталогов.
- *Speaking UNIX, Part 8*[23]. Обсуждение механизмов многозадачности в UNIX.
- Википедия: *Стандартные потоки*[24]. Дополнительное обсуждение потоков `stdin`, `stdout` и `stderr`.
- *Delve into Unix Process Creation*[25]. Объяснение жизненного цикла процесса Unix.
- *Advanced Programming in the UNIX Environment*, by W. Richard Stevens and Stephen A. Rago. Addison-Wesley, 2005. ISBN-10: 0201433079. В этой книге обсуждаются приемы работы с несколькими процессами, в том числе обработка сигналов, закрытие дубликатов файловых дескрипторов и многое другое.

17.4. `platform`: информация о версии системы

Несмотря на то что Python часто используется в качестве кроссплатформенного языка, иногда необходимо знать, в какого рода системе выполняется программа. В этой информации нуждаются инструменты автоматизации сборки программного обеспечения, но и приложению может быть известно, что некоторые библиотеки или внешние команды, которые в нем используются, имеют разные интерфейсы в различных операционных системах. Например, инструмент для управления сетевой конфигурацией операционной системы может определить переносимое представление сетевых интерфейсов, алиасных имен, IP-адресов

[22] https://docs.python.org/3.5/library/os.html

[23] www.ibm.com/developerworks/aix/library/au-speakingunix8/index.html

[24] https://ru.wikipedia.org/wiki/Стандартные_потоки

[25] www.ibm.com/developerworks/aix/library/au-unixprocess.html

и другой специфической для ОС информации. Однако при внесении изменений в конфигурационные файлы необходимо иметь больше сведений о хосте, которые обеспечили бы корректное использование конфигурационных файлов и команд, специфических для данной операционной системы. Модуль `platform` включает средства, позволяющие извлекать информацию об интерпретаторе, операционной системе и аппаратной платформе во время выполнения программы.

Примечание

Вывод примеров для этого раздела генерировался на трех системах: Mac mini с OS X 10.11.6, Dell PC с Ubuntu Linux 14.04 и VirtualBox VM с Windows 10. Установка Python в системах OS X и Windows выполнялась с использованием предварительно скомпилированных установщиков, загруженных с сайта `python.org`. В Linux выполнялась версия Python из системного пакета.

17.4.1. Интерпретатор

Для получения информации о текущем интерпретаторе Python используются четыре функции. Функции `python_version()` и `python_version_tuple()` возвращают разные формы представления версии Python, каждая из которых включает старший и младший номера версии, а также уровень обновления. Функция `python_compiler()` предоставляет информацию о компиляторе, который использовался для сборки интерпретатора, а функция `python_build()` — строку с номером и датой сборки интерпретатора.

Листинг 17.74. `platform_python.py`

```
import platform

print('Version      :', platform.python_version())
print('Version tuple:', platform.python_version_tuple())
print('Compiler     :', platform.python_compiler())
print('Build        :', platform.python_build())
```

OS X

```
$ python3 platform_python.py

Version      : 3.5.2
Version tuple: ('3', '5', '2')
Compiler     : GCC 4.2.1 (Apple Inc. build 5666) (dot 3)
Build        : ('v3.5.2:4def2a2901a5', 'Jun 26 2016 10:47:25')
```

Linux

```
$ python3 platform_python.py

Version      : 3.5.2
Version tuple: ('3', '5', '2')
Compiler     : GCC 4.8.4
Build        : ('default', 'Jul 17 2016 00:00:00')
```

Windows

```
C:\>Desktop\platform_python.py

Version      : 3.5.1
Version tuple: ('3', '5', '1')
Compiler     : MSC v.1900 64 bit (AMD64)
Build        : ('v3.5.1:37a07cee5969', 'Dec 6 2015 01:54:25')
```

17.4.2. Платформа

Функция `platform()` возвращает строку, содержащую универсальный идентификатор платформы. Эта функция имеет два необязательных булевых аргумента. Если аргумент `aliased` равен `True`, то имена в возвращаемом значении преобразуются из формального представления в более удобочитаемый вид. Если аргумент `terse` равен `True`, то вместо полной строки возвращается ее сокращенный вариант.

Листинг 17.75. `platform_platform.py`

```
import platform

print('Normal :', platform.platform())
print('Aliased:', platform.platform(aliased=True))
print('Terse  :', platform.platform(terse=True))
```

OS X

```
$ python3 platform_platform.py

Normal : Darwin-15.6.0-x86_64-i386-64bit
Aliased: Darwin-15.6.0-x86_64-i386-64bit
Terse  : Darwin-15.6.0
```

Linux

```
$ python3 platform_platform.py

Normal : Linux-3.13.0-55-generic-x86_64-with-Ubuntu-14.04-trusty
Aliased: Linux-3.13.0-55-generic-x86_64-with-Ubuntu-14.04-trusty
Terse  : Linux-3.13.0-55-generic-x86_64-with-glibc2.9
```

Windows

```
C:\>platform_platform.py

Normal : Windows-10-10.0.10240-SP0
Aliased: Windows-10-10.0.10240-SP0
Terse  : Windows-10
```

17.4.3. Информация об операционной системе и оборудовании

Также существует возможность получения более подробной информации об операционной системе и оборудовании платформы, на которой выполняется интерпретатор. Функция `uname()` возвращает кортеж, содержащий значения шести атрибутов: `system`, `node`, `release`, `version`, `machine` и `processor`. К каждому из этих значений можно обращаться по отдельности с помощью одноименных функций, приведенных в табл. 17.3.

Таблица 17.3. Функции для получения информации о платформе

Функция	Возвращаемое значение
`system()`	Имя операционной системы
`node()`	Неуточненное имя хоста сервера
`release()`	Номер выпуска операционной системы
`version()`	Более подробная информация о версии системы
`machine()`	Идентификатор типа оборудования, например `'i386'`
`processor()`	Реальный идентификатор процессора (во многих случаях совпадает со значением, возвращаемым функцией `machine()`)

Листинг 17.76. `platform_os_info.py`

```
import platform

print('uname:', platform.uname())

print()
print('system   :', platform.system())
print('node     :', platform.node())
print('release  :', platform.release())
print('version  :', platform.version())
print('machine  :', platform.machine())
print('processor:', platform.processor())
```

OS X

```
$ python3 platform_os_info.py

uname: uname_result(system='Darwin', node='hubert.local',
release='15.6.0', version='Darwin Kernel Version 15.6.0: Thu Jun
23 18:25:34 PDT 2016; root:xnu-3248.60.10~1/RELEASE_X86_64',
machine='x86_64', processor='i386')

system    : Darwin
node      : hubert.local
release   : 15.6.0
version   : Darwin Kernel Version 15.6.0: Thu Jun 23 18:25:34 PDT
2016; root:xnu-3248.60.10~1/RELEASE_X86_64
machine   : x86_64
processor : i386
```

Linux

```
$ python3 platform_os_info.py

uname: uname_result(system='Linux', node='apu',
release='3.13.0-55-generic', version='#94-Ubuntu SMP Thu Jun 18
00:27:10 UTC 2015', machine='x86_64', processor='x86_64')

system   : Linux
node     : apu
release  : 3.13.0-55-generic
version  : #94-Ubuntu SMP Thu Jun 18 00:27:10 UTC 2015
machine  : x86_64
processor: x86_64
```

Windows

```
C:\>Desktop\platform_os_info.py

uname: uname_result(system='Windows', node='IE11WIN10',
release='10', version='10.0.10240', machine='AMD64',
processor='Intel64 Family 6 Model 70 Stepping 1, GenuineIntel')

system   : Windows
node     : IE11WIN10
release  : 10
version  : 10.0.10240
machine  : AMD64
processor: Intel64 Family 6 Model 70 Stepping 1, GenuineIntel
```

17.4.4. Архитектура исполняемой программы

Информацию об архитектуре отдельной программы можно получить с помощью функции `architecture()`. Ее первым аргументом является путь к исполняемой программе (по умолчанию — `sys.executable`, интерпретатор Python), а возвращаемым значением — кортеж, содержащий разрядность архитектуры и используемый формат связывания.

Листинг 17.77. `platform_architecture.py`

```
import platform

print('interpreter:', platform.architecture())
print('/bin/ls :', platform.architecture('/bin/ls'))
```

OS X

```
$ python3 platform_architecture.py

interpreter: ('64bit', '')
/bin/ls    : ('64bit', '')
```

Linux

```
$ python3 platform_architecture.py

interpreter: ('64bit', 'ELF')
/bin/ls    : ('64bit', 'ELF')
```

Windows

```
C:\>Desktop\platform_architecture.py

interpreter: ('64bit', 'WindowsPE')
/bin/ls    : ('64bit', '')
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `platform`[26].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `platform` (раздел A.6.34).

17.5. `resource`: управление системными ресурсами

Модуль `resource` предоставляет функции, позволяющие измерять и контролировать текущие системные ресурсы, потребляемые процессом, и налагать на них ограничения, регулирующие допустимую нагрузку на систему.

17.5.1. Текущее потребление ресурсов

Функция `getrusage()` обеспечивает измерение ресурсов, используемых текущим процессом и/или его дочерними процессами. Ее возвращаемым значением является структура данных, содержащая ряд метрик, основанных на текущем состоянии ресурсов системы.

Примечание

В этом примере представлены не все ресурсы, доступные для измерения. Более полный их список содержится в разделе документации стандартной библиотеки, посвященном модулю `resource`.

Листинг 17.78. `resource_getrusage.py`

```
import resource
import time

RESOURCES = [
    ('ru_utime', 'User time'),
    ('ru_stime', 'System time'),
    ('ru_maxrss', 'Max. Resident Set Size'),
    ('ru_ixrss', 'Shared Memory Size'),
```

[26] https://docs.python.org/3.5/library/platform.html

```
    ('ru_idrss', 'Unshared Memory Size'),
    ('ru_isrss', 'Stack Size'),
    ('ru_inblock', 'Block inputs'),
    ('ru_oublock', 'Block outputs'),
]

usage = resource.getrusage(resource.RUSAGE_SELF)

for name, desc in RESOURCES:
    print('{:<25} ({:<10}) = {}'.format(
        desc, name, getattr(usage, name)))
```

Поскольку тестовая программа не отличается сложностью, она использует лишь небольшой набор ресурсов.

```
$ python3 resource_getrusage.py

User time                 (ru_utime  ) = 0.021876
System time               (ru_stime  ) = 0.0067269999999999995
Max. Resident Set Size    (ru_maxrss ) = 6479872
Shared Memory Size        (ru_ixrss  ) = 0
Unshared Memory Size      (ru_idrss  ) = 0
Stack Size                (ru_isrss  ) = 0
Block inputs              (ru_inblock) = 0
Block outputs             (ru_oublock) = 0
```

17.5.2. Лимитирование ресурсов

Кроме определения текущего фактического потребления ресурсов можно проверять действующие ограничения, налагаемые на приложение, и при необходимости изменять их.

Листинг 17.79. `resource_getrlimit.py`

```
import resource

LIMITS = [
    ('RLIMIT_CORE', 'core file size'),
    ('RLIMIT_CPU', 'CPU time'),
    ('RLIMIT_FSIZE', 'file size'),
    ('RLIMIT_DATA', 'heap size'),
    ('RLIMIT_STACK', 'stack size'),
    ('RLIMIT_RSS', 'resident set size'),
    ('RLIMIT_NPROC', 'number of processes'),
    ('RLIMIT_NOFILE', 'number of open files'),
    ('RLIMIT_MEMLOCK', 'lockable memory address'),
]

print('Resource limits (soft/hard):')
for name, desc in LIMITS:
    limit_num = getattr(resource, name)
    soft, hard = resource.getrlimit(limit_num)
    print('{:<23} {}/{}'.format(desc, soft, hard))
```

Для всех видов ограничений возвращаемое значение представляет собой кортеж, содержащий слабое предельное значение, налагаемое текущей конфигурацией, и строгое предельное значение, налагаемое операционной системой.

```
$ python3 resource_getrlimit.py

Resource limits (soft/hard):
core file size          0/9223372036854775807
CPU time                9223372036854775807/9223372036854775807
file size               9223372036854775807/9223372036854775807
heap size               9223372036854775807/9223372036854775807
stack size              8388608/67104768
resident set size       9223372036854775807/9223372036854775807
number of processes     709/1064
number of open files    7168/9223372036854775807
lockable memory address 9223372036854775807/9223372036854775807
```

Чтобы изменить предельные значения, следует использовать функцию `setrlimit()`.

Листинг 17.80. `resource_setrlimit_nofile.py`

```
import resource
import os

soft, hard = resource.getrlimit(resource.RLIMIT_NOFILE)
print('Soft limit starts as  :', soft)

resource.setrlimit(resource.RLIMIT_NOFILE, (4, hard))

soft, hard = resource.getrlimit(resource.RLIMIT_NOFILE)
print('Soft limit changed to :', soft)

random = open('/dev/random', 'r')
print('random has fd =', random.fileno())
try:
    null = open('/dev/null', 'w')
except IOError as err:
    print(err)
else:
    print('null has fd =', null.fileno())
```

В этом примере атрибут `RLIMIT_NOFILE` используется для управления разрешенным количеством одновременно открытых файлов, устанавливая для него слабое ограничение на уровне ниже того, который предусмотрен по умолчанию.

```
$ python3 resource_setrlimit_nofile.py

Soft limit starts as : 7168
Soft limit changed to : 4
random has fd = 3
[Errno 24] Too many open files: '/dev/null'
```

Также полезно ограничивать количество времени CPU, выделяемое процессу для выполнения, чтобы не допустить чрезмерно длительного владения этим ресурсом. Если длительность выполнения процесса превышает установленный предел, он получает сигнал SIGXCPU.

Листинг 17.81. resource_setrlimit_cpu.py

```
import resource
import sys
import signal
import time


# Установить обработчик сигнала, уведомляющий
# о превышении выделенного лимита процессорного времени
def time_expired(n, stack):
    print('EXPIRED :', time.ctime())
    raise SystemExit('(time ran out)')


signal.signal(signal.SIGXCPU, time_expired)

# Настроить лимит времени CPU
soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
print('Soft limit starts as  :', soft)

resource.setrlimit(resource.RLIMIT_CPU, (1, hard))

soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
print('Soft limit changed to :', soft)
print()

# Израсходовать некоторое количество времени CPU
# для проведения длительных вычислений в цикле
print('Starting:', time.ctime())
for i in range(200000):
    for i in range(200000):
        v = i * i

# Эта инструкция не будет достигнута
print('Exiting :', time.ctime())
```

Обычно обработчик сигнала сбрасывает на диск содержимое всех открытых файлов и закрывает их, но в данном случае он всего лишь выводит сообщение и осуществляет выход из программы.

```
$ python3 resource_setrlimit_cpu.py

Soft limit starts as : 9223372036854775807
Soft limit changed to : 1

Starting: Sun Aug 21 19:18:51 2016
EXPIRED : Sun Aug 21 19:18:52 2016
(time ran out)
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `resource`[27].
- `signal` (раздел 10.2). Подробное описание процедуры регистрации сигнала.

17.6. gc: сборщик мусора

Модуль `gc` предоставляет базовый механизм управления памятью в Python – автоматическую сборку мусора. Он включает функции, позволяющие принудительно вызывать сборщик мусора и исследовать известные системе объекты, которые либо намечены для удаления, либо связаны циклическими ссылками и не могут быть освобождены.

17.6.1. Отслеживание ссылок

Анализируя существующие между объектами входящие и исходящие ссылки, модуль `gc` способен выявлять наличие циклических ссылок в сложных структурах данных. Если речь идет о циклических ссылках в известной структуре, то ее свойства могут быть исследованы с помощью пользовательского кода. Если же циклические ссылки появляются в неизвестном коде, то для создания обобщенных средств отладки можно использовать функции `get_referents()` и `get_referrers()`.

Например, функция `get_referents()` отображает объекты, на которые ссылаются входные аргументы.

Листинг 17.82. gc_get_referents.py

```
import gc
import pprint


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


# Создать циклический граф
one = Graph('one')
two = Graph('two')
three = Graph('three')
```

[27] https://docs.python.org/3.5/library/resource.html

```
one.set_next(two)
two.set_next(three)
three.set_next(one)

print()
print('three refers to:')
for r in gc.get_referents(three):
    pprint.pprint(r)
```

В данном случае экземпляр three класса Graph содержит ссылки на свой словарь (в атрибуте __dict__) и свой класс.

```
$ python3 gc_get_referents.py

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)
three refers to:
{'name': 'three', 'next': Graph(one)}
<class '__main__.Graph'>
```

В следующем примере для обнаружения циклических ссылок выполняется обход всех объектов с применением алгоритма поиска "в ширину" (breadth-first traversal) и класса Queue. Элементами, помещаемыми в очередь, являются кортежи, содержащие уже имеющуюся цепочку ссылок плюс следующий объект, подлежащий проверке. Проверка начинается с экземпляра three — для него выполняется поиск всех объектов, на которые он ссылается. Пропуск классов означает, что их методы, модули и другие компоненты не проверяются.

Листинг 17.83. gc_get_referents_cycles.py

```
import gc
import pprint
import queue


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


# Создать циклический граф
one = Graph('one')
```

```
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

print()

seen = set()
to_process = queue.Queue()

# Начать с пустой цепочки объектов и узла three
to_process.put(([], three))

# Выполнить поиск циклов, создавая цепочку объектов для каждого
# объекта в цепочке, чтобы в конце можно было вывести полный цикл
while not to_process.empty():
    chain, next = to_process.get()
    chain = chain[:]
    chain.append(next)
    print('Examining:', repr(next))
    seen.add(id(next))
    for r in gc.get_referents(next):
        if isinstance(r, str) or isinstance(r, type):
            # Игнорировать строки и классы
            pass
        elif id(r) in seen:
            print()
            print('Found a cycle to {}:'.format(r))
            for i, link in enumerate(chain):
                print('  {}: '.format(i), end=' ')
                pprint.pprint(link)
        else:
            to_process.put((chain, r))
```

Образование циклической цепочки узлов легко обнаруживается путем отслеживания уже обработанных объектов. Чтобы не собирать ссылки на эти объекты, их значения `id()` кешируются в виде множества. Объекты словарей, обнаруженные в цикле, являются значениями `__dict__` для экземпляров `Graph` и хранят атрибуты их экземпляров.

```
$ python3 gc_get_referents_cycles.py

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

Examining: Graph(three)
Examining: {'next': Graph(one), 'name': 'three'}
Examining: Graph(one)
Examining: {'next': Graph(two), 'name': 'one'}
Examining: Graph(two)
Examining: {'next': Graph(three), 'name': 'two'}
```

```
Found a cycle to Graph(three):
  0:  Graph(three)
  1:  {'name': 'three', 'next': Graph(one)}
  2:  Graph(one)
  3:  {'name': 'one', 'next': Graph(two)}
  4:  Graph(two)
  5:  {'name': 'two', 'next': Graph(three)}
```

17.6.2. Принудительная сборка мусора

Несмотря на то что сборщик мусора выполняется автоматически по мере выполнения программы интерпретатором, его можно запускать в определенные моменты времени, если необходимо освободить память от большого количества ненужных объектов или если выполняется лишь небольшой объем полезной работы и запуск сборщика не повлияет на производительность программы. Для этого следует использовать функцию `collect()`.

Листинг 17.84. `gc_collect.py`

```
import gc
import pprint


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


# Создать циклический граф
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

# Удалить ссылки на узлы графа в пространстве имен данного модуля
one = two = three = None

# Продемонстрировать эффект сборки мусора
for i in range(2):
    print('\nCollecting {} ...'.format(i))
```

```
    n = gc.collect()
    print('Unreachable objects:', n)
    print('Remaining Garbage:', end=' ')
    pprint.pprint(gc.garbage)
```

В этом примере цикл удаляется сразу же, как только выполняется сборка мусора, поскольку на узлы Graph не ссылаются никакие другие объекты, кроме них самих. Функция collect() возвращает количество "недостижимых" объектов, которые ей удалось обнаружить. В данном случае это значение, равное 6, представляет три объекта с их словарями атрибутов экземпляров.

```
$ python3 gc_collect.py

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

Collecting 0 ...
Unreachable objects: 34
Remaining Garbage: []

Collecting 1 ...
Unreachable objects: 0
Remaining Garbage: []
```

17.6.3. Обнаружение ссылок на объекты, которые не могут быть отобраны сборщиком мусора

Найти объект, содержащий ссылку на другой объект, немного сложнее, чем определить объект, на который он ссылается. Поскольку код, запрашивающий информацию о ссылке, должен сам содержать эту ссылку, некоторые из ссылающихся объектов необходимо игнорировать. В следующем примере создается циклический граф, а затем обходятся все экземпляры Graph и удаляется ссылка из родительского узла.

Листинг 17.85. gc_get_referrers.py

```
import gc
import pprint


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
```

```
        return '{}({})'.format(
            self.__class__.__name__, self.name)

    def __del__(self):
        print('{}.__del__()'.format(self))


# Создать циклический граф
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

# Теперь сборщик мусора сохраняет объекты как
# не подлежащие сборке, а не как мусор
print()
print('Collecting...')
n = gc.collect()
print('Unreachable objects:', n)
print('Remaining Garbage:', end=' ')
pprint.pprint(gc.garbage)

# Игнорировать ссылки из локальных переменных этого модуля,
# глобальных переменных, а также из журнала сборщика мусора
REFERRERS_TO_IGNORE = [locals(), globals(), gc.garbage]


def find_referring_graphs(obj):
    print('Looking for references to {!r}'.format(obj))
    referrers = (r for r in gc.get_referrers(obj)
                 if r not in REFERRERS_TO_IGNORE)
    for ref in referrers:
        if isinstance(ref, Graph):
            # Узел графа
            yield ref
        elif isinstance(ref, dict):
            # Словарь экземпляра или другого пространства имен
            for parent in find_referring_graphs(ref):
                yield parent


# Выполнить поиск объектов, ссылающихся на объекты этого графа
print()
print('Clearing referrers:')
for obj in [one, two, three]:
    for ref in find_referring_graphs(obj):
        print('Found referrer:', ref)
        ref.set_next(None)
        del ref  # удалить ссылку, чтобы можно было удалить узел
    del obj  # удалить ссылку, чтобы можно было удалить узел
```

```
# Удалить ссылки, хранящиеся в gc.garbage
print()
print('Clearing gc.garbage:')
del gc.garbage[:]

# На этот раз должны быть удалены все объекты
print()
print('Collecting...')
n = gc.collect()
print('Unreachable objects:', n)
print('Remaining Garbage:', end=' ')
pprint.pprint(gc.garbage)
```

Если природа циклических ссылок понятна, то такого типа логика является чрезмерным усложнением. Тем не менее в случае необъясненных циклических ссылок в данных использование функции `get_referrers()` может выявить неожиданные отношения.

```
$ python3 gc_get_referrers.py

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

Collecting...
Unreachable objects: 28
Remaining Garbage: []

Clearing referrers:
Looking for references to Graph(one)
Looking for references to {'next': Graph(one), 'name': 'three'}
Found referrer: Graph(three)
Linking nodes Graph(three).next = None
Looking for references to Graph(two)
Looking for references to {'next': Graph(two), 'name': 'one'}
Found referrer: Graph(one)
Linking nodes Graph(one).next = None
Looking for references to Graph(three)
Looking for references to {'next': Graph(three), 'name': 'two'}
Found referrer: Graph(two)
Linking nodes Graph(two).next = None

Clearing gc.garbage:

Collecting...
Unreachable objects: 0
Remaining Garbage: []
Graph(one).__del__()
Graph(two).__del__()
Graph(three).__del__()
```

17.6.4. Пороги и поколения сборки мусора

Сборщик мусора поддерживает три списка объектов — по одному на каждое поколение сборки, отслеживаемое им. В процессе просмотра объектов в каждом поколении они либо удаляются, либо переводятся в следующее поколение, пока окончательно не достигнут состояния, в котором они постоянно находятся.

Выполняемые сборщиком рутинные операции можно настраивать так, чтобы они осуществлялись с различной частотой на основании разницы между количествами размещаемых и удаляемых объектов от запуска к запуску. Если количество размещений объектов минус количество удалений превышает установленный для данного поколения порог, запускается сборщик мусора. Для получения текущих пороговых значений следует использовать функцию `get_threshold()`.

Листинг 17.86. `gc_get_threshold.py`

```
import gc

print(gc.get_threshold())
```

Возвращаемое значение представляет собой кортеж из пороговых значений для каждого поколения.

```
$ python3 gc_get_threshold.py

(700, 10, 10)
```

Для изменения пороговых значений следует использовать функцию `set_threshold()`. В следующем примере программа устанавливает для поколения `0` пороговое значение, заданное аргументом командной строки, а затем размещает в памяти последовательность объектов.

Листинг 17.87. `gc_threshold.py`

```
import gc
import pprint
import sys

try:
    threshold = int(sys.argv[1])
except (IndexError, ValueError, TypeError):
    print('Missing or invalid threshold, using default')
    threshold = 5


class MyObj:

    def __init__(self, name):
        self.name = name
        print('Created', self.name)


gc.set_debug(gc.DEBUG_STATS)
```

```
gc.set_threshold(threshold, 1, 1)
print('Thresholds:', gc.get_threshold())

print('Clear the collector by forcing a run')
gc.collect()
print()

print('Creating objects')
objs = []
for i in range(10):
    objs.append(MyObj(i))
print('Exiting')

# Отключить отладку
gc.set_debug(0)
```

Изменение пороговых значений влияет на то, в какие моменты времени будет выполняться сборка мусора. Ниже эти значения отображаются, поскольку включен режим отладки.

```
$ python3 -u gc_threshold.py 5

gc: collecting generation 1...
gc: objects in each generation: 240 1439 4709
gc: done, 0.0013s elapsed
Thresholds: (5, 1, 1)
Clear the collector by forcing a run
gc: collecting generation 2...
gc: objects in each generation: 1 0 6282
gc: done, 0.0025s elapsed

gc: collecting generation 0...
gc: objects in each generation: 5 0 6275
gc: done, 0.0000s elapsed
Creating objects
gc: collecting generation 0...
gc: objects in each generation: 8 0 6275
gc: done, 0.0000s elapsed
Created 0
Created 1
Created 2
gc: collecting generation 1...
gc: objects in each generation: 9 2 6275
gc: done, 0.0000s elapsed
Created 3
Created 4
Created 5
gc: collecting generation 0...
gc: objects in each generation: 9 0 6280
gc: done, 0.0000s elapsed
Created 6
Created 7
Created 8
```

```
gc: collecting generation 0...
gc: objects in each generation: 9 3 6280
gc: done, 0.0000s elapsed
Created 9
Exiting
```

При меньших пороговых значениях сборка мусора выполняется чаще.

```
$ python3 -u gc_threshold.py 2

gc: collecting generation 1...
gc: objects in each generation: 240 1439 4709
gc: done, 0.0003s elapsed
Thresholds: (2, 1, 1)
Clear the collector by forcing a run
gc: collecting generation 2...
gc: objects in each generation: 1 0 6282
gc: done, 0.0010s elapsed
gc: collecting generation 0...
gc: objects in each generation: 3 0 6275
gc: done, 0.0000s elapsed

Creating objects
gc: collecting generation 0...
gc: objects in each generation: 6 0 6275
gc: done, 0.0000s elapsed
gc: collecting generation 1...
gc: objects in each generation: 3 4 6275
gc: done, 0.0000s elapsed
Created 0
Created 1
gc: collecting generation 0...
gc: objects in each generation: 4 0 6277
gc: done, 0.0000s elapsed
Created 2
gc: collecting generation 0...
gc: objects in each generation: 8 1 6277
gc: done, 0.0000s elapsed
Created 3
Created 4
gc: collecting generation 1...
gc: objects in each generation: 4 3 6277
gc: done, 0.0000s elapsed
Created 5
gc: collecting generation 0...
gc: objects in each generation: 8 0 6281
gc: done, 0.0000s elapsed
Created 6
Created 7
gc: collecting generation 0...
gc: objects in each generation: 4 2 6281
gc: done, 0.0000s elapsed
Created 8
```

```
gc: collecting generation 1...
gc: objects in each generation: 8 3 6281
gc: done, 0.0000s elapsed
Created 9
Exiting
```

17.6.5. Отладка

Обнаружение причин утечки памяти может оказаться трудной задачей. Модуль gc предоставляет несколько отладочных опций, которые упрощают решение этой задачи, позволяя увидеть детали работы внутренних механизмов кода. Указанные опции представляют собой битовые флаги, которые можно комбинировать и передавать функции set_debug() для управления поведением сборщика мусора во время выполнения программы. Отладочная информация выводится в поток sys.stderr.

Флаг DEBUG_STATS включает вывод статистики, вынуждая механизм сборщика мусора выводить в процессе своего выполнения информацию о том, сколько объектов отмечено для помещения в каждое поколение и сколько времени требуется для очистки памяти.

Листинг 17.88. gc_debug_stats.py

```
import gc

gc.set_debug(gc.DEBUG_STATS)

gc.collect()
print('Exiting')
```

Приведенный ниже вывод отображает результаты двух отдельных запусков сборщика мусора: когда он был вызван явным образом и при выходе из интерпретатора.

```
$ python3 gc_debug_stats.py

gc: collecting generation 2...
gc: objects in each generation: 123 1063 4711
gc: done, 0.0008s elapsed
Exiting
gc: collecting generation 2...
gc: objects in each generation: 1 0 5880
gc: done, 0.0007s elapsed
gc: collecting generation 2...
gc: objects in each generation: 99 0 5688
gc: done, 2114 unreachable, 0 uncollectable, 0.0011s elapsed
gc: collecting generation 2...
gc: objects in each generation: 0 0 3118
gc: done, 292 unreachable, 0 uncollectable, 0.0003s elapsed
```

Флаги DEBUG_COLLECTABLE и DEBUG_UNCOLLECTABLE вынуждают сборщик мусора выводить информацию о том, может или не может быть удален проверяемый объект. Если информации об объектах, которые не могут быть удалены, еще недо-

статочно для того, чтобы определить, где именно удерживаются данные, следует установить флаг DEBUG_SAVEALL, вынуждающий модуль gc сохранить все обнаруженные объекты, на которые отсутствуют ссылки, в списке garbage.

Листинг 17.89. gc_debug_saveall.py

```
import gc

flags = (gc.DEBUG_COLLECTABLE |
         gc.DEBUG_UNCOLLECTABLE |
         gc.DEBUG_SAVEALL
         )

gc.set_debug(flags)


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


class CleanupGraph(Graph):

    def __del__(self):
        print('{}.__del__()'.format(self))


# Создать циклический граф
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Создать самостоятельный узел
three = CleanupGraph('three')

# Создать циклический граф с помощью финализатора
four = CleanupGraph('four')
five = CleanupGraph('five')
four.set_next(five)
five.set_next(four)

# Удалить ссылки на узлы графа в пространстве имен этого модуля
one = two = three = four = five = None

# Принудительный запуск сборщика мусора
```

```
print('Collecting')
gc.collect()
print('Done')

# Вывести информацию об оставшихся объектах
for o in gc.garbage:
    if isinstance(o, Graph):
        print('Retained: {} 0x{:x}'.format(o, id(o)))

# Сбросить флаги отладки перед выходом во избежание дампа
# чрезмерно большого количества информации, что затруднило бы
# понимание примера
gc.set_debug(0)
```

Этот код разрешает проверку объектов после сборки мусора, что может пригодиться, скажем, в тех случаях, когда конструктор нельзя изменить таким образом, чтобы каждый раз, когда создается объект, выводился его идентификатор.

```
$ python3 -u gc_debug_saveall.py

CleanupGraph(three).__del__()
Collecting
gc: collectable <Graph 0x101be7240>
gc: collectable <Graph 0x101be72e8>
gc: collectable <dict 0x101994108>
gc: collectable <dict 0x101994148>
gc: collectable <CleanupGraph 0x101be73c8>
gc: collectable <CleanupGraph 0x101be7400>
gc: collectable <dict 0x101bee548>
gc: collectable <dict 0x101bee488>
CleanupGraph(four).__del__()
CleanupGraph(five).__del__()
Done
Retained: Graph(one) 0x101be7240
Retained: Graph(two) 0x101be72e8
Retained: CleanupGraph(four) 0x101be73c8
Retained: CleanupGraph(five) 0x101be7400
```

Для простоты определен флаг DEBUG_LEAK, представляющий комбинацию всех остальных опций.

Листинг 17.90. gc_debug_leak.py

```
import gc

flags = gc.DEBUG_LEAK

gc.set_debug(flags)


class Graph:

    def __init__(self, name):
        self.name = name
```

```
        self.next = None

    def set_next(self, next):
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


class CleanupGraph(Graph):

    def __del__(self):
        print('{}.__del__()'.format(self))


# Создать циклический граф
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Создать другой, независимый узел
three = CleanupGraph('three')

# Создать циклический граф с помощью финализатора
four = CleanupGraph('four')
five = CleanupGraph('five')
four.set_next(five)
five.set_next(four)

# Удалить ссылки на узлы графа в пространстве имен этого модуля
one = two = three = four = five = None

# Принудительная сборка мусора
print('Collecting')
gc.collect()
print('Done')

# Вывести информацию об оставшихся объектах
for o in gc.garbage:
    if isinstance(o, Graph):
        print('Retained: {} 0x{:x}'.format(o, id(o)))

# Сбросить флаги отладки перед выходом во избежание дампа
# чрезмерно большого количества информации, что сделало бы
# пример менее понятным
gc.set_debug(0)
```

Имейте в виду, что флаг DEBUG_LEAK автоматически включает флаг DEBUG_SAVEALL, в связи с чем будут сохранены даже те объекты, на которые отсутствуют ссылки и которые в обычных условиях были бы затребованы сборщиком мусора и в конечном счете удалены.

```
$ python3 -u gc_debug_leak.py

CleanupGraph(three).__del__()
Collecting
gc: collectable <Graph 0x1013e7240>
gc: collectable <Graph 0x1013e72e8>
gc: collectable <dict 0x101194108>
gc: collectable <dict 0x101194148>
gc: collectable <CleanupGraph 0x1013e73c8>
gc: collectable <CleanupGraph 0x1013e7400>
gc: collectable <dict 0x1013ee548>
gc: collectable <dict 0x1013ee488>
CleanupGraph(four).__del__()
CleanupGraph(five).__del__()
Done
Retained: Graph(one) 0x1013e7240
Retained: Graph(two) 0x1013e72e8
Retained: CleanupGraph(four) 0x1013e73c8
Retained: CleanupGraph(five) 0x1013e7400
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `gc`[28].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `gc` (раздел A.6.16).
- `weakref` (раздел 2.8). Модуль `weakref` позволяет создавать ссылки на объекты без увеличения их счетчиков ссылок, и поэтому они могут удаляться сборщиком мусора.
- *Supporting Cyclic Garbage Collection*[29]. Раздел дополнительной документации Python, посвященный C API.
- *How does Python manage memory?* (Fredrik Lundh) [30]. Статья, посвященная управлению памятью в Python.

17.7. `sysconfig`: управление конфигурацией интерпретатора во время компиляции

Средства, входящие в модуль `sysconfig`, были перенесены из модуля `distutils` с целью выделения их в виде независимого модуля. Этот модуль включает функции, которые определяют настройки, используемые для компиляции и установки текущего интерпретатора.

17.7.1. Конфигурационные переменные

Доступ к конфигурационным параметрам времени сборки предоставляют две функции: `get_config_vars()` и `get_config_var()`. Функция `get_config_`

[28] https://docs.python.org/3.5/library/gc.html

[29] https://docs.python.org/3/c-api/gcsupport.html

[30] http://effbot.org/pyfaq/how-does-python-manage-memory.htm

vars() возвращает словарь, сопоставляющий имена конфигурационных переменных с их значениями.

Листинг 17.91. sysconfig_get_config_vars.py

```
import sysconfig

config_values = sysconfig.get_config_vars()
print('Found {} configuration settings'.format(
    len(config_values.keys())))

print('\nSome highlights:\n')

print(' Installation prefixes:')
print('  prefix={prefix}'.format(**config_values))
print('  exec_prefix={exec_prefix}'.format(**config_values))

print('\n Version info:')
print('  py_version={py_version}'.format(**config_values))
print('  py_version_short={py_version_short}'.format(
    **config_values))
print('  py_version_nodot={py_version_nodot}'.format(
    **config_values))

print('\n Base directories:')
print('  base={base}'.format(**config_values))
print('  platbase={platbase}'.format(**config_values))
print('  userbase={userbase}'.format(**config_values))
print('  srcdir={srcdir}'.format(**config_values))

print('\n Compiler and linker flags:')
print('  LDFLAGS={LDFLAGS}'.format(**config_values))
print('  BASECFLAGS={BASECFLAGS}'.format(**config_values))
print('  Py_ENABLE_SHARED={Py_ENABLE_SHARED}'.format(
    **config_values))
```

Уровень детализации, доступный через API sysconfig, зависит от платформы, на которой выполняется программа. В случае POSIX-систем, таких как Linux и OS X, выполняется анализ файла *makefile*, используемого для сборки интерпретатора, и заголовочного файла *config.h*, сгенерированного для сборки, и все переменные, обнаруженные в каждом файле, становятся доступными. В случае таких систем, как Windows, которые не являются POSIX-системами, состав параметров ограничивается несколькими путями, расширениями имен файлов и информацией о версии.

```
$ python3 sysconfig_get_config_vars.py

Found 665 configuration settings

Some highlights:

 Installation prefixes:
  prefix=/Library/Frameworks/Python.framework/Versions/3.5
```

```
  exec_prefix=/Library/Frameworks/Python.framework/Versions/3.5

 Version info:
  py_version=3.5.2
  py_version_short=3.5
  py_version_nodot=35

 Base directories:
  base=/Users/dhellmann/Envs/pymotw35
  platbase=/Users/dhellmann/Envs/pymotw35
  userbase=/Users/dhellmann/Library/Python/3.5
  srcdir=/Library/Frameworks/Python.framework/Versions/3.5/lib/p
ython3.5/config-3.5m

 Compiler and linker flags:
  LDFLAGS=-arch i386 -arch x86_64 -g
  BASECFLAGS=-fno-strict-aliasing -Wsign-compare -fno-common
-dynamic
  Py_ENABLE_SHARED=0
```

В результате передачи имен переменных функции `get_config_vars()` возвращаемое значение превращается в список, который создается путем присоединения значений всех этих переменных.

Листинг 17.92. sysconfig_get_config_vars_by_name.py

```
import sysconfig

bases = sysconfig.get_config_vars('base', 'platbase', 'userbase')
print('Base directories:')
for b in bases:
    print('  ', b)
```

В этом примере создается список базовых каталогов установки, которые удается найти в текущей системе.

```
$ python3 sysconfig_get_config_vars_by_name.py

Base directories:
   /Users/dhellmann/Envs/pymotw35
   /Users/dhellmann/Envs/pymotw35
   /Users/dhellmann/Library/Python/3.5
```

Если необходимо значение лишь одного конфигурационного параметра, его можно получить с помощью функции `get_config_var()`.

Листинг 17.93. sysconfig_get_config_var.py

```
import sysconfig

print('User base directory:',
      sysconfig.get_config_var('userbase'))
print('Unknown variable :',
      sysconfig.get_config_var('NoSuchVariable'))
```

Если найти переменную не удается, функция `get_config_var()` не возбуждает исключение, а возвращает значение `None`.

```
$ python3 sysconfig_get_config_var.py

User base directory: /Users/dhellmann/Library/Python/3.5
Unknown variable   : None
```

17.7.2. Пути к каталогам установки

Модуль `sysconfig` в основном предназначен для использования инструментами установки и создания пакетов. Как следствие, хоть он и предоставляет доступ к общим параметрам конфигурации, таким как версия интерпретатора, он фокусируется на информации, необходимой для определения местонахождения компонентов дистрибутива Python, установленных в данный момент в системе. Местоположения, используемые для установки пакета, зависят от используемой *схемы*.

Схема — это набор платформозависимых каталогов, заданных по умолчанию и организованных в соответствии со стандартами и рекомендациями по созданию пакетов для данной платформы. Для установки в расположения, доступные всему сайту, или в личные каталоги, принадлежащие конкретным пользователям, используются различные схемы. Полный набор схем можно получить с помощью функции `get_scheme_names()`.

Листинг 17.94. `sysconfig_get_scheme_names.py`

```
import sysconfig

for name in sysconfig.get_scheme_names():
    print(name)
```

Понятия "текущая схема" как такового не существует. Вместо этого используемая по умолчанию схема зависит от опций, предоставленных программе установки. Если текущая система выполняется под управлением POSIX-совместимой операционной системы, то по умолчанию используется схема `posix_prefix`. В противном случае схемой по умолчанию является имя операционной системы, определенное в атрибуте `os.name`.

```
$ python3 sysconfig_get_scheme_names.py

nt
nt_user
osx_framework_user
posix_home
posix_prefix
posix_user
```

Каждая схема определяет набор путей, используемых для установки пакетов. Для получения списка имен этих путей следует использовать функцию `get_path_names()`.

Листинг 17.95. sysconfig_get_path_names.py

```
import sysconfig

for name in sysconfig.get_path_names():
    print(name)
```

Некоторые из этих путей могут совпадать для данной схемы, но программы-установщики не должны делать никаких предположений относительно того, что собой представляют фактические пути. Каждое имя имеет определенное семантическое значение, поэтому для поиска пути к конкретному файлу в процессе установки следует использовать корректное имя. Полный список имен путей вместе с их описаниями приведен в табл. 17.4.

Таблица 17.4. Имена путей, используемых модулем sysconfig

Имя	Описание
stdlib	Файлы стандартной библиотеки Python, не зависящие от платформы
platstdlib	Файлы стандартной библиотеки Python, зависящие от платформы
platlib	Файлы, зависящие от сайта и платформы
purelib	Файлы, зависящие от сайта, но не зависящие от платформы
include	Заголовочные файлы, не зависящие от платформы
platinclude	Заголовочные файлы, зависящие от платформы
scripts	Исполняемые файлы сценариев
data	Файлы данных

```
$ python3 sysconfig_get_path_names.py

stdlib
platstdlib
purelib
platlib
include
scripts
data
```

Фактические пути, ассоциированные со схемой, можно получить с помощью функции get_paths().

Листинг 17.96. sysconfig_get_paths.py

```
import sysconfig
import pprint
import os

for scheme in ['posix_prefix', 'posix_user']:
    print(scheme)
    print('=' * len(scheme))
    paths = sysconfig.get_paths(scheme=scheme)
    prefix = os.path.commonprefix(paths.values())
```

```
    print('prefix = {}\n'.format(prefix))
    for name, path in sorted(paths.items()):
        print('{}\n  .{}'.format(name, path[len(prefix):]))
    print()
```

Данный пример демонстрирует различия между путями, действующими в рамках всей системы и используемыми для схемы `posix_prefix` во фреймворке, созданном на Mac OS X, и специфическими для пользователя значениями для схемы `posix_user`.

```
$ python3 sysconfig_get_paths.py
posix_prefix
============
prefix = /Users/dhellmann/Envs/pymotw35
data
  .
include
  ./include/python3.5m
platinclude
  ./include/python3.5m
platlib
  ./lib/python3.5/site-packages
platstdlib
  ./lib/python3.5
purelib
  ./lib/python3.5/site-packages
scripts
  ./bin
stdlib
  ./lib/python3.5

posix_user
==========
prefix = /Users/dhellmann/Library/Python/3.5

data
  .
include
  ./include/python3.5
platlib
  ./lib/python3.5/site-packages
platstdlib
  ./lib/python3.5
purelib
  ./lib/python3.5/site-packages
scripts
  ./bin
stdlib
  ./lib/python3.5
```

Для получения отдельного пути следует использовать функцию `get_path()`.

Листинг 17.97. sysconfig_get_path.py

```
import sysconfig
import pprint

for scheme in ['posix_prefix', 'posix_user']:
    print(scheme)
    print('=' * len(scheme))
    print('purelib =', sysconfig.get_path(name='purelib',
                                          scheme=scheme))
    print()
```

Использование функции `get_path()` эквивалентно сохранению значения `get_paths()` и поиску отдельного ключа в словаре. Если требуется получить несколько путей, функция `get_paths()` более эффективна, поскольку она не вычисляет заново каждый раз все пути.

```
$ python3 sysconfig_get_path.py

posix_prefix
============
purelib = /Users/dhellmann/Envs/pymotw35/lib/python3.5/site-pack
ages

posix_user
==========
purelib = /Users/dhellmann/Library/Python/3.5/lib/python3.5/site
-packages
```

17.7.3. Информация о версии и платформе Python

В то время как модуль `sys` (см. раздел 17.2) включает некоторые базовые средства идентификации платформы (см. раздел 17.2.1.1), он недостаточно специфичен для того, чтобы его можно было использовать для установки двоичных пакетов, поскольку атрибут `sys.platform` не всегда включает информацию об архитектуре оборудования, размерах инструкций и других значениях, которые влияют на совместимость двоичных библиотек. Для получения более точного спецификатора платформы следует использовать функцию `get_platform()`.

Листинг 17.98. sysconfig_get_platform.py

```
import sysconfig

print(sysconfig.get_platform())
```

Используемый для подготовки этого простого вывода интерпретатор был скомпилирован с учетом совместимости с Mac OS X 10.6, а потому именно этот номер версии включается в строку идентификатора платформы.

```
$ python3 sysconfig_get_platform.py

macosx-10.6-intel
```

Для удобства версия интерпретатора из атрибута `sys.version_info` также доступна через функцию `get_python_version()` в модуле `sysconfig`.

Листинг 17.99. `sysconfig_get_python_version.py`

```
import sysconfig
import sys

print('sysconfig.get_python_version():',
      sysconfig.get_python_version())
print('\nsys.version_info:')
print('  major       :', sys.version_info.major)
print('  minor       :', sys.version_info.minor)
print('  micro       :', sys.version_info.micro)
print('  releaselevel:', sys.version_info.releaselevel)
print('  serial      :', sys.version_info.serial)
```

Функция `get_python_version()` возвращает строку, которую можно использовать для создания пути, специфического для версии.

```
$ python3 sysconfig_get_python_version.py

sysconfig.get_python_version(): 3.5

sys.version_info:
  major       : 3
  minor       : 5
  micro       : 2
  releaselevel: final
  serial      : 0
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `sysconfig`[31].
- `distutils`. Модуль `sysconfig` является частью пакета `distutils`.
- `site` (раздел 17.1). Модуль `site` более подробно описывает пути поиска при импорте модулей.
- `os` (раздел 17.3). Включает атрибут `os.name`, содержащий имя операционной системы.
- `sys` (раздел 17.2). Включает другую информацию времени сборки, например данные о платформе.

[31] https://docs.python.org/3.5/library/sysconfig.html

Глава 18

Инструменты языка

Кроме инструментов разработки, рассмотренных в предыдущей главе, Python включает модули, обеспечивающие доступ к внутренним средствам языка. В этой главе обсуждаются инструменты, предназначенные для работы с этими средствами, не зависящими от конкретики приложения.

Модуль `warnings` (раздел 18.1) используется для вывода предупреждений, информирующих о потенциальных проблемах или восстановимых ошибках. Типичным примером предупреждений могут служить сообщения `DeprecationWarning`, генерируемые в том случае, если используемый в приложении класс, интерфейс или модуль стандартной библиотеки был заменен новым средством и не рекомендуется к применению. Используйте предупреждения для того, чтобы информировать пользователя о возникновении ситуаций, заслуживающих его внимания, но не имеющих фатальных последствий.

Определение набора классов, удовлетворяющих требованиям общего API, может представлять трудности, если этот API определен другим разработчиком или использует множество методов. Популярным способом решения этой проблемы является наследование всех новых классов от общего базового класса, но не всегда очевидно, какие методы должны быть переопределены, а какие могут быть включены в поведение по умолчанию. Абстрактные базовые классы из модуля `abc` (раздел 18.2) формализуют API, явно обозначая, какие методы должны быть реализованы классом, и запрещая инстанциализацию класса, если он не реализовал полностью эти методы. Например, многие контейнерные типы Python имеют абстрактные базовые классы, определенные в модуле `abc` или модуле `collections` (раздел 2.2).

Модуль `dis` (раздел 18.3) может быть использован для дизассемблирования байт-кода программы с целью изучения того, какие шаги предпринимает интерпретатор для ее выполнения. Просмотр дизассемблированного кода может принести пользу при возникновении проблем с производительностью отладки или параллелизмом, поскольку позволяет увидеть атомарные операции, выполняемые интерпретатором для каждой инструкции программы.

Модуль `inspect` (раздел 18.4) обеспечивает поддержку интроспекции для всех объектов текущего процесса. К ним относятся импортированные модули, определения классов и функций, а также инстанциализированные на их основе объекты. Интроспекцию можно использовать для создания документации к исходному коду, динамической адаптации поведения объектов или исследования среды выполнения программы.

18.1. warnings: предупреждения о потенциальных проблемах

Модуль warnings был введен документом **PEP 230**[1] в качестве средства, позволяющего заблаговременно оповещать программистов об ожидаемых изменениях в языке и стандартной библиотеке и потенциальных проблемах обратной совместимости в связи с выходом версии Python 3.0. Его также можно использовать для вывода сообщений о восстановимых ошибках конфигурирования или снижении возможностей системы из-за отсутствия каких-либо библиотек. Однако для донесения информации, адресованной пользователям, все же лучше использовать модуль logging (раздел 14.8), поскольку предупреждения, выводимые на консоль, могут не достигать своей цели.

При условии, что предупреждения не относятся к фатальным ситуациям, одно и то же предупреждение может появляться несколько раз в ходе выполнения программы. Модуль warnings подавляет вывод повторных сообщений от одного и того же источника, чтобы они не раздражали пользователя. Выводом предупреждений можно очень легко управлять, используя опции командной строки интерпретатора или вызывая соответствующие функции, содержащиеся в модуле warnings.

18.1.1. Категории предупреждений и фильтрация

Предупреждения разбиты на категории с использованием подклассов встроенного класса исключений Warning. Некоторые стандартные значения описаны в онлайн-документации модуля exceptions, а пользовательские предупреждения можно добавить, наследуя от класса Warning.

Предупреждения обрабатываются на основе параметров *фильтров*. Фильтр состоит из пяти компонентов: action, message, category, module и line number. Компонент message — это регулярное выражение, используемое для сопоставления с текстом предупреждения. Компонент category — имя класса исключений. Компонент module содержит регулярное выражение, сопоставляемое с именем модуля, сгенерировавшего предупреждение. Компонент line number можно использовать для обработки специфических вхождений предупреждения.

Когда генерируется предупреждение, оно сравнивается со всеми зарегистрированными фильтрами. Первый совпавший фильтр управляет действием, предпринимаемым в ответ на появление предупреждения. Если ни один из фильтров не совпадает с предупреждением, выполняется действие, предусмотренное по умолчанию. Действия, распознаваемые механизмом фильтрации, описаны в табл. 18.1.

18.1.2. Генерация предупреждений

Простейшим способом выдачи предупреждения является вызов функции warn() с сообщением в качестве аргумента.

Листинг 18.1. warnings_warn.py

```
import warnings

print('Before the warning')
```

[1] www.python.org/dev/peps/pep-0230

```
warnings.warn('This is a warning message')
print('After the warning')
```

Сообщение выводится в ходе выполнения программы.

```
$ python3 -u warnings_warn.py

Before the warning
warnings_warn.py:13: UserWarning: This is a warning message
  warnings.warn('This is a warning message')
After the warning
```

Таблица 18.1. Действия фильтра предупреждений

Действие	Описание
`error`	Преобразовать предупреждение в исключение
`ignore`	Игнорировать предупреждение
`always`	Всегда выводить предупреждение
`default`	Выводить предупреждение, когда оно генерируется впервые в данном местоположении
`module`	Выводить предупреждение, когда оно генерируется впервые в данном модуле
`once`	Выводить предупреждение, когда оно генерируется впервые

Даже в случае вывода предупреждения программа по умолчанию продолжает выполняться. Это поведение можно изменить с помощью фильтра.

Листинг 18.2. `warnings_warn_raise.py`

```
import warnings

warnings.simplefilter('error', UserWarning)

print('Before the warning')
warnings.warn('This is a warning message')
print('After the warning')
```

В этом примере функция `simplefilter()` добавляет запись во внутренний список фильтров, информируя модуль `warnings` о необходимости возбуждения исключения, если сгенерировано предупреждение `UserWarning`.

```
$ python3 -u warnings_warn_raise.py

Before the warning
Traceback (most recent call last):
  File "warnings_warn_raise.py", line 15, in <module>
    warnings.warn('This is a warning message')
UserWarning: This is a warning message
```

Поведением фильтра можно также управлять из командной строки, используя параметр `-W` интерпретатора. Свойства фильтра указываются в виде строки, которая состоит из пяти компонентов (`action`, `message`, `category`, `module` и `line`

number), разделенных двоеточиями (:). Например, если выполнить сценарий warnings_warn.py с фильтром, установленным для возбуждения исключения при появлении сообщения UserWarning, будет сгенерировано исключение.

```
$ python3 -u -W "error::UserWarning::0" warnings_warn.py

Before the warning
Traceback (most recent call last):
  File "warnings_warn.py", line 13, in <module>
    warnings.warn('This is a warning message')
UserWarning: This is a warning message
```

Если поля для компонентов message и module оставить пустыми, то будет считаться, что они совпадают с любым значением.

18.1.3. Фильтрация с помощью шаблонов

Для добавления программным путем фильтров, основанных на более сложных правилах, следует использовать функцию filterwarnings(). Например, чтобы выполнить фильтрацию на основании содержимого текста сообщения, необходимо передать шаблон регулярного выражения в качестве аргумента message.

Листинг 18.3. warnings_filterwarnings_message.py

```
import warnings

warnings.filterwarnings('ignore', '.*do not.*',)

warnings.warn('Show this message')
warnings.warn('Do not show this message')
```

Шаблон содержит последовательность символов do not, но в фактическом сообщении используется текст Do not. Шаблон совпадет с ним, поскольку регулярное выражение всегда компилируется таким образом, чтобы при сопоставлении с текстом регистр символов игнорировался.

```
$ python3 warnings_filterwarnings_message.py

warnings_filterwarnings_message.py:14: UserWarning: Show this
message
  warnings.warn('Show this message')
```

Следующий пример генерирует два предупреждения.

Листинг 18.4. warnings_filter.py

```
import warnings

warnings.warn('Show this message')
warnings.warn('Do not show this message')
```

Одно из предупреждений можно проигнорировать, используя аргумент фильтра в командной строке.

```
$ python3 -W "ignore:do not:UserWarning::0" warnings_filter.py

warnings_filter.py:12: UserWarning: Show this message
  warnings.warn('Show this message')
```

Те же правила сопоставления с шаблоном применяются к имени модуля исходного кода, который содержит вызов, генерирующий предупреждение. Можно подавить вывод всех сообщений, генерируемых в модуле `warnings_filter`, передав имя модуля в качестве аргумента `module`.

Листинг 18.5. warnings_filterwarnings_module.py

```
import warnings

warnings.filterwarnings(
    'ignore',
    '.*',
    UserWarning,
    'warnings_filter',
)

import warnings_filter
```

Поскольку установлен фильтр, никакие предупреждения при импорте `warnings_filter` не выводятся.

```
$ python3 warnings_filterwarnings_module.py
```

Чтобы подавить лишь сообщение в строке 13 модуля `warnings_filter`, следует включить номер строки в качестве последнего аргумента функции `filterwarnings()`. Чтобы ограничить действие фильтра, следует использовать фактический номер строки исходного файла или `0` для применения фильтра ко всем вхождениям сообщения.

Листинг 18.6. warnings_filterwarnings_lineno.py

```
import warnings

warnings.filterwarnings(
    'ignore',
    '.*',
    UserWarning,
    'warnings_filter',
    13,
)

import warnings_filter
```

Шаблон совпадает с любым сообщением, поэтому важными аргументами являются имя модуля и номер строки.

```
$ python3 warnings_filterwarnings_lineno.py

.../warnings_filter.py:12: UserWarning: Show this message
  warnings.warn('Show this message')
```

18.1.4. Повторные предупреждения

По умолчанию большинство типов предупреждений выводятся только тогда, когда они впервые встречаются в данном местоположении, где "местоположение" определяется сочетанием имени модуля и номера строки, в которой генерируется предупреждение.

Листинг 18.7. warnings_repeated.py

```
import warnings


def function_with_warning():
    warnings.warn('This is a warning!')


function_with_warning()
function_with_warning()
function_with_warning()
```

В этом примере одна и та же функция вызывается несколько раз, но сообщение выводится только однократно.

```
$ python3 warnings_repeated.py

warnings_repeated.py:14: UserWarning: This is a warning!
  warnings.warn('This is a warning!')
```

Действие "once" можно использовать для подавления вывода экземпляров одного и того же сообщения из различных мест в коде.

Листинг 18.8. warnings_once.py

```
import warnings

warnings.simplefilter('once', UserWarning)

warnings.warn('This is a warning!')
warnings.warn('This is a warning!')
warnings.warn('This is a warning!')
```

Тексты сообщений сохраняются для всех предупреждений, и выводятся только уникальные предупреждения.

```
$ python3 warnings_once.py

warnings_once.py:14: UserWarning: This is a warning!
  warnings.warn('This is a warning!')
```

Точно так же действие "module" подавляет повторные сообщения от одного и того же модуля, независимо от того, в каких строках кода они генерируются.

18.1.5. Альтернативные функции доставки сообщений

Обычно предупреждения выводятся в поток sys.stderr. Чтобы изменить это поведение, следует заменить функцию showwarning() в модуле warnings. Например, чтобы отправить предупреждение в файл журнала, а не в стандартный поток ошибок, замените функцию showwarning() функцией, которая протоколирует предупреждения.

Листинг 18.9. warnings_showwarning.py

```
import warnings
import logging


def send_warnings_to_log(message, category, filename, lineno,
                         file=None):
    logging.warning(
        '%s:%s: %s:%s',
        filename, lineno,
        category.__name__, message,
    )

logging.basicConfig(level=logging.INFO)

old_showwarning = warnings.showwarning
warnings.showwarning = send_warnings_to_log

warnings.warn('message')
```

Предупреждения выводятся вместе с другими сообщениями протоколирования, если вызвана функция warn().

```
$ python3 warnings_showwarning.py

WARNING:root:warnings_showwarning.py:28: UserWarning:message
```

18.1.6. Форматирование

Если предупреждения должны направляться в стандартный поток ошибок, но при этом нуждаются в переформатировании, следует заменить функцию formatwarning().

Листинг 18.10. warnings_formatwarning.py

```
import warnings


def warning_on_one_line(message, category, filename, lineno,
                        file=None, line=None):
    return '-> {}:{}: {}:{}'.format(
```

```
        filename, lineno, category.__name__, message)

warnings.warn('Warning message, before')
warnings.formatwarning = warning_on_one_line
warnings.warn('Warning message, after')
```

Функция форматирования должна возвращать одиночную строку, содержащую представление предупреждения, которое должно быть отображено для пользователя.

```
$ python3 -u warnings_formatwarning.py

warnings_formatwarning.py:18: UserWarning: Warning message,
before
  warnings.warn('Warning message, before')
-> warnings_formatwarning.py:20: UserWarning:Warning message,
after
```

18.1.7. Глубина просмотра стека модулем `warnings`

По умолчанию сообщение предупреждения включает строку исходного кода, которая его сгенерировала, если она доступна. Однако изучение строки исходного кода с фактическим сообщением не всегда приносит какую-либо пользу. Вместо этого можно сообщить функции `warn()`, какой глубины стека она должна достигать в поиске строки, которая вызвала функцию, содержащую предупреждение. Благодаря этому пользователи устаревшей функции могут увидеть место в коде, где вызывается данная функция, а не ее реализацию.

Листинг 18.11. `warnings_warn_stacklevel.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4 import warnings
 5
 6
 7 def old_function():
 8     warnings.warn(
 9         'old_function() is deprecated, use new_function()',
10         stacklevel=2)
11
12
13 def caller_of_old_function():
14     old_function()
15
16
17 caller_of_old_function()
```

В этом примере функция `warn()` должна подняться по стеку на два уровня, один из которых соответствует ей самой, а второй — функции `old_function()`.

```
$ python3 warnings_warn_stacklevel.py

warnings_warn_stacklevel.py:14: UserWarning: old_function() is
deprecated,
 use new_function()
  old_function()
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `warnings`[2].
- **PEP 230**[3]. *Warning Framework*.
- `exceptions`. Базовые классы исключений и предупреждений.
- `logging` (раздел 14.8). Альтернативным способом доставки предупреждений является их запись в журнал.

18.2. abc: абстрактные базовые классы

Абстрактные базовые классы обеспечивают более строгий способ проверки интерфейса, чем индивидуальная проверка наличия конкретных методов с помощью метода `hasattr()`. Определив абстрактный базовый класс, можно установить общий API для набора подклассов. Такая возможность особенно полезна в ситуациях, когда другой разработчик, недостаточно хорошо знакомый с исходным кодом приложения, пишет для него расширения, но она также обеспечивает ряд преимуществ при командной разработке проекта или в случае большой кодовой базы, когда одновременное отслеживание всех классов затруднительно или невозможно.

18.2.1. Как работают абстрактные классы

Работа модуля `abc` заключается в том, чтобы пометить методы базового класса как абстрактные, а затем зарегистрировать конкретные классы как реализации абстрактного базового класса. Если приложению или библиотеке требуется конкретный API, то для проверки принадлежности абстрактному классу используются функции `issubclass()` и `isinstance()`.

Приступая к работе с модулем `abc`, определим абстрактный базовый класс для представления API набора плагинов, предназначенного для сохранения и загрузки данных. Далее зададим `ABCMeta` в качестве метакласса для нового базового класса и используем декораторы с целью создания общедоступного API для класса. В следующих примерах используется файл `abc_base.py`.

Листинг 18.12. `abc_base.py`

```
import abc


class PluginBase(metaclass=abc.ABCMeta):
```

[2] `https://docs.python.org/3.5/library/warnings.html`

[3] `www.python.org/dev/peps/pep-0230`

```
    @abc.abstractmethod
    def load(self, input):
        """Извлечь данные из входного источника
        и вернуть объект.
        """

    @abc.abstractmethod
    def save(self, output, data):
        """Сохранить объект данных для вывода."""
```

18.2.2. Регистрация конкретного класса

Существуют два способа указать, что конкретный класс реализует абстрактный API: явная регистрация класса и создание нового подкласса непосредственно на основе абстрактного класса. Если конкретный класс предоставляет требуемый API, но не является частью дерева наследования абстрактного базового класса, зарегистрируйте его, используя метод класса `register()` в качестве декоратора.

Листинг 18.13. `abc_register.py`

```
import abc
from abc_base import PluginBase


class LocalBaseClass:
    pass


@PluginBase.register
class RegisteredImplementation(LocalBaseClass):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)


if __name__ == '__main__':
    print('Subclass:', issubclass(RegisteredImplementation,
                                  PluginBase))
    print('Instance:', isinstance(RegisteredImplementation(),
                                  PluginBase))
```

В этом примере класс `RegisteredImplementation` создается путем наследования класса `LocalBaseClass`, но регистрируется как реализация API `PluginBase`. Следовательно, функции `issubclass()` и `isinstance()` трактуют его так, как если бы он был потомком класса `PluginBase`.

```
$ python3 abc_register.py

Subclass: True
Instance: True
```

18.2.3. Реализация посредством создания подклассов

Создание подкласса непосредственно на основе абстрактного базового класса позволяет избежать его явной регистрации.

Листинг 18.14. `abc_subclass.py`

```
import abc
from abc_base import PluginBase


class SubclassImplementation(PluginBase):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)


if __name__ == '__main__':
    print('Subclass:', issubclass(SubclassImplementation,
                                  PluginBase))
    print('Instance:', isinstance(SubclassImplementation(),
                                  PluginBase))
```

В этом случае для распознавания класса `SubclassImplementation` как реализации абстрактного класса `PluginBase` можно использовать обычные средства управления классами Python.

```
$ python3 abc_subclass.py

Subclass: True
Instance: True
```

Побочным эффектом непосредственного наследования является то, что это позволяет найти все реализации плагина, запросив у базового класса список известных классов, полученных из него путем наследования. (Эту возможность обеспечивает не только модуль `abc` – любой класс позволяет сделать это.)

Листинг 18.15. `abc_find_subclasses.py`

```
import abc
from abc_base import PluginBase
import abc_subclass
import abc_register

for sc in PluginBase.__subclasses__():
    print(sc.__name__)
```

Несмотря на включение `abc_register` в состав импортируемых модулей, класс `RegisteredImplementation` не входит в список найденных подклассов, поскольку он не получен путем фактического наследования базового класса.

```
$ python3 abc_find_subclasses.py

SubclassImplementation
```

18.2.4. Вспомогательный базовый класс

Если метакласс не установлен надлежащим образом, то программные интерфейсы не будут принудительно навязываться реализациям. Чтобы облегчить корректную настройку абстрактных классов, предоставляется базовый класс, в котором предусмотрено автоматическое определение метакласса.

Листинг 18.16. abc_abc_base.py

```
import abc


class PluginBase(abc.ABC):

    @abc.abstractmethod
    def load(self, input):
        """Извлечь данные из входного источника
        и вернуть объект.
        """

    @abc.abstractmethod
    def save(self, output, data):
        """Сохранить объект данных для вывода."""


class SubclassImplementation(PluginBase):

    def load(self, input):
        return input.read()

    def save(self, output, data):
        return output.write(data)


if __name__ == '__main__':
    print('Subclass:', issubclass(SubclassImplementation,
                                  PluginBase))
    print('Instance:', isinstance(SubclassImplementation(),
                                  PluginBase))
```

18.2.5. Неполные реализации

Создание подклассов непосредственно на основе абстрактного базового класса обладает тем дополнительным преимуществом, что их экземпляры не могут быть созданы, если они не реализуют полностью абстрактную часть API.

Листинг 18.17. abc_incomplete.py

```
import abc
from abc_base import PluginBase


@PluginBase.register
class IncompleteImplementation(PluginBase):

    def save(self, output, data):
        return output.write(data)


if __name__ == '__main__':
    print('Subclass:', issubclass(IncompleteImplementation,
                                  PluginBase))
    print('Instance:', isinstance(IncompleteImplementation(),
                                  PluginBase))
```

Эта мера исключает появление неожиданных ошибок времени выполнения в случае неполных реализаций.

```
$ python3 abc_incomplete.py

Subclass: True
Traceback (most recent call last):
  File "abc_incomplete.py", line 24, in <module>
    print('Instance:', isinstance(IncompleteImplementation(),
TypeError: Can't instantiate abstract class
IncompleteImplementation with abstract methods load
```

18.2.6. Конкретные методы в абстрактных базовых классах

Несмотря на то что конкретный класс обязан предоставить реализации всех абстрактных методов, абстрактный базовый класс также может предоставить реализацию, которую можно вызывать с помощью метода `super()`. После этого общую логику можно использовать повторно, помещая ее в базовый класс, но подклассы обязаны предоставлять перекрывающий метод с (потенциально) адаптированной логикой.

Листинг 18.18. abc_concrete_method.py

```
import abc
import io


class ABCWithConcreteImplementation(abc.ABC):

    @abc.abstractmethod
    def retrieve_values(self, input):
        print('base class reading data')
        return input.read()
```

```
class ConcreteOverride(ABCWithConcreteImplementation):

    def retrieve_values(self, input):
        base_data = super(ConcreteOverride,
                          self).retrieve_values(input)
        print('subclass sorting data')
        response = sorted(base_data.splitlines())
        return response


input = io.StringIO("""line one
line two
line three
""")

reader = ConcreteOverride()
print(reader.retrieve_values(input))
print()
```

Поскольку `ABCWithConcreteImplementation` — абстрактный базовый класс, невозможно создать его экземпляр, который можно было бы непосредственно использовать. Подклассы должны предоставлять переопределение метода `retrieve_values()`, и в этом случае конкретный класс сортирует данные, прежде чем вернуть их.

```
$ python3 abc_concrete_method.py

base class reading data
subclass sorting data
['line one', 'line three', 'line two']
```

18.2.7. Абстрактные свойства

Если в дополнение к методам спецификация API включает атрибуты, она может затребовать определение атрибутов в конкретных классах путем сочетания методов `abstractmethod()` и `property()`.

Листинг 18.19. `abc_abstractproperty.py`

```
import abc


class Base(abc.ABC):

    @property
    @abc.abstractmethod
    def value(self):
        return 'Should never reach here'

    @property
    @abc.abstractmethod
    def constant(self):
        return 'Should never reach here'
```

```
class Implementation(Base):

    @property
    def value(self):
        return 'concrete property'

    constant = 'set by a class attribute'


try:
    b = Base()
    print('Base.value:', b.value)
except Exception as err:
    print('ERROR:', str(err))

i = Implementation()
print('Implementation.value    :', i.value)
print('Implementation.constant:', i.constant)
```

В этом примере класс `Base` не может быть инстанциализирован, поскольку он имеет лишь абстрактные версии методов-получателей для свойств `value` и `constant`. В классе `Implementation` свойству `value` предоставляется конкретный метод-получатель, а свойство `constant` определяется через атрибут класса.

```
$ python3 abc_abstractproperty.py

ERROR: Can't instantiate abstract class Base with abstract
methods constant, value
Implementation.value : concrete property
Implementation.constant: set by a class attribute
```

Также возможно определение свойств с разрешением доступа к ним по чтению и записи.

Листинг 18.20. `abc_abstractproperty_rw.py`

```
import abc


class Base(abc.ABC):

    @property
    @abc.abstractmethod
    def value(self):
        return 'Should never reach here'

    @value.setter
    @abc.abstractmethod
    def value(self, new_value):
        return


class PartialImplementation(Base):
```

```
    @property
    def value(self):
        return 'Read-only'


class Implementation(Base):

    _value = 'Default value'

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
        self._value = new_value


try:
    b = Base()
    print('Base.value:', b.value)
except Exception as err:
    print('ERROR:', str(err))

p = PartialImplementation()
print('PartialImplementation.value:', p.value)

try:
    p.value = 'Alteration'
    print('PartialImplementation.value:', p.value)
except Exception as err:
    print('ERROR:', str(err))

i = Implementation()
print('Implementation.value:', i.value)

i.value = 'New value'
print('Changed value:', i.value)
```

Конкретное свойство должно определяться так же, как и абстрактное: как доступное для чтения/записи или только для чтения. Переопределение в классе `PartialImplementation` свойства, доступного для чтения и записи, как доступного только для чтения, делает его доступным только для чтения, т.е. метод-установщик данного свойства повторно не используется.

```
$ python3 abc_abstractproperty_rw.py

ERROR: Can't instantiate abstract class Base with abstract
methods value
PartialImplementation.value: Read-only
ERROR: can't set attribute
Implementation.value: Default value
Changed value: New value
```

Чтобы синтаксис декоратора был применим к абстрактным свойствам, доступным для чтения и записи, имена метода-получателя и метода-установщика такого свойств должны совпадать.

18.2.8. Абстрактный класс и статические методы

Методы класса и статические методы также могут помечаться как абстрактные.

Листинг 18.21. abc_class_static.py

```
import abc


class Base(abc.ABC):

    @classmethod
    @abc.abstractmethod
    def factory(cls, *args):
        return cls()

    @staticmethod
    @abc.abstractmethod
    def const_behavior():
        return 'Should never reach here'


class Implementation(Base):

    def do_something(self):
        pass

    @classmethod
    def factory(cls, *args):
        obj = cls(*args)
        obj.do_something()
        return obj

    @staticmethod
    def const_behavior():
        return 'Static behavior differs'


try:
    o = Base.factory()
    print('Base.value:', o.const_behavior())
except Exception as err:
    print('ERROR:', str(err))

i = Implementation.factory()
print('Implementation.const_behavior :', i.const_behavior())
```

Несмотря на то что метод класса вызывается для класса, а не для экземпляра, он по-прежнему предотвращает создание экземпляров класса, если они не переопределяют этот метод.

```
$ python3 abc_class_static.py

ERROR: Can't instantiate abstract class Base with abstract
methods const_behavior, factory
Implementation.const_behavior : Static behavior differs
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `abc`[4].
- **PEP 3119**[5]. *Introducing Abstract Base Classes*.
- `collections` (раздел 2.2). Модуль `collections` включает абстрактные базовые классы для нескольких типов коллекций.
- **PEP 3141**[6]. *A Type Hierarchy for Numbers*.
- Википедия: *Стратегия (шаблон проектирования)*[7]. Описание и примеры использования шаблона Strategy — широко используемого шаблона реализации плагинов.
- *Dynamic Code Patterns: Extending Your Applications with Plugins* (Doug Hellmann)[8]. Материал, представленный на конференции PyCon 2013.
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `abc` (раздел A.6.1).

18.3. `dis`: дизассемблирование байт-кода Python

Модуль `dis` включает функции, позволяющие работать с байт-кодом Python, представленным в удобочитаемой форме посредством дизассемблирования. Изучение байт-кода, выполняемого интерпретатором, предоставляет отличные возможности для тонкой настройки критических циклов вручную и выполнения других видов оптимизации кода. Кроме того, это может пригодиться для обнаружения состояний гонки в многопоточных процессах, поскольку позволяет оценить, где находятся точки в коде, в которых происходит переключение управления потоком выполнения.

Предупреждение

Используемые байт-коды зависят от деталей реализации, специфических для версии интерпретатора CPython. С каноническим списком байт-кодов для используемой вами версии интерпретатора вы сможете ознакомиться, заглянув в заголовочный файл *Include/opcode.h*, входящий в состав исходного кода.

18.3.1. Простой пример дизассемблирования

Функция `dis()` выводит дизассемблированное представление исходного кода Python (модуль, класс, метод, функция или объект кода). Модули наподобие при-

[4] `https://docs.python.org/3.5/library/abc.html`

[5] `www.python.org/dev/peps/pep-3119`

[6] `www.python.org/dev/peps/pep-3141`

[7] `https://ru.wikipedia.org/wiki/Стратегия_(шаблон_проектирования)`

[8] `http://pyvideo.org/pycon-us-2013/dynamic-code-patterns-extending-your-application.html`

веденного ниже можно дизассемблировать, выполнив команду dis в командной строке.

Листинг 18.22. dis_simple.py

```
#!/usr/bin/env python3
# encoding: utf-8

my_dict = {'a': 1}
```

Вывод организован в виде колонок, содержащих номер строки исходного кода, адрес инструкции в объекте кода, имя опкода (кода операции) и аргументы, переданные опкоду.

```
$ python3 -m dis dis_simple.py

  4           0 LOAD_CONST               0 ('a')
              3 LOAD_CONST               1 (1)
              6 BUILD_MAP                1
              9 STORE_NAME               0 (my_dict)
             12 LOAD_CONST               2 (None)
             15 RETURN_VALUE
```

В данном случае исходный код транслируется в четыре операции, создающие и заполняющие значениями словарь, а затем сохраняющие результаты в локальной переменной. Поскольку интерпретатор Python использует стек, то прежде всего константы помещаются в стек в корректном порядке с помощью команды LOAD_CONST, после чего команда BUILD_MAP извлекает из стека новый ключ и значение для добавления в словарь. Результирующий объект dict связывается с именем my_dict с помощью команды STORE_NAME.

18.3.2. Дизассемблирование функций

К сожалению, дизассемблирование модуля в целом не включает автоматический рекурсивный заход в функции.

Листинг 18.23. dis_function.py

```
#!/usr/bin/env python3
# encoding: utf-8


def f(*args):
    nargs = len(args)
    print(nargs, args)


if __name__ == '__main__':
    import dis
    dis.dis(f)
```

В результатах дизассемблирования исходного кода `dis_function.py` отображаются операции для загрузки объекта кода функции в стек и его последующего преобразования в функцию (`LOAD_CONST`, `MAKE_FUNCTION`), но не тело функции.

```
$ python3 -m dis dis_function.py
  5           0 LOAD_CONST               0 (<code object f at
0x10141ba50, file "dis_function.py", line 5>)
              3 LOAD_CONST               1 ('f')
              6 MAKE_FUNCTION            0
              9 STORE_NAME               0 (f)

 10          12 LOAD_NAME                1 (__name__)
             15 LOAD_CONST               2 ('__main__')
             18 COMPARE_OP               2 (==)
             21 POP_JUMP_IF_FALSE       49

 11          24 LOAD_CONST               3 (0)
             27 LOAD_CONST               4 (None)
             30 IMPORT_NAME              2 (dis)
             33 STORE_NAME               2 (dis)

 12          36 LOAD_NAME                2 (dis)
             39 LOAD_ATTR                2 (dis)
             42 LOAD_NAME                0 (f)
             45 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             48 POP_TOP
        >>   49 LOAD_CONST               4 (None)
             52 RETURN_VALUE
```

Чтобы заглянуть внутрь функции, следует передать ее функции `dis()`.

```
$ python3 dis_function.py

  6           0 LOAD_GLOBAL              0 (len)
              3 LOAD_FAST                0 (args)
              6 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
              9 STORE_FAST               1 (nargs)

  7          12 LOAD_GLOBAL              1 (print)
             15 LOAD_FAST                1 (nargs)
             18 LOAD_FAST                0 (args)
             21 CALL_FUNCTION            2 (2 positional, 0
keyword pair)
             24 POP_TOP
             25 LOAD_CONST               0 (None)
             28 RETURN_VALUE
```

Чтобы вывести краткую информацию о функции, включая аргументы и используемые имена, следует вызвать функцию `show_code()`, передав ей исследуемую функцию в качестве первого аргумента.

```
#!/usr/bin/env python3

# encoding: utf-8


def f(*args):
    nargs = len(args)
    print(nargs, args)


if __name__ == '__main__':
    import dis
    dis.show_code(f)
```

Полученный функцией `show_code()` аргумент передается функции `code_info()`, которая возвращает аккуратно отформатированную сводку сведений о функции, методе, строке или другом объекте кода, готовую к выводу на экран.

```
$ python3 dis_show_code.py

Name:              f
Filename:          dis_show_code.py
Argument count:    0
Kw-only arguments: 0
Number of locals:  2
Stack size:        3
Flags:             OPTIMIZED, NEWLOCALS, VARARGS, NOFREE
Constants:
   0: None
Names:
   0: len
   1: print
Variable names:
   0: args
   1: nargs
```

18.3.3. Классы

Классы также можно передавать функции `dis()`, которая в данном случае по очереди дизассемблирует все методы.

Листинг 18.24. `dis_class.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4 import dis
 5
 6
 7 class MyObject:
 8     """Пример для модуля dis."""
 9
```

```
10      CLASS_ATTRIBUTE = 'some value'
11
12      def __str__(self):
13          return 'MyObject({})'.format(self.name)
14
15      def __init__(self, name):
16          self.name = name
17
18
19  dis.dis(MyObject)
```

Методы выводятся в алфавитном порядке, а не в порядке их появления в файле.

```
$ python3 dis_class.py

Disassembly of __init__:
 16           0 LOAD_FAST                1 (name)
              3 LOAD_FAST                0 (self)
              6 STORE_ATTR               0 (name)
              9 LOAD_CONST               0 (None)
             12 RETURN_VALUE

Disassembly of __str__:
 13           0 LOAD_CONST               1 ('MyObject({})')
              3 LOAD_ATTR                0 (format)
              6 LOAD_FAST                0 (self)
              9 LOAD_ATTR                1 (name)
             12 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             15 RETURN_VALUE
```

18.3.4. Исходный код

Часто удобнее работать с исходным кодом программы, чем с самими объектами кода. Функции в модуле `dis` получают строковые аргументы, содержащие исходный код, и преобразуют их в объекты кода перед дизассемблированием или выводом других результатов.

Листинг 18.25. `dis_string.py`

```
import dis

code = """
my_dict = {'a': 1}
"""

print('Disassembly:\n')
dis.dis(code)

print('\nCode details:\n')
dis.show_code(code)
```

Передача строки означает, что стадия компиляции кода и сохранения ссылки на результаты может быть опущена. Такой подход более удобен в тех случаях, когда исследуются инструкции вне функции.

```
$ python3 dis_string.py

Disassembly:

  2           0 LOAD_CONST               0 ('a')
              3 LOAD_CONST               1 (1)
              6 BUILD_MAP                1
              9 STORE_NAME               0 (my_dict)
             12 LOAD_CONST               2 (None)
             15 RETURN_VALUE

Code details:

Name:              <module>
Filename:          <disassembly>
Argument count:    0
Kw-only arguments: 0
Number of locals:  0
Stack size:        2
Flags: NOFREE
Constants:
   0: 'a'
   1: 1
   2: None
Names:
   0: my_dict
```

18.3.5. Использование дизассемблирования в целях отладки

Иногда при отладке исключений полезно знать, какой именно байт-код стал виновником возникновения проблем. Существует несколько вариантов дизассемблирования проблемных мест кода. Одна из стратегий заключается в использовании функции `dis()` в интерактивном интерпретаторе для вывода информации о последнем исключении. Если никакие аргументы функции `dis()` не передаются, она находит исключение и дизассемблирует верхний кадр стека, ставший причиной его возникновения.

```
$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import dis
>>> j = 4
>>> i = i + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'i' is not defined
>>> dis.dis()
  1 -->        0 LOAD_NAME                0 (i)
               3 LOAD_CONST               0 (4)
               6 BINARY_ADD
               7 STORE_NAME               0 (i)
              10 LOAD_CONST               1 (None)
              13 RETURN_VALUE
>>>
```

Стрелка (-->) после номера строки указывает на опкод, в котором произошла ошибка. Переменная `i` не определена, поэтому связанное с ней значение не может быть загружено в стек.

Программа также может выводить информацию об активном стеке вызовов, передавая его непосредственно функции `distb()`. В следующем примере идентифицируется исключение `DivideByZero`, но поскольку формула включает две операции деления, не всегда очевидно, какой именно элемент имеет нулевое значение.

Листинг 18.26. `dis_traceback.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4 i = 1
 5 j = 0
 6 k = 3
 7
 8 try:
 9     result = k * (i / j) + (i / k)
10 except:
11     import dis
12     import sys
13     exc_type, exc_value, exc_tb = sys.exc_info()
14     dis.distb(exc_tb)
```

Ошибку легко локализовать, если она загружена в стек в дизассемблированном виде. Некорректная операция указана посредством стрелки (-->), а предыдущая строка помещает в стек значение `j`.

```
$ python3 dis_traceback.py

  4           0 LOAD_CONST               0 (1)
              3 STORE_NAME               0 (i)

  5           6 LOAD_CONST               1 (0)
              9 STORE_NAME               1 (j)

  6          12 LOAD_CONST               2 (3)
             15 STORE_NAME               2 (k)

  8          18 SETUP_EXCEPT            26 (to 47)

  9          21 LOAD_NAME                2 (k)
```

```
              24 LOAD_NAME                0 (i)
              27 LOAD_NAME                1 (j)
    -->       30 BINARY_TRUE_DIVIDE
              31 BINARY_MULTIPLY
              32 LOAD_NAME                0 (i)
              35 LOAD_NAME                2 (k)
              38 BINARY_TRUE_DIVIDE
              39 BINARY_ADD
              40 STORE_NAME               3 (result)

...продолжение вывода...
```

18.3.6. Анализ производительности циклов

Модуль `dis` способен помочь не только при отладке ошибок, но и при решении проблем с производительностью. Исследование дизассемблированного кода особенно полезно в случае критических циклов, когда количество инструкций невелико, но они транслируются в неэффективный набор байт-кодов. В пользе дизассемблирования нетрудно убедиться на примере исследования нескольких различных реализаций класса `Dictionary`, читающего список слов и группирующего их по первой букве.

Листинг 18.27. `dis_test_loop.py`

```
import dis
import sys
import textwrap
import timeit

module_name = sys.argv[1]
module = __import__(module_name)
Dictionary = module.Dictionary

dis.dis(Dictionary.load_data)
print()
t = timeit.Timer(
    'd = Dictionary(words)',
    textwrap.dedent("""
    from {module_name} import Dictionary
    words = [
        l.strip()
        for l in open('/usr/share/dict/words', 'rt')
    ]
    """).format(module_name=module_name)
)
iterations = 10
print('TIME: {:0.4f}'.format(t.timeit(iterations) / iterations))
```

Чтобы протестировать каждый из вариантов реализации класса `Dictionary`, используем приложение `dis_test_loop.py`, начав с самого простого, но медленного варианта.

Листинг 18.28. dis_slow_loop.py

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4
 5 class Dictionary:
 6
 7     def __init__(self, words):
 8         self.by_letter = {}
 9         self.load_data(words)
10
11     def load_data(self, words):
12         for word in words:
13             try:
14                 self.by_letter[word[0]].append(word)
15             except KeyError:
16                 self.by_letter[word[0]] = [word]
```

Ниже приведены результаты тестирования этой версии, отображающие дизассемблированный код программы и время, которое ушло на ее выполнение.

```
$ python3 dis_test_loop.py dis_slow_loop
 12           0 SETUP_LOOP              83 (to 86)
              3 LOAD_FAST                1 (words)
              6 GET_ITER
        >>    7 FOR_ITER                75 (to 85)
             10 STORE_FAST               2 (word)

 13          13 SETUP_EXCEPT            28 (to 44)

 14          16 LOAD_FAST                0 (self)
             19 LOAD_ATTR                0 (by_letter)
             22 LOAD_FAST 2 (word)
             25 LOAD_CONST 1 (0)
             28 BINARY_SUBSCR
             29 BINARY_SUBSCR
             30 LOAD_ATTR                1 (append)
             33 LOAD_FAST                2 (word)
             36 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             39 POP_TOP
             40 POP_BLOCK
             41 JUMP_ABSOLUTE            7

 15      >>  44 DUP_TOP
             45 LOAD_GLOBAL              2 (KeyError)
             48 COMPARE_OP              10 (exception match)
             51 POP_JUMP_IF_FALSE       81
             54 POP_TOP
             55 POP_TOP
             56 POP_TOP
```

```
 16          57 LOAD_FAST                2 (word)
             60 BUILD_LIST               1
             63 LOAD_FAST                0 (self)
             66 LOAD_ATTR                0 (by_letter)
             69 LOAD_FAST                2 (word)
             72 LOAD_CONST               1 (0)
             75 BINARY_SUBSCR
             76 STORE_SUBSCR
             77 POP_EXCEPT
             78 JUMP_ABSOLUTE            7
        >>   81 END_FINALLY
             82 JUMP_ABSOLUTE            7
        >>   85 POP_BLOCK
        >>   86 LOAD_CONST               0 (None)
             89 RETURN_VALUE

TIME: 0.0568
```

Как следует из этих результатов, для загрузки 235886 слов в экземпляр словаря `/usr/share/dict/words` в системе OS X программе `dis_slow_loop.py` потребовалось 0,0568 секунды. Это не так уж плохо, но, как показало дизассемблирование, цикл выполняет лишнюю работу. При входе в цикл в опкоде 13 программа настраивает контекст исключения (`SETUP_EXCEPT`). Затем для поиска элемента `self.by_letter[word[0]]` используются шесть опкодов, прежде чем слово присоединится к списку. Если причиной возникновения исключения является тот факт, что элемент `word[0]` еще не находится в словаре, то обработчик исключений проделывает ту же самую работу, чтобы определить `word[0]` (три опкода) и задать в переменной `self.by_letter[word[0]]` новый список, содержащий данное слово.

Один из способов, позволяющих избавиться от настройки исключения, заключается в предварительном заполнении атрибута `self.by_letter` списками, по одному для каждой буквы алфавита. Это означает, что поиск списка для нового слова всегда будет успешным и завершаться сохранением слова.

Листинг 18.29. `dis_faster_loop.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4 import string
 5
 6
 7 class Dictionary:
 8
 9     def __init__(self, words):
10         self.by_letter = {
11             letter: []
12             for letter in string.ascii_letters
13         }
14         self.load_data(words)
15
16     def load_data(self, words):
```

```
17          for word in words:
18              self.by_letter[word[0]].append(word)
```

В результате внесения изменений количество соответствующих опкодов удается уменьшить наполовину, но результирующее время снижается всего лишь до 0,0567 секунды. Очевидно, что обработка исключений вносит дополнительные накладные расходы, хотя они не очень велики.

```
$ python3 dis_test_loop.py dis_faster_loop

 17           0 SETUP_LOOP              38 (to 41)
              3 LOAD_FAST 1 (words)
              6 GET_ITER
        >>    7 FOR_ITER                30 (to 40)
             10 STORE_FAST               2 (word)

 18          13 LOAD_FAST                0 (self)
             16 LOAD_ATTR                0 (by_letter)
             19 LOAD_FAST                2 (word)
             22 LOAD_CONST               1 (0)
             25 BINARY_SUBSCR
             26 BINARY_SUBSCR
             27 LOAD_ATTR                1 (append)
             30 LOAD_FAST                2 (word)
             33 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             36 POP_TOP
             37 JUMP_ABSOLUTE            7
        >>   40 POP_BLOCK
        >>   41 LOAD_CONST               0 (None)
             44 RETURN_VALUE

TIME: 0.0567
```

Производительность можно дополнительно повысить, если вынести поиск для `self.by_letter` за пределы цикла.

Листинг 18.30. `dis_fastest_loop.py`

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4 import collections
 5
 6
 7 class Dictionary:
 8
 9     def __init__(self, words):
10         self.by_letter = collections.defaultdict(list)
11         self.load_data(words)
12
13     def load_data(self, words):
14         by_letter = self.by_letter
```

```
15          for word in words:
16              by_letter[word[0]].append(word)
```

Теперь опкоды 0–6 определяют значение атрибута `self.by_letter` и сохраняют его в локальной переменной `by_letter`. Использование локальной переменной требует всего лишь одного опкода вместо двух (инструкция 22 использует команду `LOAD_FAST` для помещения словаря в стек). После внесения этого изменения время выполнения программы сократилось до 0,0473 секунды.

```
$ python3 dis_test_loop.py dis_fastest_loop

 14           0 LOAD_FAST                0 (self)
              3 LOAD_ATTR                0 (by_letter)
              6 STORE_FAST               2 (by_letter)

 15           9 SETUP_LOOP              35 (to 47)
             12 LOAD_FAST                1 (words)
             15 GET_ITER
        >>   16 FOR_ITER                27 (to 46)
             19 STORE_FAST               3 (word)

 16 22 LOAD_FAST                         2 (by_letter)
             25 LOAD_FAST                3 (word)
             28 LOAD_CONST               1 (0)
             31 BINARY_SUBSCR
             32 BINARY_SUBSCR
             33 LOAD_ATTR                1 (append)
             36 LOAD_FAST                3 (word)
             39 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             42 POP_TOP
             43 JUMP_ABSOLUTE           16
        >>   46 POP_BLOCK
        >>   47 LOAD_CONST               0 (None)

             50 RETURN_VALUE
TIME: 0.0473
```

Дополнительная оптимизация, предложенная Брэндоном Роудсом, заключается в том, чтобы полностью исключить из этого кода Python-версию цикла `for`. Использование функции `itertools.groupby()` для упорядочения входных данных позволяет вынести итерации в C. Этот переход безопасен, поскольку известно, что входные данные поступают отсортированными. Если бы это было не так, то программа должна была бы предварительно сортировать их.

Листинг 18.31. `dis_eliminate_loop.py`

```
#!/usr/bin/env python3
# encoding: utf-8

import operator
import itertools

```

```
 7
 8 class Dictionary:
 9
10     def __init__(self, words):
11         self.by_letter = {}
12         self.load_data(words)
13
14     def load_data(self, words):
15         # Упорядочить слова
16         grouped = itertools.groupby(
17             words,
18             key=operator.itemgetter(0),
19         )
20         # Сохранить упорядоченные наборы слов
21         self.by_letter = {
22             group[0][0]: group
23             for group in grouped
24         }
```

Версия, использующая модуль `itertools`, выполняется всего лишь 0,0332 секунды, что составляет приблизительно 60% от времени выполнения первоначального варианта программы.

```
$ python3 dis_test_loop.py dis_eliminate_loop

 16           0 LOAD_GLOBAL              0 (itertools)
              3 LOAD_ATTR                1 (groupby)

 17           6 LOAD_FAST                1 (words)
              9 LOAD_CONST               1 ('key')

 18          12 LOAD_GLOBAL              2 (operator)
             15 LOAD_ATTR                3 (itemgetter)
             18 LOAD_CONST               2 (0)
             21 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             24 CALL_FUNCTION          257 (1 positional, 1
keyword pair)
             27 STORE_FAST               2 (grouped)

 21          30 LOAD_CONST               3 (<code object
<dictcomp> at 0x101517930, file ".../dis_eliminate_loop.py",
line 21>)
             33 LOAD_CONST               4
('Dictionary.load_data.<locals>.<dictcomp>')
36 MAKE_FUNCTION 0

 23          39 LOAD_FAST                2 (grouped)
             42 GET_ITER
             43 CALL_FUNCTION            1 (1 positional, 0
keyword pair)
             46 LOAD_FAST                0 (self)
             49 STORE_ATTR               4 (by_letter)
```

```
              52 LOAD_CONST               0 (None)
              55 RETURN_VALUE

TIME: 0.0332
```

18.3.7. Оптимизация, выполняемая компилятором

Дизассемблирование скомпилированного исходного кода предоставляет возможность ознакомиться с некоторыми видами оптимизации, выполняемой компилятором. Например, литеральные выражения по возможности свертываются в процессе компиляции.

Листинг 18.32. dis_constant_folding.py

```
 1 #!/usr/bin/env python3
 2 # encoding: utf-8
 3
 4 # Свертываются
 5 i = 1 + 2
 6 f = 3.4 * 5.6
 7 s = 'Hello,' + ' World!'
 8
 9 # Не свертываются
10 I = i * 3 * 4
11 F = f / 2 / 3
12 S = s + '\n' + 'Fantastic!'
```

Ни одно из значений в выражениях в строках 5–7 не может изменить способ выполнения операции, поэтому результат каждого из выражений может быть вычислен на стадии компиляции и свернут в одну инструкцию LOAD_CONST. В отличие от этого выражения в строках 10–12 включают переменные, а поскольку переменная может ссылаться на объект, перегружающий оператор, вычисление выражения должно быть отложено до стадии выполнения.

```
$ python3 -m dis dis_constant_folding.py

  5           0 LOAD_CONST              11 (3)
              3 STORE_NAME               0 (i)

  6           6 LOAD_CONST              12 (19.04)
              9 STORE_NAME               1 (f)

  7          12 LOAD_CONST              13 ('Hello, World!')
             15 STORE_NAME               2 (s)

 10          18 LOAD_NAME                0 (i)
             21 LOAD_CONST               6 (3)
             24 BINARY_MULTIPLY
             25 LOAD_CONST               7 (4)
             28 BINARY_MULTIPLY
             29 STORE_NAME               3 (I)
```

```
11          32 LOAD_NAME                1 (f)
            35 LOAD_CONST               1 (2)
            38 BINARY_TRUE_DIVIDE
            39 LOAD_CONST               6 (3)
            42 BINARY_TRUE_DIVIDE
            43 STORE_NAME               4 (F)

12          46 LOAD_NAME                2 (s)
            49 LOAD_CONST               8 ('\n')
            52 BINARY_ADD
            53 LOAD_CONST               9 ('Fantastic!')
            56 BINARY_ADD
            57 STORE_NAME               5 (S)
            60 LOAD_CONST              10 (None)
            63 RETURN_VALUE
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `dis`[9]. Содержит список инструкций байт-кода[10].
- *Include/opcode.h*. Файл исходного кода для интерпретатора CPython, содержащий инструкции байт-кода.
- *Python Essential Reference, Fourth Edition*, by David M. Beazley.
- Сайт `thomas.apestaart.org`: *Python Disassembly*[11]. Краткое обсуждение различий между версиями Python 2.5 и 2.6 в отношении сохранения значений в словарях.
- *Why is looping over range() in Python faster than using a while loop?*[12] Приведенный на сайте StackOverflow сравнительный анализ двух примеров использования циклов с привлечением их дизассемблированных байт-кодов.
- *Decorator for Binding Constants at Compile Time (Python Recipe)* (Raymond Hettinger, Skip Montanaro)[13]. Декоратор функций, переписывающий байт-код функции посредством вставки констант, чтобы избежать поиска имен во время выполнения.

18.4. `inspect`: инспектирование активных объектов

Модуль `inspect` предоставляет функции, позволяющие получать сведения об активных объектах, включая модули, классы, экземпляры, функции и методы. Функции, входящие в состав этого модуля, могут применяться для извлечения оригинального исходного кода функции, просмотра аргументов метода в стеке и извлечения информации, которую можно использовать для создания библиотечной документации исходного кода.

---

[9] `https://docs.python.org/3.5/library/dis.html`

[10] `https://docs.python.org/3.5/library/dis.html#python-bytecode-instructions`

[11] `http://thomas.apestaart.org/log/?p=927`

[12] `http://stackoverflow.com/questions/869229/why-is-looping-over-range-in-python-fasterthan-using-a-while-loop`

[13] `http://code.activestate.com/recipes/277940/`

18.4.1. Образец модуля для примеров

В примерах из этого раздела используется файл модуля `example.py`.

Листинг 18.33. `example.py`

```
# This comment appears first
# and spans 2 lines.

# This comment does not show up in the output of getcomments().

"""Sample file to serve as the basis for inspect examples.
"""


def module_level_function(arg1, arg2='default', *args, **kwargs):
    """This function is declared in the module."""
    local_variable = arg1 * 2
    return local_variable


class A(object):
    """The A class."""

    def __init__(self, name):
        self.name = name

    def get_name(self):
        "Returns the name of the instance."
        return self.name


instance_of_a = A('sample_instance')


class B(A):
    """This is the B class.
    It is derived from A.
    """

    # This method is not part of A.
    def do_something(self):
        """Does some work"""

    def get_name(self):
        "Overrides version from A"
        return 'B(' + self.name + ')'
```

18.4.2. Инспектирование модулей

Первым типом инспектирования, который мы рассмотрим, будет сбор информации об активных объектах. Функция `getmembers()` позволяет получать сведения об атрибутах объектов. Тип элементов, которые могут возвращаться, зависит

от типа сканируемого объекта. Модули могут содержать классы и функции, классы могут содержать методы и атрибуты и т.д.

Аргументами функции `getmembers()` являются инспектируемый объект (модуль, класс или экземпляр) и необязательная функция-предикат, используемая для фильтрации возвращаемых объектов. Возвращаемым значением является список кортежей, содержащих два значения: имя и тип элемента. Модуль `inspect` включает несколько функций-предикатов с именами наподобие `ismodule()`, `isclass()` и т.п.

Листинг 18.34. `inspect_getmembers_module.py`

```
import inspect

import example

for name, data in inspect.getmembers(example):
    if name.startswith('__'):
        continue
    print('{} : {!r}'.format(name, data))
```

В этом примере программа выводит элементы модуля `example`. Модули имеют несколько закрытых атрибутов, которые используются в качестве части реализации импорта, а также набор `__builtins__`. Все они игнорируются в выводе данного примера, поскольку не являются частью собственно инспектируемого модуля, а их список довольно длинный.

```
$ python3 inspect_getmembers_module.py
A : <class 'example.A'>
B : <class 'example.B'>
instance_of_a : <example.A object at 0x1014814a8>
module_level_function : <function module_level_function at
0x10148bc80>
```

Аргумент `predicate` можно использовать для фильтрации возвращаемых объектов.

Листинг 18.35. `inspect_getmembers_module_class.py`

```
import inspect

import example

for name, data in inspect.getmembers(example, inspect.isclass):
    print('{} : {!r}'.format(name, data))
```

Теперь в вывод включены лишь классы.

```
$ python3 inspect_getmembers_module_class.py

A : <class 'example.A'>
B : <class 'example.B'>
```

18.4.3. Инспектирование классов

Функция `getmembers()` позволяет инспектировать классы точно так же, как и модули, хотя типы элементов в этом случае будут другими.

Листинг 18.36. `inspect_getmembers_class.py`

```
import inspect
from pprint import pprint

import example

pprint(inspect.getmembers(example.A), width=65)
```

Поскольку фильтрация не применяется, в выводе отображаются атрибуты, методы и другие члены класса.

```
$ python3 inspect_getmembers_class.py

[('__class__', <class 'type'>),
 ('__delattr__',
  <slot wrapper '__delattr__' of 'object' objects>),
 ('__dict__',
  mappingproxy({'__dict__': <attribute '__dict__' of 'A'
objects>,
                '__doc__': 'The A class.',
                '__init__': <function A.__init__ at
0x101c99510>,
                '__module__': 'example',
                '__weakref__': <attribute '__weakref__' of 'A'
objects>,
                'get_name': <function A.get_name at
0x101c99598>})),
 ('__dir__', <method '__dir__' of 'object' objects>),
 ('__doc__', 'The A class.'),
 ('__eq__', <slot wrapper '__eq__' of 'object' objects>),
 ('__format__', <method '__format__' of 'object' objects>),
 ('__ge__', <slot wrapper '__ge__' of 'object' objects>),
 ('__getattribute__',
  <slot wrapper '__getattribute__' of 'object' objects>),
 ('__gt__', <slot wrapper '__gt__' of 'object' objects>),
 ('__hash__', <slot wrapper '__hash__' of 'object' objects>),
 ('__init__', <function A.__init__ at 0x101c99510>),
 ('__le__', <slot wrapper '__le__' of 'object' objects>),
 ('__lt__', <slot wrapper '__lt__' of 'object' objects>),
 ('__module__', 'example'),
 ('__ne__', <slot wrapper '__ne__' of 'object' objects>),
 ('__new__',
  <built-in method __new__ of type object at 0x10022bb20>),
 ('__reduce__', <method '__reduce__' of 'object' objects>),
 ('__reduce_ex__', <method '__reduce_ex__' of 'object'
objects>),
 ('__repr__', <slot wrapper '__repr__' of 'object' objects>),
 ('__setattr__',
```

```
  <slot wrapper '__setattr__' of 'object' objects>),
 ('__sizeof__', <method '__sizeof__' of 'object' objects>),
 ('__str__', <slot wrapper '__str__' of 'object' objects>),
 ('__subclasshook__',
  <built-in method __subclasshook__ of type object at
0x10061fba8>),
 ('__weakref__', <attribute '__weakref__' of 'A' objects>),
 ('get_name', <function A.get_name at 0x101c99598>)]
```

Чтобы найти методы класса, следует использовать предикат `isfunction()`. Предикат `ismethod()` распознает только связанные методы экземпляров.

Листинг 18.37. `inspect_getmembers_class_methods.py`

```
import inspect
from pprint import pprint

import example

pprint(inspect.getmembers(example.A, inspect.isfunction))
```

Теперь возвращаются лишь несвязанные методы.

```
$ python3 inspect_getmembers_class_methods.py
[('__init__', <function A.__init__ at 0x10139d510>),
 ('get_name', <function A.get_name at 0x10139d598>)]
```

Вывод для класса `B` включает переопределенный метод `get_name()`, а также новый метод и метод `__init__()`, унаследованный от класса `A`.

Листинг 18.38. `inspect_getmembers_class_methods_b.py`

```
import inspect
from pprint import pprint

import example

pprint(inspect.getmembers(example.B, inspect.isfunction))
```

Методы, унаследованные от класса `A`, такие как `__init__()`, идентифицируются как методы класса `B`.

```
$ python3 inspect_getmembers_class_methods_b.py

[('__init__', <function A.__init__ at 0x10129d510>),
  ('do_something', <function B.do_something at 0x10129d620>),
  ('get_name', <function B.get_name at 0x10129d6a8>)]
```

18.4.4. Инспектирование экземпляров

Инспектирование экземпляров работает точно так же, как инспектирование других объектов.

Листинг 18.39. inspect_getmembers_instance.py

```
import inspect
from pprint import pprint

import example

a = example.A(name='inspect_getmembers')
pprint(inspect.getmembers(a, inspect.ismethod))
```

Предикат ismethod() распознает в образце экземпляра два связанных метода класса A.

```
$ python3 inspect_getmembers_instance.py

[('__init__', <bound method A.__init__ of <example.A object at 0
x101ab1ba8>>),
 ('get_name', <bound method A.get_name of <example.A object at 0
x101ab1ba8>>)]
```

18.4.5. Строки документирования

Чтобы извлечь строку документирования объекта, следует использовать функцию getdoc(). Возвращаемым значением является атрибут __doc__, в котором символы табуляции расширены до пробелов, а отступы унифицированы.

Листинг 18.40. inspect_getdoc.py

```
import inspect
import example

print('B.__doc__:')
print(example.B.__doc__)
print()
print('getdoc(B):')
print(inspect.getdoc(example.B))
```

Вторая строка документирования имеет отступ, если извлекается непосредственно через атрибут, но функция getdoc() смещает ее к левому полю.

```
$ python3 inspect_getdoc.py

B.__doc__:
This is the B class.
    It is derived from A.

getdoc(B):
This is the B class.
It is derived from A.
```

Также возможно извлечение не только текста самой строки документирования, но и комментариев, если доступен исходный файл, в котором реализован объект. Функция getcomments() просматривает исходный код объекта и находит комментарии, предшествующие реализации.

Листинг 18.41. inspect_getcomments_method.py

```
import inspect
import example

print(inspect.getcomments(example.B.do_something))
```

Возвращенные строки включают префикс комментария, но все ведущие пробелы удаляются.

```
$ python3 inspect_getcomments_method.py

# This method is not part of A.
```

Если функции `getcomments()` передается модуль, возвращаемым значением всегда является первый комментарий, содержащийся в данном модуле.

Листинг 18.42. inspect_getcomments_module.py

```
import inspect
import example

print(inspect.getcomments(example))
```

Смежные строки из файла примера включаются в качестве единого комментария, но вывод комментария прекращается сразу же, как только встречается пустая строка.

```
$ python3 inspect_getcomments_module.py

# This comment appears first
# and spans 2 lines.
```

18.4.6. Извлечение исходного кода

Если для модуля доступен *.py*-файл, то для извлечения исходного кода класса или метода можно использовать функции `getsource()` и `getsourcelines()`.

Листинг 18.43. inspect_getsource_class.py

```
import inspect
import example

print(inspect.getsource(example.A))
```

В случае передачи класса в качестве аргумента в вывод включаются все методы данного класса.

```
$ python3 inspect_getsource_class.py

class A(object):
    """The A class."""

    def __init__(self, name):
```

```
        self.name = name

def get_name(self):
        "Returns the name of the instance."
        return self.name
```

Чтобы извлечь исходный код конкретного метода, следует передать ссылку на него функции `getsource()`.

Листинг 18.44. `inspect_getsource_method.py`

```
import inspect
import example

print(inspect.getsource(example.A.get_name))
```

В этом случае исходные уровни отступов сохраняются.

```
$ python3 inspect_getsource_method.py

    def get_name(self):
        "Returns the name of the instance."
        return self.name
```

Чтобы извлечь из файла строки исходного кода и разбить их на отдельные строки, следует использовать функцию `getsourcelines()` вместо функции `getsource()`.

Листинг 18.45. `inspect_getsourcelines_method.py`

```
import inspect
import pprint
import example

pprint.pprint(inspect.getsourcelines(example.A.get_name))
```

Функция `getsourcelines()` возвращает кортеж, содержащий список строк (строки из файла с исходным кодом) и номер строки в файле, с которой начинается исходный код.

```
$ python3 inspect_getsourcelines_method.py

(['    def get_name(self):\n',
  '        "Returns the name of the instance."\n',
  '        return self.name\n'],
 23)
```

В случае недоступности файла с исходным кодом функции `getsource()` и `getsourcelines()` возбуждают исключение `IOError`.

18.4.7. Сигнатуры методов и функций

Помимо документации функции или метода можно получить полную спецификацию аргументов, которые передаются вызываемому объекту, включая их значе-

ния по умолчанию. Функция `signature()` возвращает экземпляр `Signature`, содержащий информацию об аргументах функции.

Листинг 18.46. `inspect_signature_function.py`

```
import inspect
import example

sig = inspect.signature(example.module_level_function)
print('module_level_function{}'.format(sig))

print('\nParameter details:')
for name, param in sig.parameters.items():
    if param.kind == inspect.Parameter.POSITIONAL_ONLY:
        print('  {} (positional-only)'.format(name))
    elif param.kind == inspect.Parameter.POSITIONAL_OR_KEYWORD:
        if param.default != inspect.Parameter.empty:
            print('  {}={!r}'.format(name, param.default))
        else:
            print(,  {}'.format(name))
    elif param.kind == inspect.Parameter.VAR_POSITIONAL:
        print(,  *{}'.format(name))
    elif param.kind == inspect.Parameter.KEYWORD_ONLY:
        if param.default != inspect.Parameter.empty:
            print('  {}={!r} (keyword-only)'.format(
                name, param.default))
        else:
            print('  {} (keyword-only)'.format(name))
    elif param.kind == inspect.Parameter.VAR_KEYWORD:
        print(,  **{}'.format(name))
```

Аргументы функции доступны через атрибут `parameters` экземпляра `Signature`. Этот атрибут представляет собой упорядоченный словарь, который сопоставляет имена параметров с экземплярами `Parameter`, описывающими аргументы. В данном примере первый аргумент функции, `arg1`, не имеет значения по умолчанию, тогда как второй аргумент, `arg2`, имеет.

```
$ python3 inspect_signature_function.py

module_level_function(arg1, arg2='default', *args, **kwargs)

Parameter details:
  arg1
  arg2='default'
  *args
  **kwargs
```

Созданный для функции экземпляр `Signature` может использоваться декораторами или другими функциями для проверки корректности ввода, предоставления различных значений по умолчанию, а также для других целей. Однако при попытке написания повторно используемого декоратора проверки, пригодного для работы с функциями любого типа, возникает одно затруднение: сопоставление входных аргументов с их именами может осложняться, если функция при-

нимает как позиционные, так и именованные аргументы. В подобных случаях необходимую для обработки привязок логику предоставляют методы `bind()` и `bind_partial()`. Оба метода возвращают экземпляр `BoundArguments`, который содержит аргументы, связанные с именами аргументов указанной функции.

Листинг 18.47. `inspect_signature_bind.py`

```
import inspect
import example

sig = inspect.signature(example.module_level_function)

bound = sig.bind(
    'this is arg1',
    'this is arg2',
    'this is an extra positional argument',
    extra_named_arg='value',
)

print('Arguments:')
for name, value in bound.arguments.items():
    print('{} = {!r}'.format(name, value))

print('\nCalling:')
print(example.module_level_function(*bound.args, **bound.kwargs))
```

Экземпляр `BoundArguments` имеет атрибуты `args` и `kwargs`, которые могут быть использованы для вызова функции с помощью синтаксиса, помещающего содержимое кортежа и словаря в стек в качестве аргументов.

```
$ python3 inspect_signature_bind.py

Arguments:
arg1 = 'this is arg1'
arg2 = 'this is arg2'
args = ('this is an extra positional argument',)
kwargs = {'extra_named_arg': 'value'}

Calling:
this is arg1this is arg1
```

Даже если доступны только некоторые аргументы, метод `bind_partial()` все равно создает экземпляр `BoundArgument`. Этот экземпляр может быть не вполне работоспособным до тех пор, пока не будут добавлены остальные аргументы.

Листинг 18.48. `inspect_signature_bind_partial.py`

```
import inspect
import example

sig = inspect.signature(example.module_level_function)

partial = sig.bind_partial(
    'this is arg1',
```

```
)

print('Without defaults:')
for name, value in partial.arguments.items():
    print('{} = {!r}'.format(name, value))

print('\nWith defaults:')
partial.apply_defaults()
for name, value in partial.arguments.items():
    print('{} = {!r}'.format(name, value))
```

Метод `apply_defaults()` позволяет добавить любые значения из тех, которые предусмотрены для параметров по умолчанию.

```
$ python3 inspect_signature_bind_partial.py

Without defaults:
arg1 = 'this is arg1'

With defaults:
arg1 = 'this is arg1'
arg2 = 'default'
args = ()
kwargs = {}
```

18.4.8. Иерархии классов

Модуль `inspect` включает два метода, обеспечивающих возможность работать непосредственно с иерархиями классов. Один из них, метод `getclasstree()`, создает древовидную структуру данных на основании предоставленных ему классов и их базовых классов. Каждый элемент возвращаемого им списка содержит либо кортеж, содержащий класс и его базовые классы, либо другой список, содержащий кортежи для подклассов.

Листинг 18.49. `inspect_getclasstree.py`

```
import inspect
import example


class C(example.B):
    pass


class D(C, example.A):
    pass


def print_class_tree(tree, indent=-1):
    if isinstance(tree, list):
        for node in tree:
            print_class_tree(node, indent + 1)
```

```
    else:
        print('  ' * indent, tree[0].__name__)
    return


if __name__ == '__main__':
    print('A, B, C, D:')
    print_class_tree(inspect.getclasstree(
        [example.A, example.B, C, D])
    )
```

В этом примере выводится дерево наследования для классов A, B, C и D. Класс D встречается дважды, поскольку он является наследником как класса C, так и класса A.

```
$ python3 inspect_getclasstree.py

A, B, C, D:
 object
   A
     D
     B
       C
         D
```

Если вызвать метод getclasstree() с аргументом unique, равным True, то вывод будет другим.

Листинг 18.50. inspect_getclasstree_unique.py

```
import inspect
import example
from inspect_getclasstree import *

print_class_tree(inspect.getclasstree(
    [example.A, example.B, C, D],
    unique=True,
))
```

Теперь класс D встречается в выводе только один раз.

```
$ python3 inspect_getclasstree_unique.py

object
  A
    B
      C
        D
```

18.4.9. Порядок разрешения методов

Другая функция для работы с иерархиями классов, getmro(), возвращает кортеж классов в той последовательности, в которой они просматривались бы при разрешении атрибута, наследуемого от базового класса с использованием *порядка*

разрешения методов (method resolution order – MRO). В этой последовательности каждый класс встречается только один раз.

Листинг 18.51. inspect_getmro.py

```
import inspect
import example


class C(object):
    pass


class C_First(C, example.B):
    pass


class B_First(example.B, C):
    pass


print('B_First:')
for c in inspect.getmro(B_First):
    print(,  {}'.format(c.__name__))
print()
print('C_First:')
for c in inspect.getmro(C_First):
    print(,  {}'.format(c.__name__))
```

Вывод в этом примере демонстрирует свойственную MRO стратегию "поиска в глубину" (depth-first search). Для класса `B_First` класс `A` также встречается перед классом `C` в порядке выполнения поиска, поскольку класс `B` является потомком класса `A`.

```
$ python3 inspect_getmro.py

B_First:
  B_First
  B
  A
  C
  object

C_First:
  C_First
  C
  B
  A
  object
```

18.4.10. Стек и фреймы

Кроме функций, позволяющих инспектировать объекты кода, модуль inspect содержит функции, предназначенные для инспектирования окружения во время выполнения программы. Большинство этих функций работает со стеком вызовов и воздействует на фреймы (кадры) стека. Объекты фреймов хранят текущий контекст выполнения, включая ссылки на выполняющийся код, выполняемую операцию, а также значения локальных и глобальных переменных. В типичных случаях эти данные используются для создания трассировочной информации при возбуждении исключений. Их также можно использовать для протоколирования событий или при отладке программ, поскольку из фреймов стека могут запрашиваться значения аргументов, передаваемые функциям.

Функция currentframe() возвращает объект фрейма, соответствующий фрейму стека вызывающей функции.

Листинг 18.52. inspect_currentframe.py

```
import inspect
import pprint


def recurse(limit, keyword='default', *, kwonly='must be named'):
    local_variable = '.' * limit
    keyword = 'changed value of argument'
    frame = inspect.currentframe()
    print('line {} of {}'.format(frame.f_lineno,
                                 frame.f_code.co_filename))
    print('locals:')
    pprint.pprint(frame.f_locals)
    print()
    if limit <= 0:
        return
    recurse(limit - 1)
    return local_variable

if __name__ == '__main__':
    recurse(2)
```

Значения аргументов функции recurse() включаются в словарь локальных переменных фрейма.

```
$ python3 inspect_currentframe.py

line 14 of inspect_currentframe.py
locals:
{'frame': <frame object at 0x1022a7b88>,
 'keyword': 'changed value of argument',
 'kwonly': 'must be named',
 'limit': 2,
 'local_variable': '..'}

line 14 of inspect_currentframe.py
locals:
```

```
{'frame': <frame object at 0x102016b28>,
 'keyword': 'changed value of argument',
 'kwonly': 'must be named',
 'limit': 1,
 'local_variable': '.'}

line 14 of inspect_currentframe.py
locals:
{'frame': <frame object at 0x1020176b8>,
 'keyword': 'changed value of argument',
 'kwonly': 'must be named',
 'limit': 0,
 'local_variable': ''}
```

Функция `stack()` обеспечивает доступ ко всем фреймам стека: от текущего до фрейма первого вызывающего объекта. Следующий пример аналогичен предыдущему, за исключением того, что вывод информации о стеке откладывается до тех пор, пока не будет достигнут предел глубины рекурсии.

Листинг 18.53. `inspect_stack.py`

```
import inspect
import pprint


def show_stack():
    for level in inspect.stack():
        print(,{}[{}]\n  -> {}'.format(
            level.frame.f_code.co_filename,
            level.lineno,
            level.code_context[level.index].strip(),
        ))
        pprint.pprint(level.frame.f_locals)
        print()


def recurse(limit):
    local_variable = '.' * limit
    if limit <= 0:
        show_stack()
        return
    recurse(limit - 1)
    return local_variable


if __name__ == '__main__':
    recurse(2)
```

Последняя часть вывода представляет основную программу, код которой располагается вне функции `recurse()`.

```
$ python3 inspect_stack.py

inspect_stack.py[11]
```

```
  -> for level in inspect.stack():
{'level': FrameInfo(frame=<frame object at 0x10127e5d0>,
filename='inspect_stack.py', lineno=11, function='show_stack',
code_context=[' for level in inspect.stack():\n'], index=0)}

inspect_stack.py[24]
  -> show_stack()
{'limit': 0, 'local_variable': ''}

inspect_stack.py[26]
  -> recurse(limit - 1)
{'limit': 1, 'local_variable': '.'}

inspect_stack.py[26]
 -> recurse(limit - 1)
{'limit': 2, 'local_variable': '..'}

inspect_stack.py[30]
 -> recurse(2)
{'__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': 'Inspecting the call stack.\n',
 '__file__': 'inspect_stack.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader
object at 0x1007a97f0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'inspect': <module 'inspect' from
'.../lib/python3.5/inspect.py'>,
 'pprint': <module 'pprint' from '.../lib/python3.5/pprint.py'>,
 'recurse': <function recurse at 0x1012aa400>,
 'show_stack': <function show_stack at 0x1007a6a60>}
```

Также доступны другие функции, предназначенные для создания списков фреймов в различных контекстах, например в контексте обрабатываемого исключения. Для более подробного ознакомления с функциями `trace()`, `getouterframes()` и `getinnerframes()` следует обратиться к документации.

18.4.11. Интерфейс командной строки

Модуль `inspect` также включает интерфейс командной строки, обеспечивающий получение подробной информации об объектах без написания отдельной программы с нужными вызовами. Входными данными служат имя модуля и необязательный объект, входящий в состав модуля. По умолчанию выводится исходный код объекта. Если используется аргумент `--details`, то вместо исходного кода выводятся метаданные.

```
$ python3 -m inspect -d example

Target: example
Origin: .../example.py
```

```
Cached: .../__pycache__/example.cpython-35.pyc
Loader: <_frozen_importlib_external.SourceFileLoader object at 0
x101527860>


$ python3 -m inspect -d example:A
Target: example:A
Origin: .../example.py
Cached: .../__pycache__/example.cpython-35.pyc
Line: 16

$ python3 -m inspect example:A.get_name

    def get_name(self):
        "Returns the name of the instance."
        return self.name
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `inspect`[14].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `inspect` (раздел A.6.21).
- *Python 2.3 Method Resolution Order*[15]. Документация порядка разрешения методов, используемого в версиях Python 2.3 и выше.
- `pyclbr` (раздел 16.12). Модуль `pyclbr` предоставляет доступ к части той информации, которую предоставляет и модуль `inspect`, но за счет синтаксического анализа модуля без его импортирования.
- **PEP 362**[16]. *Function Signature Object.*

[14] https://docs.python.org/3.5/library/inspect.html

[15] www.python.org/download/releases/2.3/mro/

[16] www.python.org/dev/peps/pep-0362

Глава 19

Модули и пакеты

Основной механизм расширения Python использует исходный код, хранимый в модулях и импортируемый в программу посредством инструкции `import`. Возможности, ассоциируемые большинством разработчиков с понятием "Python", на самом деле реализованы в виде коллекции модулей – *стандартной библиотеки*, которая и является предметом обсуждения в этой книге. Несмотря на то что средство импорта встроено в интерпретатор, библиотека также включает несколько модулей, связанных с импортом.

Модуль `importlib` (раздел 19.1) предоставляет базовую реализацию механизма импорта, используемую интерпретатором. Его можно использовать для динамического импортирования модулей в процессе выполнения программы вместо инструкции `import`, загружающей модули при запуске приложения. Динамическая загрузка модулей может пригодиться, когда имя импортируемого модуля неизвестно заранее, как это бывает в случае дополнений и расширений.

Наряду с исходным кодом пакеты Python могут включать вспомогательные файлы ресурсов – конфигурационные файлы, изображения и другие данные. Интерфейс доступа к файлам исходного кода способом, обеспечивающим переносимость, реализован в модуле `pkgutil` (раздел 19.2). Он также поддерживает изменение путей импорта для пакета, чтобы его содержимое можно было устанавливать в несколько каталогов, сохраняя при этом целостность пакета.

В модуле `zipimport` (раздел 19.3) предоставляется пользовательский импортер модулей и пакетов, сохраненных в ZIP-архивах. Он используется для загрузки EGG-файлов Python, но также обеспечивает удобный способ пакетирования и распространения приложений.

19.1. `importlib`: механизм импорта Python

Модуль `importlib` включает функции, которые реализуют механизм импорта Python, предназначенный для загрузки кода из пакетов и модулей. Он является единой точкой доступа к динамическому импорту модулей и в некоторых случаях может быть полезным, если во время написания кода имя модуля, который должен импортироваться, неизвестно (как это бывает при написании дополнений или расширений для приложения).

19.1.1. Пакет `example`

Во всех примерах, приведенных в этом разделе, используется пакет `example`, содержащий модуль `__init__.py`.

Листинг 19.1. `example/__init__.py`

```
print('Importing example package')
```

Кроме того, данный пакет содержит также модуль submodule.py.

Листинг 19.2. example/submodule.py

```
print('Importing submodule')
```

Вывод функции print() помогает определить, какой именно пакет или модуль импортируется в том или ином примере.

19.1.2. Типы модулей

Python поддерживает несколько разновидностей модулей. Каждый из них обрабатывается по-своему при открытии и добавлении в пространство имен, а поддержка форматов меняется от платформы к платформе. Так, на компьютерах, работающих под управлением Windows, общие библиотеки загружаются из файлов с расширениями .dll и .pyd, а не .so. Расширения для модулей C также могут быть другими, когда вместо финальной сборки используется отладочная, поскольку последняя может компилироваться с включением отладочной информации. Если библиотека расширения C или другой модуль не загружается, как ожидалось, используйте константы, определенные в модуле importlib.machinery, для выяснения того, какие типы поддерживаются для текущей платформы, а также какие параметры используются для их загрузки.

Листинг 19.3. importlib_suffixes.py

```
import importlib.machinery

SUFFIXES = [
    ('Source:', importlib.machinery.SOURCE_SUFFIXES),
    ('Debug:',
     importlib.machinery.DEBUG_BYTECODE_SUFFIXES),
    ('Optimized:',
     importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES),
    ('Bytecode:', importlib.machinery.BYTECODE_SUFFIXES),
    ('Extension:', importlib.machinery.EXTENSION_SUFFIXES),
]


def main():
    tmpl = '{:<10}  {}'
    for name, value in SUFFIXES:
        print(tmpl.format(name, value))


if __name__ == '__main__':
    main()
```

Возвращаемым значением является последовательность кортежей, содержащая расширение имени файла, режим, который можно использовать для открытия файла, содержащего модуль, и код типа из константы, определенной в модуле. Следующий вывод не является полным, поскольку некоторые из импортируемых типов модулей и пакетов не соответствуют одиночным файлам.

```
$ python3 importlib_suffixes.py

Source: ['.py']
Debug: ['.pyc']
Optimized: ['.pyc']
Bytecode: ['.pyc']
Extension: ['.cpython-35m-darwin.so', '.abi3.so', '.so']
```

19.1.3. Импортирование модулей

Высокоуровневый API, предлагаемый модулем importlib, упрощает процесс импорта модуля, заданного абсолютным или относительным именем. В случае использования относительного имени модуля пакет, который его содержит, указывается в виде отдельного аргумента.

Листинг 19.4. importlib_import_module.py

```
import importlib

m1 = importlib.import_module('example.submodule')
print(m1)

m2 = importlib.import_module('.submodule', package='example')
print(m2)

print(m1 is m2)
```

Функция import_module() возвращает объект модуля, созданный в результате импорта.

```
$ python3 importlib_import_module.py

Importing example package
Importing submodule
<module 'example.submodule' from '.../example/submodule.py'>
<module 'example.submodule' from '.../example/submodule.py'>
True
```

Если модуль не удается импортировать, функция import_module() возбуждает исключение ImportError.

Листинг 19.5. importlib_import_module_error.py

```
import importlib

try:
   importlib.import_module('example.nosuchmodule')
except ImportError as err:
    print('Error:', err)
```

Сообщение об ошибке включает имя отсутствующего модуля.

```
$ python3 importlib_import_module_error.py

Importing example package
Error: No module named 'example.nosuchmodule'
```

Для перезагрузки существующего модуля следует использовать функцию `reload()`.

Листинг 19.6. importlib_reload.py

```
import importlib

m1 = importlib.import_module('example.submodule')
print(m1)

m2 = importlib.reload(m1)
print(m1 is m2)
```

Функция `reload()` возвращает новый модуль. В зависимости от типа используемого загрузчика им может быть тот же экземпляр модуля.

```
$ python3 importlib_reload.py

Importing example package
Importing submodule
<module 'example.submodule' from '.../example/submodule.py'>
Importing submodule
True
```

19.1.4. Загрузчики

Низкоуровневый API, предлагаемый модулем `importlib`, предоставляет доступ к объектам загрузчиков (раздел 17.2.6). Чтобы получить загрузчик для модуля, используйте функцию `find_loader()`. Для извлечения модуля следует использовать метод `load_module()` загрузчика.

Листинг 19.7. importlib_find_loader.py

```
import importlib

loader = importlib.find_loader('example')
print('Loader:', loader)

m = loader.load_module()
print('Module:', m)
```

В этом примере загружается верхний уровень пакета `example`.

```
$ python3 importlib_find_loader.py

Loader: <_frozen_importlib_external.SourceFileLoader object at
0x101be0da0>
Importing example package
Module: <module 'example' from '.../example/__init__.py'>
```

Подмодули, находящиеся в пакете, должны загружаться отдельно с использованием пути из пакета. В следующем примере первым загружается пакет, а затем его путь передается функции `find_loader()` для создания загрузчика, способного загрузить подмодуль.

Листинг 19.8. `importlib_submodule.py`

```
import importlib

pkg_loader = importlib.find_loader('example')
pkg = pkg_loader.load_module()

loader = importlib.find_loader('submodule', pkg.__path__)
print('Loader:', loader)

m = loader.load_module()
print('Module:', m)
```

В отличие от функции `import_module()` имя подмодуля должно предоставляться без какого-либо префикса относительного пути, поскольку загрузчик уже будет ограничен путем к пакету.

```
$ python3 importlib_submodule.py

Importing example package
Loader: <_frozen_importlib_external.SourceFileLoader object at
0x1012e5390>
Importing submodule
Module: <module 'submodule' from '.../example/submodule.py'>
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `importlib`[1].
- Раздел 17.2.6. Перехватчики импорта, пути поиска модулей и другие аналогичные средства, содержащиеся в модуле `sys`.
- `inspect` (раздел 18.14). Загрузка информации из модуля программным способом.
- **PEP 302**[2]. *New-import hooks.*
- **PEP 369**[3]. *Post-import hooks.*
- **PEP 488**[4]. *Elimination of PYO files.*

19.2. `pkgutil`: вспомогательные функции для упаковывания программ

Модуль `pkgutil` включает функции, позволяющие изменять правила импорта пакетов Python и загружать ресурсы, не являющиеся кодом, из файлов, распространяемых в составе пакета.

[1] `https://docs.python.org/3.5/library/importlib.html`
[2] `www.python.org/dev/peps/pep-0302`
[3] `www.python.org/dev/peps/pep-0369`
[4] `www.python.org/dev/peps/pep-0488`

19.2.1. Пути импорта пакетов

Функция extend_path() используется для изменения пути поиска модулей и способа импортирования подмодулей из пакета, позволяя объединять несколько каталогов так, как если бы они представляли собой один каталог. Эту функцию можно использовать для переопределения установленных версий пакетов разрабатываемыми версиями или объединения платформозависимых и общих модулей в единое пространство имен пакета.

Наиболее типичный способ вызова функции extend_path() заключается в добавлении двух строк кода в файл __init__.py, находящийся в пакете.

```
import pkgutil
__path__ = pkgutil.extend_path(__path__, __name__)
```

Функция extend_path() просматривает список путей sys.path в поиске каталогов, включающих подкаталог, имя которого базируется на пакете, предоставленном в качестве второго аргумента. Список каталогов объединяется со значением пути, переданным в качестве первого аргумента, и в результате возвращается единый список, пригодный для использования в качестве пути импорта пакета.

Используемый в следующем примере пакет demopkg1 включает два файла: __init__.py и shared.py. Файл __init__.py в пакете demopkg1 содержит инструкции вывода, которые отображают путь поиска до и после его изменения, чтобы можно было увидеть различия между этими двумя путями.

Листинг 19.9. demopkg1/__init__.py

```
import pkgutil
import pprint

print('demopkg1.__path__ before:')
pprint.pprint(__path__)
print()

__path__ = pkgutil.extend_path(__path__, __name__)

print('demopkg1.__path__ after:')
pprint.pprint(__path__)
print()
```

Каталог extension, дополняющий возможности пакета demopkg, содержит три других исходных файла: файлы __init__.py (по одному в каждом каталоге) и файл not_shared.py.

```
$ find extension -name '*.py'

extension/__init__.py
extension/demopkg1/__init__.py
extension/demopkg1/not_shared.py
```

Следующая простая тестовая программа импортирует пакет demopkg1.

Листинг 19.10. pkgutil_extend_path.py

```
import demopkg1
print('demopkg1           :', demopkg1.__file__)

try:
    import demopkg1.shared
except Exception as err:
    print('demopkg1.shared    : Not found ({})'.format(err))
else:
    print('demopkg1.shared    :', demopkg1.shared.__file__)

try:
    import demopkg1.not_shared
except Exception as err:
    print('demopkg1.not_shared: Not found ({})'.format(err))
else:
    print('demopkg1.not_shared:', demopkg1.not_shared.__file__)
```

Если выполнить эту тестовую программу из командной строки, то модуль `not_shared` не будет найден.

Примечание

В этих примерах пути к файлам выводятся в сокращенном виде, чтобы более отчетливо выделить их изменяющиеся части.

```
$ python3 pkgutil_extend_path.py

demopkg1.__path__ before:
['.../demopkg1']

demopkg1.__path__ after:
['.../demopkg1']

demopkg1           : .../demopkg1/__init__.py
demopkg1.shared    : .../demopkg1/shared.py
demopkg1.not_shared: Not found (No module named 'demopkg1.not_sh
ared')
```

Однако, если добавить каталог `extension` в переменную `PYTHONPATH` и повторно запустить программу, то результаты будут другими.

```
$ PYTHONPATH=extension python3 pkgutil_extend_path.py
demopkg1.__path__ before:
['.../demopkg1']
demopkg1.__path__ after:
['.../demopkg1',
'.../extension/demopkg1']
demopkg1 : .../demopkg1/__init__.py
demopkg1.shared : .../demopkg1/shared.py
demopkg1.not_shared: .../extension/demopkg1/not_shared.py
```

В этом случае модуль not_shared удается найти, поскольку в список путей поиска была добавлена версия demopkg1, находящаяся в каталоге extension.

Описанный способ расширения путей удобно использовать для объединения платформозависимых версий пакетов с общими пакетами, особенно если платформозависимые версии включают модули C-расширений.

19.2.2. Разработка версий пакетов

В процессе улучшения проекта разработчику часто приходится тестировать изменения, вносимые в установленный пакет. Простая замена установленного экземпляра разрабатываемой версией – не совсем удачная идея, поскольку корректность этой версии еще должна быть подтверждена, тогда как от установленного пакета могут зависеть другие инструменты.

Полностью независимый экземпляр пакета можно сконфигурировать в среде разработки с помощью модуля virtualenv или venv (раздел 16.13). Однако в случае незначительных изменений накладные расходы, связанные с настройкой виртуальной среды, могут сделать такой подход неоправданно затратным.

Другая возможность заключается в том, чтобы использовать модуль pkgutil для изменения пути поиска модулей, принадлежащих разрабатываемому пакету. В этом случае путь должен быть изменен таким образом, чтобы разрабатываемая версия переопределила установленную.

Предположим, что пакет demopkg2 включает файлы __init__.py и overloaded.py, причем разрабатываемая версия функции находится в файле demopkg2/overloaded.py. Установленная версия содержит следующий код.

Листинг 19.11. demopkg2/overloaded.py

```
def func():
    print('This is the installed version of func().')
```

Файл demopkg2/__init__.py содержит следующий код.

Листинг 19.12. demopkg2/__init__.py

```
import pkgutil

__path__ = pkgutil.extend_path(__path__, __name__)
__path__.reverse()
```

Использование функции reverse() гарантирует, что каталоги, добавленные в путь поиска средствами модуля pkgutil, будут просматриваться в процессе поиска импортируемых модулей *до* просмотра расположений, заданных по умолчанию.

Следующая программа импортирует модуль demopkg2.overloaded и вызывает функцию func().

Листинг 19.13. pkgutil_devel.py

```
import demopkg2
print('demopkg2           :', demopkg2.__file__)

import demopkg2.overloaded
print('demopkg2.overloaded:', demopkg2.overloaded.__file__)
```

```
print()
demopkg2.overloaded.func()
```

Выполнение этой программы без какой-либо специальной обработки путей поиска дает следующий вывод из установленной версии `func()`.

```
$ python3 pkgutil_devel.py

demopkg2           : .../demopkg2/__init__.py
demopkg2.overloaded: .../demopkg2/overloaded.py

This is the installed version of func().
```

Каталог разработки содержит следующий код.

```
$ find develop/demopkg2 -name '*.py'

develop/demopkg2/__init__.py
develop/demopkg2/overloaded.py
```

Измененная версия файла содержит следующий код.

Листинг 19.14. `develop/demopkg2/overloaded.py`

```
def func():
    print('This is the development version of func().')
```

Она будет загружена в том случае, если каталог разработки включен в путь поиска модулей.

```
$ PYTHONPATH=develop python3 pkgutil_devel.py

demopkg2 : .../demopkg2/__init__.py
demopkg2.overloaded: .../develop/demopkg2/overloaded.py

This is the development version of func().
```

19.2.3. Управление путями с помощью PKG-файлов

Первый пример иллюстрирует расширение списка путей поиска за счет дополнительных каталогов, включаемых с помощью константы `PYTHONPATH`. Для расширения списка путей поиска можно также использовать файлы **.pkg*, содержащие имена каталогов. Эти файлы аналогичны *.pth*-файлам, которые используются модулем `site` (см. раздел 17.1). Они содержат имена каталогов, по одному на строку, присоединяемых к пути поиска для пакета.

Другим способом структурирования платформозависимых частей приложения из первого примера является использование отдельного каталога для каждой операционной системы и включение *.pkg*-файла для расширения списка путей поиска.

В приведенном ниже примере используются те же файлы `demopkg1` и дополнительно включаются следующие файлы.

```
$ find os_* -type f

os_one/demopkg1/__init__.py
os_one/demopkg1/not_shared.py
os_one/demopkg1.pkg
os_two/demopkg1/__init__.py
os_two/demopkg1/not_shared.py
os_two/demopkg1.pkg
```

Здесь *.pkg*-файлам присвоены имена *demopkg1.pkg*, чтобы они соответствовали расширяемому пакету. Оба они содержат одну строку:

```
demopkg
```

Приведенная ниже демонстрационная программа отображает версию импортируемого модуля.

Листинг 19.15. pkgutil_os_specific.py

```
import demopkg1
print('demopkg1:', demopkg1.__file__)

import demopkg1.shared
print('demopkg1.shared:', demopkg1.shared.__file__)

import demopkg1.not_shared
print('demopkg1.not_shared:', demopkg1.not_shared.__file__)
```

Для переключения между двумя пакетами можно использовать простой сценарий.

Листинг 19.16. with_os.sh

```
#!/bin/sh

export PYTHONPATH=os_${1}
echo "PYTHONPATH=$PYTHONPATH"
echo

python3 pkgutil_os_specific.py
```

Запуск этого сценария с использованием строки "one" или "two" в качестве аргумента приводит к соответствующему изменению пути.

```
$ ./with_os.sh one

PYTHONPATH=os_one

demopkg1.__path__ before:
['.../demopkg1']

demopkg1.__path__ after:
['.../demopkg1',
 '.../os_one/demopkg1',
```

```
 'demopkg']

demopkg1: .../demopkg1/__init__.py
demopkg1.shared: .../demopkg1/shared.py
demopkg1.not_shared: .../os_one/demopkg1/not_shared.py

$ ./with_os.sh two

PYTHONPATH=os_two

demopkg1.__path__ before:
['.../demopkg1']
demopkg1.__path__ after:
['.../demopkg1',
 '.../os_two/demopkg1',
 'demopkg']

demopkg1: .../demopkg1/__init__.py
demopkg1.shared: .../demopkg1/shared.py
demopkg1.not_shared: .../os_two/demopkg1/not_shared.py
```

PKG-файлы могут встретиться где угодно в обычном пути поиска, поэтому для включения дерева разработки можно было бы также использовать единственный PKG-файл в текущем рабочем каталоге.

19.2.4. Вложенные пакеты

В случае вложенных пакетов в изменении нуждается только путь к пакету верхнего уровня. В качестве примера рассмотрим приведенную ниже структуру каталогов.

```
$ find nested -name '*.py'

nested/__init__.py
nested/second/__init__.py
nested/second/deep.py
nested/shallow.py
```

Здесь файл `nested/__init__.py` содержит следующий код.

Листинг 19.17. `nested/__init__.py`

```
import pkgutil

__path__ = pkgutil.extend_path(__path__, __name__)
__path__.reverse()
```

Дерево разработки выглядит так.

```
$ find develop/nested -name '*.py'

develop/nested/__init__.py
develop/nested/second/__init__.py
develop/nested/second/deep.py
develop/nested/shallow.py
```

Каждый из модулей shallow и deep содержит простую функцию, которая выводит сообщение, указывающее на то, какую версию представляет данная функция: установленную или разрабатываемую. Следующая тестовая программа испытывает новые пакеты.

Листинг 19.18. pkgutil_nested.py

```
import nested

import nested.shallow
print('nested.shallow:', nested.shallow.__file__)
nested.shallow.func()

print()
import nested.second.deep
print('nested.second.deep:', nested.second.deep.__file__)
nested.second.deep.func()
```

При запуске сценария pkgutil_nested.py без каких-либо манипуляций с путями оба модуля используют установленные версии.

```
$ python3 pkgutil_nested.py

nested.shallow: .../nested/shallow.py
This func() comes from the installed version of nested.shallow

nested.second.deep: .../nested/second/deep.py
This func() comes from the installed version of nested.second.de
ep
```

После добавления в список путей поиска каталога develop разрабатываемые версии обеих функций перекрывают установленные.

```
$ PYTHONPATH=develop python3 pkgutil_nested.py

nested.shallow: .../develop/nested/shallow.py
This func() comes from the development version of nested.shallow

nested.second.deep: .../develop/nested/second/deep.py
This func() comes from the development version of nested.second.
deep
```

19.2.5. Пакетные данные

Помимо кода пакеты Python могут содержать файлы данных — шаблоны, конфигурационные файлы, изображения и другие вспомогательные файлы, используемые кодом пакетов. Функция get_data() предоставляет доступ к данным, обеспечивая распознавание файлов разных форматов, в связи с чем не имеет значения, распространяется ли пакет в виде EGG-файла, замороженного двоичного файла или в виде обычных файлов файловой системы.

Предположим, пакет pkgwithdata содержит каталог templates.

```
$ find pkgwithdata -type f

pkgwithdata/__init__.py
pkgwithdata/templates/base.html
```

В файле pkgwithdata/templates/base.html содержится простой HTML-шаблон.

Листинг 19.19. pkgwithdata/templates/base.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>

<p>This is a sample data file.</p>

</body>
</html>
```

В следующей программе функция get_data() используется для извлечения и вывода содержимого шаблона.

Листинг 19.20. pkgutil_get_data.py

```
import pkgutil

template = pkgutil.get_data('pkgwithdata', 'templates/base.html')
print(template.decode('utf-8'))
```

Аргументами функции get_data() являются имя пакета, заданное с использованием точечной нотации, и имя файла, заданное относительно верхнего уровня пакета. Функция возвращает байтовую последовательность, поэтому перед выводом на печать она декодируется из формата UTF-8.

```
$ python3 pkgutil_get_data.py

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>

<p>This is a sample data file.</p>

</body>
</html>
```

Функция get_data() распознает формат дистрибутивов, поскольку использует для доступа к содержимому пакетов перехватчики импорта (расширения), определенные в документе **PEP 302**. Можно использовать любой загрузчик, который предоставляет такие расширения, включая импортер ZIP-архивов из модуля zipfile (раздел 8.5).

Листинг 19.21. pkgutil_get_data_zip.py

```
import pkgutil
import zipfile
import sys

# Создать ZIP-файл с кодом из текущего каталога и шаблон,
# используя имя, которое не встречается в локальной файловой
# системе
with zipfile.PyZipFile('pkgwithdatainzip.zip', mode='w') as zf:
    zf.writepy('.')
    zf.write('pkgwithdata/templates/base.html',
             'pkgwithdata/templates/fromzip.html',
             )

# Добавить ZIP-файл в путь импорта
sys.path.insert(0, 'pkgwithdatainzip.zip')

# Импортировать пакет pkgwithdata, чтобы продемонстрировать,
# что он взят из ZIP-архива
import pkgwithdata
print('Loading pkgwithdata from', pkgwithdata.__file__)
# Вывести на экран тело шаблона
print('\nTemplate:')
data = pkgutil.get_data('pkgwithdata', 'templates/fromzip.html')
print(data.decode('utf-8'))
```

В этом примере с помощью метода PyZipFile.writepy() создается ZIP-архив, содержащий копию пакета pkgwithdata, которая включает переименованную версию файла шаблона. Затем этот архив добавляется в путь импорта, после чего шаблон загружается с помощью модуля pkgutil и выводится на экран. Более подробно метод writepy() обсуждался при рассмотрении модуля zipfile (см. раздел 8.2).

```
$ python3 pkgutil_get_data_zip.py

Loading pkgwithdata from
pkgwithdatainzip.zip/pkgwithdata/__init__.pyc

Template:
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>
```

```
<p>This is a sample data file.</p>

</body>
</html>
```

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `pkgutil`[5].
- `virtualenv` (Ian Bicking)[6]. Сценарий виртуальной среды.
- `distutils`. Предлагаемые стандартной библиотекой Python инструменты для создания пакетов.
- `setuptools`[7]. Следующее поколение инструментов, предназначенных для создания пакетов.
- **PEP 302**[8]. *Import Hooks*.
- `zipfile` (раздел 8.2). Создание ZIP-архивов, пригодных для импорта.
- `zipimport` (раздел 19.3). Средство импортирования пакетов из ZIP-архивов.

19.3. `zipimport`: загрузка кода Python из ZIP-архивов

Модуль `zipimport` реализует класс `zipimporter`, который можно использовать для поиска и загрузки модулей Python из ZIP-архивов. Класс `zipimporter` реализует API расширений (перехватчиков импорта), специфицированный в документе **PEP 302**; именно так работает *.egg*-формат Python.

Необходимость в явном использовании модуля `zipimport` возникает лишь в редких случаях, поскольку модули можно импортировать непосредственно из ZIP-архива, включив его в список путей `sys.path`. Тем не менее знакомство с API импортера поможет программисту изучить доступные возможности и понять, как работает модуль `import`. Знание того, как работает ZIP-импортер, может также пригодиться при отладке распространения пакетов приложений в виде ZIP-архивов, созданных средствами класса `zipfile.PyZipFile`.

19.3.1. Пример

В следующих примерах для создания образца ZIP-архива, содержащего несколько модулей `Python`, используется код, который приводился при обсуждении модуля `zipfile` (см. раздел 8.2).

Листинг 19.22. `zipimport_make_example.py`

```
import sys
import zipfile
```

[5] https://docs.python.org/3.5/library/pkgutil.html
[6] http://pypi.python.org/pypi/virtualenv
[7] https://setuptools.readthedocs.io/en/latest/
[8] www.python.org/dev/peps/pep-0302

```
if __name__ == '__main__':
    zf = zipfile.PyZipFile('zipimport_example.zip', mode='w')
    try:
        zf.writepy('.')
        zf.write('zipimport_get_source.py')
        zf.write('example_package/README.txt')
    finally:
        zf.close()
    for name in zf.namelist():
        print(name)
```

Прежде чем приступить к выполнению остальных примеров, выполните сценарий `zipimport_make_example.py` для создания ZIP-архива всех модулей, содержащихся в каталоге примеров, и необходимых тестовых данных к ним.

```
$ python3 zipimport_make_example.py

__init__.pyc
example_package/__init__.pyc
zipimport_find_module.pyc
zipimport_get_code.pyc
zipimport_get_data.pyc
zipimport_get_data_nozip.pyc
zipimport_get_data_zip.pyc
zipimport_get_source.pyc
zipimport_is_package.pyc
zipimport_load_module.pyc
zipimport_make_example.pyc
zipimport_get_source.py
example_package/README.txt
```

19.3.2. Поиск модуля

При условии, что предоставлено полное имя модуля, метод `find_module()` пытается найти этот модуль в ZIP-архиве.

Листинг 19.23. `zipimport_find_module.py`

```
import zipimport

importer = zipimport.zipimporter('zipimport_example.zip')

for module_name in ['zipimport_find_module', 'not_there']:
    print(module_name, ':', importer.find_module(module_name))
```

Если модуль успешно найден, возвращается экземпляр `zipimporter`. В противном случае возвращается значение `None`.

```
$ python3 zipimport_find_module.py

zipimport_find_module : <zipimporter object
"zipimport_example.zip">
not_there : None
```

19.3.3. Доступ к коду

Метод `get_code()` загружает объект кода для модуля, импортируемого из архива.

Листинг 19.24. `zipimport_get_code.py`

```
import zipimport

importer = zipimport.zipimporter('zipimport_example.zip')
code = importer.get_code('zipimport_get_code')
print(code)
```

Объект кода — это не то же самое, что объект модуля, но используется для создания последнего.

```
$ python3 zipimport_get_code.py

<code object <module> at 0x1012b4ae0, file
"./zipimport_get_code.py", line 6>
```

Чтобы загрузить код в качестве модуля, пригодного для использования, следует вызвать метод `load_module()`.

Листинг 19.25. `zipimport_load_module.py`

```
import zipimport

importer = zipimport.zipimporter('zipimport_example.zip')
module = importer.load_module('zipimport_get_code')
print('Name :', module.__name__)
print('Loader :', module.__loader__)
print('Code :', module.code)
```

Результат представляет собой объект модуля, сконфигурированный так, как если бы код был загружен средствами обычного импорта.

```
$ python3 zipimport_load_module.py

<code object <module> at 0x1007b4c00, file
"./zipimport_get_code.py", line 6>
Name : zipimport_get_code
Loader : <zipimporter object "zipimport_example.zip">
Code : <code object <module> at 0x1007b4c00, file
"./zipimport_get_code.py", line 6>
```

19.3.4. Исходный код

Как и в случае модуля `inspect` (см. раздел 18.4), существует возможность извлечения исходного кода модуля из ZIP-архива с помощью модуля `zipimport`, если архив включает этот исходный код. В следующем примере в файл `zipimport_example.zip` добавляется лишь исходный файл `zipimport_get_source.py`. Все остальные модули добавляются только в виде *.pyc*-файлов.

Листинг 19.26. zipimport_get_source.py

```
import zipimport

modules = [
    'zipimport_get_code',
    'zipimport_get_source',
]

importer = zipimport.zipimporter('zipimport_example.zip')
for module_name in modules:
    source = importer.get_source(module_name)
    print('=' * 80)
    print(module_name)
    print('=' * 80)
    print(source)
    print()
```

Если исходный код для модуля недоступен, метод `get_source()` возвращает значение `None`.

```
$ python3 zipimport_get_source.py
================================================================================
zipimport_get_code
================================================================================
None

================================================================================
zipimport_get_source
================================================================================
#!/usr/bin/env python3
#
# Copyright 2007 Doug Hellmann.
#
"""Retrieving the source code for a module within a zip archive.
"""

#end_pymotw_header
import zipimport

modules = [
    'zipimport_get_code',
    'zipimport_get_source',
]

importer = zipimport.zipimporter('zipimport_example.zip')
for module_name in modules:
    source = importer.get_source(module_name)
    print('=' * 80)
    print(module_name)
    print('=' * 80)
    print(source)
    print()
```

19.3.5. Пакеты

Чтобы определить, ссылается ли имя на пакет, а не на модуль, следует использовать метод `is_package()`.

Листинг 19.27. `zipimport_is_package.py`

```
import zipimport

importer = zipimport.zipimporter('zipimport_example.zip')
for name in ['zipimport_is_package', 'example_package']:
    print(name, importer.is_package(name))
```

В данном случае имени `zipimport_is_package` соответствует модуль, а имени `example_package` — пакет.

```
$ python3 zipimport_is_package.py

zipimport_is_package False
example_package True
```

19.3.6. Данные

Иногда исходные модули или пакеты должны распространяться вместе с данными, не являющимися кодом. Изображения, конфигурационные файлы, данные для использования по умолчанию, тестовые данные — это лишь небольшой перечень возможных типов таких данных. Во многих случаях для определения пути к таким файлам относительно места установки кода можно воспользоваться атрибутами `__path__` и `__file__` модуля.

Например, в случае обычного модуля путь в файловой системе можно сконструировать на основе атрибута `__file__` импортированного пакета, как показано в приведенном ниже коде.

Листинг 19.28. `zipimport_get_data_nozip.py`

```
import os
import example_package

# Найти каталог, содержащий импортируемый пакет, и создать
# на его основе имя файла данных
pkg_dir = os.path.dirname(example_package.__file__)
data_filename = os.path.join(pkg_dir, 'README.txt')

# Прочитать файл и отобразить его содержимое
print(data_filename, ':')
print(open(data_filename, 'r').read())
```

Вывод зависит от того, где именно в файловой системе располагается образец кода.

```
$ python3 zipimport_get_data_nozip.py

.../example_package/README.txt :
```

```
This file represents sample data which could be embedded in the
ZIP archive. You could include a configuration file, images, or
any other sort of noncode data.
```

Если пакет `example_package` импортируется из ZIP-архива, а не из файловой системы, то использование атрибута `__file__` не приведет к нужному результату.

Листинг 19.29. zipimport_get_data_zip.py

```
import sys
sys.path.insert(0, 'zipimport_example.zip')

import os
import example_package
print(example_package.__file__)
data_filename = os.path.join(
    os.path.dirname(example_package.__file__),
    'README.txt',
)
print(data_filename, ':')
print(open(data_filename, 'rt').read())
```

Атрибут `__file__` пакета ссылается на ZIP-архив, а не на каталог, поэтому конструирование пути к файлу *README.txt* приводит к неверному значению.

```
$ python3 zipimport_get_data_zip.py

zipimport_example.zip/example_package/__init__.pyc
zipimport_example.zip/example_package/README.txt :
Traceback (most recent call last):
  File "zipimport_get_data_zip.py", line 20, in <module>
    print(open(data_filename, 'rt').read())
NotADirectoryError: [Errno 20] Not a directory:
'zipimport_example.zip/example_package/README.txt'
```

Более надежный способ извлечения файла предоставляет метод `get_data()`. Доступ к экземпляру `zipimporter`, загрузившему модуль, обеспечивает атрибут `__loader__` импортированного модуля.

Листинг 19.30. zipimport_get_data.py

```
import sys
sys.path.insert(0, 'zipimport_example.zip')

import os
import example_package
print(example_package.__file__)
data = example_package.__loader__.get_data(
    'example_package/README.txt')
print(data.decode('utf-8'))
```

Функция `pkgutil.get_data()` использует этот интерфейс для доступа к данным, содержащимся в пакете. Извлеченное значение представляет собой байто-

вую строку, которая должна быть декодирована в строку Unicode перед выводом на экран.

```
$ python3 zipimport_get_data.py

zipimport_example.zip/example_package/__init__.pyc
This file represents sample data which could be embedded in the
ZIP archive. You could include a configuration file, images, or
any other sort of noncode data.
```

Метод `__loader__` не устанавливается для модулей, не импортированных посредством модуля `zipimport`.

Дополнительные ссылки

- Раздел документации стандартной библиотеки, посвященный модулю `zipimport`[9].
- Замечания относительно портирования программ из Python 2 в Python 3, касающиеся модуля `zipimport` (раздел A.6.52).
- `imp`. Другие функции, связанные с импортом.
- `pkgutil` (раздел 19.2). Предоставляет более общий интерфейс для функции `get_data()`.
- `zipfile` (раздел 8.5). Чтение и запись файлов ZIP-архивов.
- **PEP 302**[10]. *New Import Hooks.*

[9] https://docs.python.org/3.5/library/zipimport.html

[10] www.python.org/dev/peps/pep-0302

Приложение А

Замечания относительно портирования программ

Этот раздел содержит замечания и рекомендации по обновлению программ при переходе от версии Python 2 к версии Python 3, включая краткие описания изменений в каждом модуле и соответствующие ссылки.

А.1. Ссылки

Приведенные в этом разделе примечания основаны на документах *What's New*, подготовленных командой разработчиков Python и руководителями выпусков каждой версии.

- *What's New In Python 3.0*[1]
- *What's New In Python 3.1*[2]
- *What's New In Python 3.2*[3]
- *What's New In Python 3.3*[4]
- *What's New In Python 3.4*[5]
- *What's New In Python 3.5*[6]

Для получения более подробной информации относительно перехода на версию Python 3 обратитесь к следующим документам:

- *Porting Python 2 Code to Python 3*[7];
- *Supporting Python 3* (Lennart Regebro)[8];
- *Python-porting*[9]. Список рассылки.

А.2. Новые модули

Python 3 включает ряд новых модулей, предоставляющих средства, отсутствующие в Python 2:

---

[1] https://docs.python.org/3.0/whatsnew/3.0.html
[2] https://docs.python.org/3.1/whatsnew/3.1.html
[3] https://docs.python.org/3.2/whatsnew/3.2.html
[4] https://docs.python.org/3.3/whatsnew/3.3.html
[5] https://docs.python.org/3.4/whatsnew/3.4.html
[6] https://docs.python.org/3.5/whatsnew/3.5.html
[7] https://docs.python.org/3/howto/pyporting.html
[8] http://python3porting.com/
[9] http://mail.python.org/mailman/listinfo/python-porting

- `asyncio` (стр. 625). Асинхронный ввод-вывод, цикл событий и другие инструменты для параллельных вычислений.
- `concurrent.futures` (стр. 684). Управление пулами параллельно выполняющихся задач.
- `ensurepip` (стр. 1159). Установка программы `pip` (Python Package Installer).
- `enum` (стр. 98). Определяет тип `enumeration`.
- `ipaddress` (стр. 695). Классы для работы с IP-адресами.
- `pathlib` (стр. 329). Объектно-ориентированный API для работы с путями в файловой системе.
- `selectors` (стр. 731). Абстракции мультиплексирования ввода-вывода.
- `statistics` (стр. 315). Статистические расчеты.
- `venv` (стр. 1155). Создание изолированных контекстов установки и выполнения.

А.3. Переименованные модули

В качестве части мер, описанных в документе **PEP3108**, имена многих модулей, перешедших из версии Python 2 в версию Python 3, были изменены. Во всех новых именах модулей последовательно используется нижний регистр, а некоторые из них были перемещены в пакеты с целью улучшения организации родственных модулей. Во многих случаях код, использующий эти модули, можно легко обновить для работы в версии Python 3, всего лишь изменив инструкции `import`. Полный список переименованных модулей хранится в словаре `lib2to3.fixes.fix_imports.MAPPING` (ключами являются имена модулей в Python 2, а значениями — имена модулей в Python 3) и воспроизведен в табл. А.1.

Дополнительные ссылки

- Для написания кода, способного выполняться в версиях Python 2 и 3, можно использовать пакет `Six`[10]. В частности, модуль `six.moves` позволяет коду импортировать переименованные модули без изменения инструкции `import`, автоматически перенаправляя операцию импорта на корректную версию имени, в зависимости от используемой версии Python.
- **PEP 3108**[11]. *Standard Library Reorganization.*

Таблица А.1. Переименованные модули

Имя в Python 2	Имя в Python 3
`__builtin__`	`builtins`
`_winreg`	`winreg`
`BaseHTTPServer`	`http.server` (стр. 788)
`CGIHTTPServer`	`http.server` (стр. 788)
`commands`	`subprocess` (стр. 548)
`ConfigParser`	`configparser` (стр. 958)

[10] `http://pythonhosted.org/six/`

[11] `www.python.org/dev/peps/pep-3108`

Окончание табл. А.1

Имя в Python 2	Имя в Python 3
`Cookie`	`http.cookies` (стр. 796)
`cookielib`	`http.cookiejar`
`copy_reg`	`copyreg`
`cPickle`	`pickle` (стр. 416)
`cStringIO`	`io` (стр. 411)
`dbhash`	`dbm.bsd`
`dbm`	`dbm.ndbm`
`Dialog`	`tkinter.dialog`
`DocXMLRPCServer`	`xmlrpc.server` (стр. 832)
`dumbdbm`	`dbm.dumb`
`FileDialog`	`tkinter.filedialog`
`gdbm`	`dbm.gnu`
`htmlentitydefs`	`html.entities`
`HTMLParser`	`html.parser`
`httplib`	`http.client`
`Queue`	`queue` (стр. 141)
`repr`	`reprlib`
`robotparser`	`urllib.robotparser` (стр. 780)
`ScrolledText`	`tkinter.scrolledtext`
`SimpleDialog`	`tkinter.simpledialog`
`SimpleHTTPServer`	`http.server` (стр. 788)
`SimpleXMLRPCServer`	`xmlrpc.server` (стр. 832)
`SocketServer`	`socketserver` (стр. 749)
`StringIO`	`io` (стр. 411)
`Tix`	`tkinter.tix`
`tkColorChooser`	`tkinter.colorchooser`
`tkCommonDialog`	`tkinter.commondialog`
`Tkconstants`	`tkinter.constants`
`Tkdnd`	`tkinter.dnd`
`tkFileDialog`	`tkinter.filedialog`
`tkFont`	`tkinter.font`
`Tkinter`	`tkinter`
`tkMessageBox`	`tkinter.messagebox`
`tkSimpleDialog`	`tkinter.simpledialog`
`ttk`	`tkinter.ttk`
`urlparse`	`urllib.parse` (стр. 762)
`UserList`	`collections` (стр. 107)
`UserString`	`collections` (стр. 107)
`xmlrpclib`	`xmlrpc.client` (стр. 821)

A.4. Удаленные модули

Перечисленные ниже модули либо вообще отсутствуют в версии Python 3, либо содержавшиеся в них ранее средства перемещены в другие существующие модули.

A.4.1. bsddb

Модули bsddb и dbm.bsd удалены. Теперь привязки для Berkeley DB поддерживаются независимым модулем bsddb3[12], не входящим в состав стандартной библиотеки.

A.4.2. commands

Модуль commands был объявлен устаревшим в Python 2.6 и удален в версии Python 3.0. См. описание вытеснившего его модуля subprocess (стр. 548).

A.4.3. compiler

Модуль compiler удален. См. описание вытеснившего его модуля ast.

A.4.4. dircache

Модуль dircache удален полностью, без предоставления замены.

A.4.5. EasyDialogs

Модуль EasyDialogs удален. См. описание вытеснившего его модуля tkinter.

A.4.6. exceptions

Модуль exceptions удален, поскольку определенные в нем исключения теперь доступны в виде встроенных классов.

A.4.7. htmllib

Модуль htmllib удален. См. описание вытеснившего его модуля html.parser.

A.4.8. md5

Реализация алгоритма вычисления контрольных сумм сообщений MD5 перемещена в модуль hashlib (стр. 537).

A.4.9. mimetools, MimeWriter, mimify, multifile и rfc822

Модули mimetools, MimeWriter, mimify, multifile и rfc822 удалены. См. описание вытеснившего их модуля email.

A.4.10. popen2

Модуль popen2 удален. См. описание вытеснившего его модуля subprocess (стр. 548).

---

[12] https://pypi.python.org/pypi/bsddb3

A.4.11. posixfile

Модуль posixfile удален. См. описание вытеснившего его модуля io (стр. 411).

A.4.12. sets

Модуль sets был объявлен устаревшим в версии Python 2.6 и удален в версии Python 3.0. Вместо него следует использовать встроенные типы set и orderedset.

A.4.13. sha

Реализация алгоритма вычисления контрольных сумм сообщений SHA-1 перемещена в модуль hashlib (стр. 537).

A.4.14. sre

Модуль sre был объявлен устаревшим в версии Python 2.5 и удален в версии Python 3.0. Вместо него следует использовать модуль re (стр. 47).

A.4.15. statvfs

Модуль statvfs был объявлен устаревшим в версии Python 2.6 и удален в версии Python 3.0. См. описание вытеснившей его функции os.statvfs() модуля os (стр. 1217).

A.4.16. thread

Модуль thread удален. Вместо него следует использовать высокоуровневый API модуля threading (стр. 571).

A.4.17. user

Модуль user был объявлен устаревшим в Python 2.6 и удален в версии Python 3.0. См. описание вытеснивших его средств пользовательской адаптации, предоставляемых модулем site (стр. 1162).

A.5. Устаревшие модули

Эти модули все еще имеются в стандартной библиотеке, но объявлены устаревшими, и их не следует использовать в новых программах на Python 3.

A.5.1. asyncore и asynchat

Средства асинхронного ввода-вывода и обработчики протоколов. См. описание модуля asyncio (стр. 625).

A.5.2. formatter

Обобщенный форматировщик вывода и интерфейс устройств. Более подробная информация приведена в документе **Issue 18716**[13].

[13] http://bugs.python.org/issue18716

A.5.3. imp

Доступ к реализации инструкции import. Вместо этого модуля следует использовать модуль importlib (стр. 1315).

A.5.4. optparse

Библиотека средств синтаксического анализа параметров командной строки. Программный интерфейс модуля argparse (стр. 888) аналогичен тому, который предоставляет модуль optparse, и во многих случаях модуль argparse можно использовать в качестве непосредственной замены модуля optparse после соответствующего обновления имен используемых классов и методов.

A.6. Сводка изменений, внесенных в модули

A.6.1. abc

Декораторы abstractproperty(), abstractclassmethod() и abstract staticmethod() объявлены устаревшими. Сочетание декоратора abstract method() с декораторами property(), classmethod() и staticmethod() работает в соответствии с ожиданиями (документ **Issue 11610**[14]).

A.6.2. anydbm

В Python 3 модуль anydbm переименован в dbm (стр. 428).

A.6.3. argparse

Аргумент version функции ArgumentParser удален в пользу специального типа action (документ **Issue 13248**[15]).

В прежней форме номер версии передавался с помощью аргумента version.

```
parser = argparse.ArgumentParser(version='1.0')
```

Новая форма требует добавления явного определения аргумента.

```
parser = argparse.ArgumentParser()
parser.add_argument('--version', action='version',
                    version='%(prog)s 1.0')
```

Имя параметра и форматная строка номера версии могут быть изменены в соответствии со спецификой приложения.

В Python 3.4 действие параметра action для версии было изменено: строка с номером версии выводится в поток stdout вместо потока stderr (документ **Issue 18920**[16]).

[14] http://bugs.python.org/issue11610

[15] http://bugs.python.org/issue13248

[16] http://bugs.python.org/issue18920

A.6.4. array

Тип 'c', используемый в ранней версии Python 2 для символьных байтов, удален. Вместо него для байтов следует использовать 'b' или 'B'.

Тип 'u' для символов строк Unicode объявлен устаревшим и будет удален в версии Python 4.0.

Методы tostring() и fromstring() переименованы соответственно в tobytes() и frombytes() для устранения неоднозначности (документ **Issue 8990**[17]).

A.6.5. atexit

При обновлении модуля atexit (989) для включения C-реализации (документ **Issue 1680961**[18]) было допущено некоторое ухудшение функциональности в логике обработки ошибок, приведшее к тому, что отображались лишь краткие сведения об исключении без трассировочной информации. Этот недостаток был устранен в версии Python 3.3 (документ **Issue 18776**[19]).

A.6.6. base64

Функции encodestring() и decodestring() переименованы в encodebytes() и decodebytes() соответственно. Прежние имена все еще работают как псевдонимы, но их использование не рекомендуется (документ **Issue 3613**[20]).

Добавлены две новые кодировки, использующие 85-символьные алфавиты. Функция b85encode() реализует кодировку, используемую в системах управления версиями Mercurial и git, тогда как функция a85encode() реализует формат Ascii85, используемый в файлах PDF (документ **Issue 17618**[21]).

A.6.7. bz2

Теперь экземпляры класса BZ2File поддерживают протокол контекстного менеджера и не должны обертываться функцией contextlib.closing().

A.6.8. collections

Абстрактные базовые классы, ранее определенные в модуле collections (стр. 107), перемещены в модуль collections.abc (стр. 127) с обеспечением обратной совместимости с операциями импорта, доступными в модуле collections (документ **Issue 11085**[22]).

A.6.9. comands

Функции getoutput() и getstatusoutput() перемещены в модуль subprocess (стр. 548), а модуль commands удален.

---

[17] http://bugs.python.org/issue8990

[18] http://bugs.python.org/issue1680961

[19] http://bugs.python.org/issue18776

[20] http://bugs.python.org/issue3613

[21] http://bugs.python.org/issue17618

[22] http://bugs.python.org/issue11085

A.6.10. configparser

Прежний модуль ConfigParser переименован в configparser (стр. 958).

Прежний класс ConfigParser удален в пользу класса SafeConfigParser, который, в свою очередь, переименован в ConfigParser. Устаревшее интерполяционное поведение доступно через класс LegacyInterpolation.

Теперь метод read() поддерживает аргумент encoding, поэтому для чтения конфигурационных файлов, содержащих значения Unicode, не требуется использовать модуль codecs (стр. 386).

Использовать прежний класс RawConfigParser не рекомендуется. Для получения того же поведения в новых проектах следует использовать экземпляры ConfigParser(interpolation=None).

A.6.11. contextlib

Функция contextlib.nested() удалена. Вместо ее использования следует передавать множественные контекстные менеджеры той же инструкции with.

A.6.12. csv

Вместо того чтобы непосредственно вызывать метод next() объекта reader, следует использовать встроенную функцию next() для вызова итератора.

A.6.13. datetime

Начиная с версии Python 3.3 проверка на равенство между простыми и учитывающими часовые пояса экземплярами time возвращает значение False вместо возбуждения исключения TypeError (документ **Issue 15006**[23]).

До выпуска версии Python 3.5 объект datetime.time, представляющий полночь, при преобразовании в булево значение дает значение False. Это поведение устранено в версии Python 3.5 (документ **Issue 13936**[24]).

A.6.14. decimal

В Python 3.3 была внедрена C-реализация типа decimal (стр. 265), основанная на библиотеке libmpdec. В результате повысилась производительность, но это сопровождалось некоторыми изменениями API и отличиями в поведении от реализации на языке Python. Более подробно об этом можно прочитать в примечаниях к выпуску Python 3.3[25].

A.6.15. fractions

В методах класса from_float() и from_decimal() больше нет необходимости. Теперь значения с плавающей десятичной точкой и значения типа Decimal можно передавать непосредственно конструктору Fraction.

---

[23] http://bugs.python.org/issue15006

[24] http://bugs.python.org/issue13936

[25] https://docs.python.org/3.3/whatsnew/3.3.html#decimal

А.6.16. gc

Флаги DEBUG_OBJECT и DEBUG_INSTANCE удалены ввиду отсутствия необходимости в проведении различий между классами нового и старого стиля.

А.6.17. gettext

Во всех функциях перевода, входящих в модуль gettext (стр. 1001), предполагается, что форматом входных и выходных данных является Unicode, и все специальные варианты функций для работы с текстом в формате Unicode, такие как ugettext(), удалены.

А.6.18. glob

Новая функция escape() реализует обходной вариант для поиска в файлах, имена которых содержат метасимволы (документ **Issue 8402**[26]).

А.6.19. http.cookies

Помимо экранирования кавычек класс SimpleCookie кодирует запятые и двоеточия в значениях, что лучше отражает поведение реальных браузеров (документ **Issue 9824**[27]).

А.6.20. imaplib

В версии Python 3 функции, содержащиеся в модуле imaplib (стр. 865), возвращают байтовые строки в кодировке UTF-8. Предусмотрена поддержка получения строк Unicode и их автоматического кодирования при использовании в качестве отправляемых исходящих команд или имени пользователя и пароля для входа на сервер.

А.6.21. inspect

Функции getargspec(), getfullargspec(), getargvalues(), getcallargs(), getargvalues(), formatargspec() и formatargvalues() удалены в пользу функции signature() (документ **Issue 20438**[28]).

А.6.22. itertools

Функции imap(), izip() и ifilter() заменены версиями встроенных функций, которые возвращают итерируемые объекты вместо списочных объектов (map(), zip() и filter() соответственно). Функция ifilterfalse() переименована в filterfalse().

А.6.23. json

API модуля json (стр. 809) был обновлен, чтобы обеспечивалась только поддержка типа str, но не bytes, поскольку спецификация JSON определена с использованием Unicode.

---

[26] bugs.python.org/issue8402

[27] http://bugs.python.org/issue9824

[28] bugs.python.org/issue20438

A.6.24. locale

Нормализованная версия имени кодировки UTF-8 была изменена с "UTF8" на "UTF-8", поскольку Mac OS X и OpenBSD не поддерживают использование имени "UTF8" (документы **Issue 10154**[29] и **Issue 10090**[30]).

A.6.25. logging

Модуль logging (стр. 976) в настоящее время включает средство протоколирования lastResort, которое задействуется, если не предусмотрено использование какой-либо другой конфигурации ведения журнала приложением. Это избавляет приложение от необходимости конфигурировать ведение журнала исключительно для того, чтобы избежать вывода сообщений об ошибках в тех случаях, когда библиотека, импортируемая приложением, использует средства протоколирования, хотя само приложение их не использует.

A.6.26. mailbox

Модуль mailbox читает и записывает файлы почтового ящика, полагаясь на выполнение синтаксического анализа сообщений средствами пакета email. Класс StringIO и средства ввода текстовых файлов объявлены устаревшими (документ **Issue 9124**[31]).

A.6.27. mmap

Значения, возвращаемые программными интерфейсами средств чтения, представляют собой байтовые строки и нуждаются в декодировании, прежде чем их можно будет обрабатывать как текст.

A.6.28. operator

Функция div() удалена. В зависимости от требуемой семантики вместо нее следует использовать функцию floordiv() или truediv().

Функция repeat() удалена. Вместо нее нужно использовать функцию mul().

Функции getslice(), setslice() и delslice() удалены. Вместо них следует использовать функции getitem(), setitem() и delitem() соответственно, с применением синтаксиса срезов.

Функция isCallable() удалена. Вместо нее нужно использовать абстрактный базовый класс collections.Callable.

```
isinstance(obj, collections.Callable)
```

Функции проверки типов isMappingType(), isSequenceType() и isNumberType() удалены. Вместо них следует использовать соответствующие абстрактные базовые классы из модуля collections (стр. 107) или numbers.

[29] http://bugs.python.org/issue10154

[30] http://bugs.python.org/issue10090

[31] http://bugs.python.org/issue9124

```
isinstance(obj, collections.Mapping)
isinstance(obj, collections.Sequence)
isinstance(obj, numbers.Number)
```

Функция `sequenceIncludes()` удалена. Вместо нее следует использовать функцию `contains()`.

A.6.29. os

Функции `popen2()`, `popen3()` и `popen4()` удалены. Функция `popen()` по-прежнему доступна, но объявлена устаревшей, и в случае ее использования выводятся соответствующие предупреждения. Код, использующий эти функции, должен быть переписан с использованием модуля `subprocess` (стр. 548), чтобы обеспечить лучшую переносимость между операционными системами.

Функции `os.tmpnam()`, `os.tempnam()` и `os.tmpfile()` удалены. Вместо них следует использовать модуль `tempfile` (стр. 353).

Функция `os.stat_float_times()` объявлена устаревшей (документ **Issue 14711**[32]). Функция `os.unsetenv()` уже не игнорирует ошибки (документ **Issue 13415**[33]).

A.6.30. os.path

Функция `os.path.walk()` удалена. Вместо нее нужно использовать функцию `os.walk()`.

A.6.31. pdb

Команда `print` удалена, поэтому она не может использоваться в качестве эквивалента функции `print()` (документ **Issue 18764**[34]). Сокращенная форма `p` команды сохранена.

A.6.32. pickle

C-реализация модуля `pickle` из Python 2 перемещена в новый модуль, который по возможности автоматически используется вместо реализации на языке Python. В использовании прежней идиомы импорта, заключающейся в том, что перед поиском модуля `pickle` выполняется поиск `cPickle`, больше нет необходимости.

```
try:
    import cPickle as pickle
except:
    import pickle
```

В случае автоматического импортирования C-реализации требуется импортировать лишь непосредственно модуль `pickle`.

```
import pickle
```

[32] http://bugs.python.org/issue14711

[33] http://bugs.python.org/issue13415

[34] http://bugs.python.org/issue18764

Функциональная совместимость версий Python 2.*x* и 3.*x* улучшена для сериализованных данных, использующих протокол уровня 2 или более старый для разрешения проблем, обусловленных появлением значительного количества переименованных модулей стандартной библиотеки в процессе перехода к версии Python 3. Поскольку сериализованные с помощью модуля `pickle` данные включают ссылки на имена классов и типов, обмен сериализованными данными между программами, использующими версии Python 2 и 3, был затруднен. Теперь для данных, сериализованных по протоколу уровня 2 или более старому, при записи или чтении данных из потока `pickle` автоматически используются прежние имена классов.

Такое поведение доступно по умолчанию, но его можно отключить с помощью параметра `fix_imports`. Это изменение улучшает ситуацию, но не устраняет имеющуюся несовместимость полностью. В частности, данные, сериализованные с помощью модуля `pickle` в версии Python 3.1, не могут быть прочитаны в версии Python 3.0. Для гарантии максимальной переносимости между приложениями, рассчитанными на работу с версией Python 3, следует использовать протокол уровня 3, которым эта возможность улучшения совместимости не обеспечивается.

Используемая ранее по умолчанию версия 0 протокола, обеспечивающая удобочитаемость данных, теперь заменена версией 3, в которой используется двоичный формат, обеспечивающий наилучшую совместимость при совместном использовании данных приложениями, рассчитанными на работу с версией Python 3.

Данные в виде байтовых строк, записанные с помощью модуля `pickle` приложениями, рассчитанными на версию Python 2.*x*, декодируются при их чтении для создания объектов строк Unicode. По умолчанию кодирование осуществляется с использованием кодировки ASCII, но это поведение можно изменить, передавая значения классу `Unpickler`.

A.6.33. `pipes`

Функция `pipes.quote()` перемещена в модуль `shlex` (стр. 949). (Документ **Issue 9723**[35].)

A.6.34. `platform`

Функция `platform.popen()` объявлена устаревшей. Вместо нее следует использовать функцию `subprocess.popen()` (документ **Issue 11377**[36]).

Функция `platform.uname()` теперь возвращает именованный кортеж.

Поскольку для дистрибутивов Linux последовательный способ самоописания отсутствует, функции, предназначенные для получения описаний (`platform.dist()` и `platform.linux_distribution()`), объявлены устаревшими и запланированы для удаления в версии Python 3.7 (документ **Issue 1322**[37]).

A.6.35. `random`

Функция `jumpahead()` была удалена в версии Python 3.0.

---

[35] `http://bugs.python.org/issue9723`

[36] `http://bugs.python.org/issue11377`

[37] `http://bugs.python.org/issue1322`

A.6.36. re

Флаг UNICODE представляет поведение по умолчанию. Для восстановления ASCII-специфического поведения Python 2 следует использовать флаг ASCII.

A.6.37. shelve

Используемый по умолчанию выходной формат для модуля shelve (стр. 425) может создать файл с расширением *.db*, которое добавляется к имени, предоставленному функции shelve.open().

A.6.38. signal

Документ **PEP 475**[38] обязывает делать повторные попытки выполнения системных вызовов, которые были прерваны и вернули код ошибки EINTR. Это требование изменяет поведение обработчиков сигналов и других системных вызовов. Теперь после возврата управления обработчиком сигнала прерванный вызов будет повторяться, если обработчик сигнала не возбудил исключение. Для получения более подробной информации следует обратиться к указанному документу PEP.

A.6.39. socket

В Python 2 объекты типа str обычно могут непосредственно передаваться через сокет. Поскольку в Python 3 тип str заменяет тип unicode, эти значения должны преобразовываться перед их отправкой. В примерах, приведенных при описании модуля socket (701), используются уже преобразованные байтовые строки.

A.6.40. socketserver

Модуль socketserver (стр. 749) в Python 2 назывался SocketServer.

A.6.41. string

Все функции из модуля string (стр. 35), являющиеся также методами объектов str, удалены.

Константы letters, lowercase и uppercase удалены. Новые константы с аналогичными именами ограничены символьным набором ASCII.

Функция maketrans() заменена методами для типов str, bytes и bytearray с целью более точного указания того, какие входные типы поддерживаются каждой таблицей преобразования.

A.6.42. struct

Функция struct.pack() теперь поддерживает только байтовые строки, если упаковываются строки s, и не выполняет неявное кодирование строковых объектов в UTF-8 (документ **Issue 10783**[39]).

---

[38] www.python.org/dev/peps/pep-0475

[39] http://bugs.python.org/issue10783

A.6.43. subprocess

Значение по умолчанию аргумента close_fds конструктора subprocess.Popen теперь не всегда равно False. Оно всегда равно True для Unix. В Windows значение этого аргумента по умолчанию равно True, если аргументы стандартных потоков ввода-вывода установлены в None. Во всех остальных случаях значение close_fds по умолчанию равно False.

A.6.44. sys

Значение переменной sys.exitfunc больше не используется для проверки необходимости выполнения операций по очистке ресурсов при выходе из программы. Вместо этого следует использовать модуль atexit (стр. 989).

Переменная variable sys.subversion не определена.

Флаги sys.flags.py3k_warning, sys.flags.division_warning, sys.flags.division_new, sys.flags.tabcheck и sys.flags.unicode не определены.

Переменная sys.maxint не определена. Вместо нее следует использовать переменную sys.maxsize. См. документ **PEP 237** (*Unifying Long Integers and Integers*)[40].

Переменные sys.exc_type, sys.exc_value и sys.exc_traceback, используемые для доступа к трассировочной информации исключений, удалены. Функция sys.exc_clear() также удалена.

Переменная sys.version_info теперь представляет экземпляр именованного кортежа с атрибутами major, minor, micro, releaselevel и serial (документ **Issue 4285**[41]).

Проверка интервала переключения потоков, управляющая допустимым количеством опкодов, которые могут быть выполнены до того, как будет переключен контекст потока, заменена проверкой значения абсолютного времени, которое управляется функцией sys.setswitchinterval(). Прежние функции, управляющие проверкой интервала переключения, sys.getcheckinterval() и sys.setcheckinterval(), объявлены устаревшими.

Переменные sys.meta_path и sys.path_hooks теперь предоставляют все средства и расширения для поиска импортируемых модулей. В прежних версиях предоставлялись только те средства и расширения, которые были добавлены явным образом, а С-импорт использовал значения из собственной реализации, которые невозможно было изменить внешними средствами.

Для Linux-систем переменная sys.platform уже не включает номер версии. Теперь это просто значение linux, а не linux2 или linux3.

A.6.45. threading

Модуль thread объявлен устаревшим, и вместо него рекомендуется применять API модуля threading (стр. 571).

Доступные в модуле threading средства отладки, включая аргумент "verbose", исключены из программных интерфейсов (документ **Issue 13550**[42]).

В прежних реализациях модуля threading для некоторых классов использовались функции-фабрики, поскольку они были реализованы в виде С-расширений и

[40] www.python.org/dev/peps/pep-0237

[41] http://bugs.python.org/issue4285

[42] http://bugs.python.org/issue13550

не могли наследоваться. Это языковое ограничение снято, поэтому многие функции-фабрики преобразованы в стандартные классы, допускающие создание подклассов (документ **Issue 10968**[43]).

Общедоступные символы, экспортируемые из модуля `threading`, переименованы для приведения в соответствие с требованиями документа **PEP 8**[44]. Прежние имена сохранены для обеспечения обратной совместимости, но будут удалены в будущих выпусках.

A.6.46. `time`

Функции `time.asctime()` и `time.ctime()` изменены так, что не могут использовать системные функции с тем же временем, чтобы разрешить указание более высоких значений годов. Функция `time.ctime()` теперь поддерживает года от 1900 до `maxint`, хотя для значений свыше 9999 длина выходной строки превышает стандартные 24 символа, позволяя использовать дополнительные цифры для года (документ **Issue 8013**[45]).

A.6.47. `unittest`

Методы класса `TestCase`, названия которых начинаются с префикса "fail" (например, `failIf()`, `failUnless()`), объявлены устаревшими. Вместо них следует использовать альтернативные формы методов "assert".

Некоторые из псевдонимов прежних методов объявлены устаревшими и заменены предпочтительными именами. Использование устаревших имен сопровождается выводом предупреждающего сообщения (документ **Issue 9424**[46]). Соответствие между старыми и новыми именами приведено в табл. А.2.

Таблица А.2. Устаревшие методы класса `unittest.TestCase`

Устаревшее имя	Предпочтительное имя
`assert_()`	`assertTrue()`
`assertEquals()`	`assertEqual()`
`assertNotEquals()`	`assertNotEqual()`
`assertAlmostEquals()`	`assertAlmostEqual()`
`assertNotAlmostEquals()`	`assertNotAlmostEqual()`

A.6.48. Классы `UserDict`, `UserList` и `UserString`

Классы `UserDict`, `UserList` и `UserString` перемещены из их собственных модулей в модуль `collections` (стр. 107). Типы `dict`, `list` и `str` могут образовываться путем непосредственного создания подклассов, но классы, содержащиеся в модуле `collections`, могут упростить реализацию подклассов, поскольку содержимое контейнера доступно непосредственно через атрибут экземпляра. Абстрактные классы из модуля `collections.abc` (стр. 127) также могут быть по-

[43] `http://bugs.python.org/issue10968`

[44] `www.python.org/dev/peps/pep-0008`

[45] `http://bugs.python.org/issue8013`

[46] `http://bugs.python.org/issue9424`

лезными для создания пользовательских контейнеров, следующих программным интерфейсам встроенных типов.

A.6.49. uuid

Теперь функция uuid.getnode() использует переменную среды PATH для поиска программ, которые могут предоставлять информацию о MAC-адресе хоста в Unix (документ **Issue 19855**[17]). Если такие программы не удается найти вдоль путей поиска модулей, поиск продолжается в каталогах /sbin и /usr/sbin. При наличии альтернативных версий таких программ, как netstat, ifconfig, ip и arp, описанное поведение поиска может приводить к результатам, которые отличаются от результатов, получаемых в более ранних версиях Python.

A.6.50. whichdb

Функциональность модуля whichdb перемещена в модуль dbm (стр. 578).

A.6.51. xml.etree.ElementTree

Класс XMLTreeBuilder переименован в TreeBuilder, а API претерпел несколько изменений.

Функция ElementTree.getchildren() объявлена устаревшей. Вместо нее для получения списка дочерних объектов следует использовать функцию list(elem).

Функция ElementTree.getiterator() объявлена устаревшей. Вместо нее следует использовать функцию iter(), создающую итератор с применением обычного итерационного протокола.

Если выполнить синтаксический анализ не удается, то теперь вместо исключения xml.parsers.expat.ExpatError анализатор XMLParser возбуждает исключение xml.etree.ElementTree.ParseError.

A.6.52. zipimport

Функция get_data() возвращает данные в виде байтовой строки, которая должна быть декодирована, прежде чем ее можно будет использовать в качестве строки Unicode.

[47] http://bugs.python.org/issue19855

Приложение Б

Внешние ресурсы, дополняющие стандартную библиотеку

Несмотря на обширность стандартной библиотеки Python, она дополнена робастной экосистемой модулей, предоставляемых сторонними разработчиками и доступных через репозиторий Python Package Index (PyPI)[1]. В данном приложении описаны некоторые из этих модулей и ситуации, в которых они могут дополнить или даже заменить модули стандартной библиотеки.

Б.1. Текст

Модуль `string` (раздел 1.1) включает простейший инструмент для создания шаблонов. Многие веб-фреймворки включают более мощные инструменты этой категории, однако наиболее популярными независимыми альтернативами являются Jinja[2] и Mako[3]. Оба фреймворка поддерживают циклические и условные управляющие конструкции, а также другие возможности объединения данных с шаблонами для получения текстового вывода.

Модуль `re` (раздел 1.3) включает функции, обеспечивающие поиск и синтаксический анализ текста с использованием формализованного описания шаблонов на основе регулярных выражений. Однако это не единственный способ анализа текста.

Пакет PLY[4] поддерживает создание парсеров в стиле инструментов GNU `lexx` и `yacc`, которые часто используются для создания компиляторов. Предоставляя входные данные, описывающие допустимые лексемы, грамматику и действия, которые необходимо предпринимать для каждой встретившейся лексемы, можно создавать полнофункциональные компиляторы и интерпретаторы, а также более совершенные лексические анализаторы данных.

Еще одним инструментом для создания парсеров является PyParsing[5]. Входными данными для него служат экземпляры классов, которые могут объединяться в цепочку с помощью операторов и вызовов функций для создания грамматики.

Наконец, NLTK[6] — это пакет, предназначенный для обработки текста, написанного на естественном языке, т.е. не на компьютерном языке, а на языке человеческого общения. Этот пакет поддерживает разбиение предложений на части речи, нахождение корней слов и простейшую семантическую обработку.

---

[1] `https://pypi.python.org/pypi`

[2] `http://jinja.pocoo.org`

[3] `http://docs.makotemplates.org/en/latest/`

[4] `www.dabeaz.com/ply/`

[5] `http://pyparsing.wikispaces.com`

[6] `www.nltk.org`

Б.2. Алгоритмы

Модуль `functools` (раздел 3.1) включает инструменты для создания декораторов, которые являются функциями, обертывающими другие функции для изменения их поведения. Пакет `wrapt`[7] предлагает больше, чем функция `functools.wrap()`: он гарантирует корректность конструирования декоратора и его нормальную работу в граничных случаях.

Б.3. Дата и время

Модули `time` (раздел 4.1) и `datetime` (раздел 4.2) предоставляют функции и классы для манипулирования значениями даты и времени. Оба они включают функции, анализирующие строки с целью преобразования их во внутреннее представление. Пакет `dateutil`[8] включает более гибкий парсер, упрощающий создание робастных приложений, более терпимых к различным входным форматам.

Модуль `datetime` включает класс, учитывающий часовые пояса для представления конкретного времени и конкретной даты. В то же время он не включает полную базу данных часовых поясов. Такую базу предоставляет пакет `pytz`[9]. Он распространяется отдельно от стандартной библиотеки, поскольку поддерживается другими авторами и оперативно обновляется в случае изменения часовых поясов или параметров перехода на летнее время органами, которые их контролируют.

Б.4. Математика

Модуль `math` (раздел 5.4) содержит быструю реализацию сложных математических функций. Библиотека NumPy[10] расширяет набор поддерживаемых функций, включая в него функции для выполнения операций линейной алгебры и преобразований Фурье. Кроме того, она включает реализацию быстрых многомерных массивов, которая улучшает версию, предлагаемую модулем `array` (раздел 2.3).

Б.5. Постоянное хранение и обмен данными

В приведенных при описании модуля `sqlite3` (раздел 7.4) примерах SQL-инструкции применялись непосредственно и работали с низкоуровневыми структурами данных. В случае крупных приложений часто желательно сопоставлять классы с таблицами баз данных, используя *объектно-реляционное отображение* (ORM). Библиотека ORM SQLAlchemy[11] предоставляет программные интерфейсы для связывания классов с таблицами, создания запросов и подключения к различным промышленным реляционным базам данных.

---

[7] `http://wrapt.readthedocs.org/`

[8] `https://dateutil.readthedocs.io/`

[9] `http://pythonhosted.org/pytz/`

[10] `www.numpy.org`

[11] `www.sqlalchemy.org`

Пакет `lxml`[12] обертывает библиотеки `libxml2` и `libxslt` для создания альтернативы XML-анализатору, предлагаемому модулем `xml.etree.ElementTree` (раздел 7.5). Возможно, разработчикам, имеющим опыт использования этих библиотек в других языках, будет легче адаптировать этот пакет для работы в Python.

Пакет `defusedxml`[13] содержит исправления, обеспечивающие защиту от атак типа "billion laughs"[14] и устраняющие другие уязвимости в XML-библиотеках Python в отношении DoS-атак, основанных на лавинообразном увеличении количества анализируемых объектов. Кроме того, использование этого пакета повышает безопасность работы с не заслуживающими доверия XML-документами по сравнению с использованием одной только стандартной библиотеки.

Б.6. Криптография

Разработчики пакета `cryptography`[15], по их собственным словам, ставили перед собой цель создать продукт, заслуживающий названия "стандартная библиотека криптографии". Данный криптографический пакет предоставляет высокоуровневые программные интерфейсы, упрощающие добавление средств криптографии в приложения. Пакет активно сопровождается и часто обновляется для устранения проблем, связанных с уязвимостями базовых библиотек, таких как OpenSSL.

Б.7. Параллельные вычисления: процессы, потоки и сопрограммы

Цикл событий, встроенный в модуль `asyncio` (раздел 10.5), является эталонной реализацией, которая основана на определенном в модуле абстрактном API. Имеется возможность использовать вместо этого цикла событий библиотеку, такую как `uvloop`[16], которая обеспечивает лучшую производительность в обмен на дополнительные зависимости приложения.

Пакет `curio`[17] – другой пакет, аналогичный модулю `asyncio` и поддерживающий параллелизм, но с меньшим API, который трактует все как сопрограмму. Он не поддерживает обратные вызовы так, как это делает пакет `asyncio`.

Библиотека Twisted[18] предоставляет расширяемый фреймворк для создания программ на языке Python, специально сфокусированный на сетевом программировании, основанном на событиях и интеграции нескольких протоколов. Это прошедшая испытание временем, робастная и хорошо документированная библиотека.

---

[12] http://lxml.de

[13] https://pypi.python.org/pypi/defusedxml

[14] http://en.wikipedia.org/wiki/Billion_laughs

[15] https://cryptography.io/en/latest/

[16] http://uvloop.readthedocs.io

[17] https://github.com/dabeaz/curio

[18] https://twistedmatrix.com/

Б.8. Интернет

Пакет `requests`[19] – весьма популярная замена модулю `urllib.request` (раздел 12.2). Он предоставляет унифицированный API для работы с удаленными ресурсами, адресуемыми посредством HTTP, включает надежную поддержку SSL и может использовать создание пулов соединений для повышения производительности многопоточных приложений. Он также предоставляет средства, которые делают его хорошо приспособленным для доступа к программным интерфейсам REST, таким как встроенный парсинг JSON.

Модуль `html` в Python включает базовый анализатор корректно сформированных HTML-данных. Однако реальные данные хорошо структурированы лишь в редких случаях, что делает анализ проблематичным. Библиотеки BeautifulSoup[20] и PyQuery[21] – это альтернативы модулю `html`, являющиеся более надежными при работе с небрежно подготовленными данными. Обе библиотеки определяют программные интерфейсы для парсинга, изменения и конструирования HTML-документов.

Встроенный пакет `http.server` (раздел 12.5) включает базовые классы, предназначенные для создания простых HTTP-серверов с нуля. Он предлагает не так уж много поддержки сверх той, которая требуется для создания веб-приложений. Django[22] и Pyramid[23] – два популярных фреймворка, которые обеспечивают более широкую поддержку таких развитых средств, как анализ запросов, перенаправление URL-адресов и обработка cookie-файлов.

Многие из существующих библиотек не работают с модулем `asyncio` (раздел 10.5), поскольку они не обновлялись для работы с циклом событий. Для заполнения этой бреши в настоящее время создается новый набор библиотек, в том числе таких, как библиотека `asaiohttp`[24], которые разрабатываются в рамках проекта `aio-libs`[25].

Б.9. Электронная почта

API модуля `imaplib` (раздел 13.4) – относительно низкоуровневый и требует от разработчика знания протокола IMAP для создания запросов и анализа результатов. Пакет `imapclient`[26] предоставляет высокоуровневый API, упрощающий работу при создании приложений, которым приходится манипулировать почтовыми ящиками IMAP.

---

[19] http://docs.python-requests.org/
[20] www.crummy.com/software/BeautifulSoup/
[21] http://pyquery.rtfd.org/
[22] www.djangoproject.com/
[23] https://trypyramid.com/
[24] http://aiohttp.readthedocs.io/
[25] https://github.com/aio-libs
[26] http://imapclient.freshfoo.com/

Б.10. Строительные блоки приложений

Оба модуля стандартной библиотеки, предназначенные для создания интерфейса командной строки, `argparse` (раздел 14.1) и `getopt` (раздел 14.2), отделяют определение аргументов командной строки от их анализа и обработки значений. Альтернативный пакет `click`[27] ("*C*ommand-*L*ine *I*nterface *C*onstruction *K*it") определяет функции для обработки команд, а затем связывает определения командной подсказки и параметров с этими командами, используя декораторы.

Пакет `cliff`[28] ("*C*ommand-*L*ine *I*nterface *F*ormulation *F*ramework") предоставляет набор базовых классов для определения команд и систему плагинов для расширения приложений с помощью подкоманд, которые можно распределять по отдельным пакетам. Для создания справочного текста и средств разбора аргументов командной строки этот пакет использует модуль `argparse` (раздел 14.1) стандартной библиотеки.

Пакет `docopt`[29] обращает типичную последовательность действий, предлагая разработчику подготовить текст справки для программы, который затем анализируется для определения корректных комбинаций параметров и подкоманд.

Пакет `prompt_toolkit`[30] включает такие развитые средства для интерактивных терминальных программ, как поддержка цвета, цветовая подсветка синтаксиса, редактирование вводимых данных, поддержка мыши и история команд с возможностями поиска. Его можно использовать для создания программ с текстовым интерфейсом, напоминающим интерфейс, обеспечиваемый модулем `cmd` (раздел 14.5), или полноэкранных приложений наподобие текстовых редакторов.

Наряду с таким популярным средством конфигурирования приложений, как INI-файлы, аналогичные тем, которые используются модулем `configparser` (раздел 14.7), для тех же целей широко используется файловый формат YAML[31]. Он предоставляет многие из возможностей описания структур данных, предлагаемых форматом JSON, но в более удобочитаемом виде. Библиотека PyYAML[32] предоставляет доступ к парсеру и сериализатору данных в формате YAML.

Б.11. Инструменты разработки

Модуль `venv` (раздел 16.13) стандартной библиотеки — новый в Python 3. Аналогичную изоляцию приложений в рамках как Python 2, так и Python 3 обеспечивает модуль `virtualenv`.[33]

Пакет `fixtures`[34] предоставляет несколько тестовых классов для управления ресурсами, приспособленных для работы с методом `addCleanup()` экземпляров `TestCase` из модуля `unittest` (раздел 16.3). Предлагаемые пакетом классы могут управлять средствами протоколирования, переменными среды, временными

[27] `http://click.pocoo.org`
[28] `http://docs.openstack.org/developer/cliff/`
[29] `http://docopt.org`
[30] `http://python-prompt-toolkit.readthedocs.io/en/stable/`
[31] `http://yaml.org`
[32] `http://pyyaml.org`
[33] `https://virtualenv.pypa.io/`
[34] `https://pypi.python.org/pypi/fixtures`

файлами и многими другими объектами последовательным и безопасным способом, гарантируя полную изоляцию каждого теста от остальных тестов набора.

Модуль `distutils` стандартной библиотеки, предназначенный для создания дистрибутивных пакетов модулей Python, объявлен устаревшим, Расширение `setuptools`[35] поставляется отдельно от стандартной библиотеки с целью упрощения частой доставки его обновленных версий. Его API включает инструменты для создания списка файлов, включаемых в пакет. Доступны расширения, автоматически создающие такие списки из набора файлов, охватываемых системами управления версиями. Например, использование расширения `setuptools-git`[36] по отношению к исходному коду из репозитория Git[37] по умолчанию приводит к тому, что в пакет включаются все отслеживаемые файлы. После того как пакет будет сформирован, приложение `twine`[38] выгрузит его в каталог пакетов для совместного использования с другими разработчиками.

Такие инструменты, как `tabnanny` (раздел 16.10), хорошо справляются с выявлением распространенных ошибок форматирования в коде Python. Организация Python Code Quality Authority[39] поддерживает широкий спектр более совершенных инструментов статического анализа, включая инструменты, обеспечивающие соблюдение стиля оформления кода, поиск распространенных ошибок программирования и даже оказание содействия в снижении избыточной сложности кода.

Дополнительные ссылки

- Python Package Index[40] (PyPI). Сайт, обеспечивающий поиск и загрузку модулей расширения, распространяемых независимо от основных пакетов Python.

[35] `https://setuptools.readthedocs.io/en/latest/`
[36] `https://pypi.python.org/pypi/setuptools-git`
[37] `https://git-scm.com`
[38] `https://pypi.python.org/pypi/twine`
[39] `http://meta.pycqa.org/en/latest/`
[40] `https://pypi.python.org/pypi`

Указатель модулей Python

Предметный указатель

К

Т

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON
учебный курс

Роберт Седжвик
Кевин Уэйн
Роберт Дондеро

www.dialektika.com

Авторы книги сосредоточиваются на самых полезных и важных средствах языка Python и не стремятся к его абсолютно полному охвату. Весь код этой книги был отработан и проверен на совместимость как с языком Python 2, так и Python 3, что делает его подходящим для каждого программиста и любого курса на много лет вперед.

Особенности книги:

- всеобъемлющий, основанный на приложениях подход: изучение языка Python на примерах из области науки, математики, техники и коммерческой деятельности;
- основное внимание главному: самым полезным и важным средствам языка Python;
- совместимость примеров кода проверена на языках Python 2.x и Python 3.x;
- во все главы включены разделы с вопросами и ответами, упражнениями и практическими упражнениями.

ISBN 978-5-9908462-1-0 **в продаже**

АВТОМАТИЗАЦИЯ РУТИННЫХ ЗАДАЧ С ПОМОЩЬЮ PYTHON

практическое руководство для начинающих

Эл Свейгарт

www.williamspublishing.com

Книга научит вас использовать Python для написания программ, способных в считанные секунды сделать то, на что раньше у вас уходили часы ручного труда, причем никакого опыта программирования от вас не требуется. Как только вы овладеете основами программирования, вы сможете создавать программы на языке Python, которые будут без труда выполнять в автоматическом режиме различные полезные задачи, такие как:

- поиск определенного текста в файле или в множестве файлов;
- создание, обновление, перемещение и переименование файлов и папок;
- поиск в Интернете и загрузка онлайн-контента;
- обновление и форматирование данных в электронных таблицах Excel любого размера;
- разбиение, слияние, разметка водяными знаками и шифрование PDF-документов;
- рассылка напоминаний в виде сообщений электронной почты или текстовых уведомлений;
- заполнение онлайновых форм.

ISBN 978-5-8459-2090-4 в продаже