Содержание

Предисловие

С 2008 года весь мир программистов на Python наблюдает за медленной эволюцией версии Python 3. Все были практически уверены, что переход к версии Python 3 займет много времени. Фактически даже на момент написания этой книги (2013 год) большинство практикующих программистов на Python продолжает использовать версию Python 2 в своих проектах. Во многом потому, что у версии Python 3 нет обратной совместимости с предыдущими релизами. Безусловно, обратная совместимость – проблема для всех, у кого уже есть кодовая база на предыдущей версии языка. Но если вы взглянете в будущее, то обнаружите, что Python 3 предлагает гораздо больше возможностей, чем кажется на первый взгляд.

Как и будущие преимущества версии Python 3, данная книга представляет собой серьезное обновление по сравнению с предыдущими изданиями. Прежде всего это весьма перспективное руководство. Все примеры исходного кода были написаны и протестированы в среде Python 3.3, независимо от предыдущих версий Python или «старого способа» верстки кода. Фактически многие из примеров будут работать только в версии Python 3.3 или более поздней. Возможно, это рискованно, но финальная цель – написать книгу рецептов, основанную на самых современных инструментах и существующих идиомах. Есть надежда, что эти рецепты могут послужить руководством для людей, программирующих на Python 3, или тех, кто пытается модернизировать существующий код.

Излишне упоминать, что написание книги рецептов в таком стиле представляет собой определенную издательскую проблему. При поиске примеров кода на языке Python в интернете находятся тысячи полезных листингов на таких сайтах, как ActiveState[1] или Stack Overflow[2]. Тем не менее большинство из этих рецептов насквозь пропахло историей и прошлым. Помимо того что почти все они написаны исключительно для Python 2, листинги часто содержат обходные пути и приемы, связанные с различиями между старыми версиями Python (например, версиями 2.3 и 2.4). Более того, в них нередко используются устаревшие методы, которые в дальнейшем стали встроенным функционалом в версии Python 3.3. Найти рецепты, ориентированные исключительно на Python 3, довольно ложно.

Поэтому, вместо того чтобы искать заточенные под Python 3 примеры кода, главы этой книги написаны под вдохновением от существующего кода и методов. Эти идеи стали базисом при разработке оригинальных, новых рецептов, использующих самые современные методы программирования на языке Python. Таким образом, данная книга отлично подойдет всем, кто хочет переписать свой код в современном стиле.

При выборе рецептов, которые следует включить, становится ясно, что просто невозможно написать книгу, которая охватила бы все приемы, которые можно сотворить на Python. Поэтому приоритет был отдан темам, фокусирующимся на ключевых аспектах языка Python, а также задачам, подходящим для широкого

---

[1] code.activestate.com/recipes/langs/python/.

[2] stackoverflow.com/questions/tagged/python.

спектра областей применения. Кроме того, многие из рецептов призваны проиллюстрировать функции, впервые появившиеся в версии Python 3 и, скорее всего, неизвестные даже опытным программистам, использующим более старые версии языка. Также отдавалось предпочтение рецептам, иллюстрирующим общепринятые методы программирования (т. е. так называемые шаблоны (или паттерны) программирования), а не пытающимся узким методом решить крайне специфическую практическую проблему. Хотя некоторые сторонние пакеты и описаны в книге, большинство рецептов касается основного языка и стандартной библиотеки Python.

Кому предназначена эта книга

Эта книга предназначена опытным программистам на Python, которые хотят углубить свои познания языка и современных идиом программирования. Бóльшая часть материала посвящена продвинутым методам, используемым библиотеками, фреймворками и приложениями. Примеры кода, приведенные в книге, обычно предполагают, что у читателя уже есть необходимые знания по соответствующей теме (например, общие аспекты информатики, структуры данных, сложность, системное программирование, параллелизм, программирование на C и т. п.). Более того, рецепты кода часто представляют собой лишь каркасы, которые предоставляют необходимую информацию для начала работы, но требуют от читателя дополнительных действий по верстке недостающих деталей кода. Поэтому предполагается, что читатель знает, как использовать поисковые системы и шикарную документацию Python в интернете.

Многие из самых продвинутых рецептов вознаградят труды читателя гораздо более глубоким пониманием, как Python работает «за кадром». Вы изучите новые приемы и приемы, которые можно применить к вашему собственному коду.

Кому эта книга противопоказана

Эта книга – не лучший выбор для начинающих, пытающихся научиться азам работы с Python. Предполагается, что вы уже знаете основы, которым можно научиться по учебнику языка Python для начинающих или другому самоучителю по теме. Данная книга не сыграет роль краткого справочного руководства (например, для быстрого поиска тех или иных функций в определенном модуле). Вместо этого книга нацелена на конкретные темы программирования, демонстрирует возможные решения и служит базисом для перехода к более сложным материалам, которые вы можете найти в интернете или в других, более продвинутых книгах.

Условные обозначения

В этой книге используются следующие условные обозначения:

Курсивный шрифт
: Курсивом обозначены новые термины, URL-адреса, адреса электронной почты, имена файлов и расширения файлов.

`Моноширинный шрифт`
: Используется для оформления листингов программ, а также в основном тексте для обозначения фрагментов кода программы, таких как имена переменных или функций, имена баз данных, типы данных, переменные среды, инструкции и ключевые слова.

`Моноширинный полужирный шрифт`
: Обозначает команды или другой текст, который должен быть набран пользователем.

`Моноширинный курсивный шрифт`
: Обозначает текст, который должен быть заменен пользователем его собственными значениями или значениями, определенными контекстом.


Этим значком обозначены советы, указания и примечания.


Этим значком обозначены предупреждения или предостережения.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Благодарности

Мы хотели бы поблагодарить технических рецензентов, Джейка Вандерпласа, Роберта Керна и Андреа Кротти, за их очень полезные комментарии, а также сообщество программистов Python за их всестороннюю поддержку. Мы также благодарим редакторов предыдущего издания – Алекса Мартелли, Анну Равенскрофт и Дэвида Ашера. Хотя это издание практически переписано с нуля, предыдущая книга послужила ориентиром при выборе потенциально интересных тем и рецептов. Наконец, что не менее важно, мы хотели бы поблагодарить читателей предыдущих изданий книги за их комментарии и предложения по улучшению.

Благодарности Дэвида Бизли

Написать книгу – задача не из легких. Поэтому я хотел бы поблагодарить мою жену Паулу и двух моих мальчиков за их терпение и поддержку во время работы над этим проектом. Бóльшая часть информации для этой книги была взята из материалов, которые я разработал, преподавая курсы программирования на Python последние шесть лет. Поэтому я хотел бы поблагодарить всех студентов, которые посещали мои курсы и в конечном итоге привели к созданию этой книги. Я также хотел бы поблагодарить Неда Батчелдера, Трэвиса Олифанта, Питера Вана, Брайана Ван де Вена, Хьюго Ши, Рэймонда Хеттингера, Майкла Фурда и Дэниела Клайна за замещение меня в поездках в обучающие центры по всему миру, пока я писал книгу у себя дома в Чикаго. Меган Бланшетт и Рэйчел Румелиотис из издательства O'Reilly также сыграли важную роль в доведении этого проекта до конца, несмотря на несколько трагических провалов и непредвиденных задержек. Наконец, что не менее важно, я хотел бы поблагодарить сообщество программистов на Python за их непрерывную поддержку и терпение к полетам моих дьявольских фантазий.

Дэвид М. Бизли
www.dabeaz.com
twitter.com/dabeaz

Благодарности Брайана Джонса

Я благодарю моего соавтора, Дэвида Бизли, а также Меган Бланшетт и Рэйчел Румелиотис из издательства O'Reilly за сотрудничество со мной во время работы над этим проектом. Я также хотел бы поблагодарить мою удивительную жену Наташу за ее терпение и поддержку как в этом проекте, так и во всех моих амбициях. Но больше всего я хотел бы поблагодарить сообщество программистов на Python. Несмотря на то что я участвовал в поддержке различных проектов с открытым исходным кодом, языков, клубов и тому подобного, ни одна работа не была настолько приятной и полезной, как связанная с сообществом программистов на Python.

Брайан К. Джонс
www.protocolostomy.com
twitter.com/bkjones

Об авторе

Дэвид Бизли является независимым разработчиком программного обеспечения и автором книг, живет в городе Чикаго. Он в основном работает над инструментами программирования, обеспечивая разработку пользовательского программного обеспечения и преподавая практические курсы программирования для разработчиков, ученых и инженеров. Он наиболее известен своей работой с языком программирования Python, для которого создал несколько пакетов с открытым исходным кодом (например, Swig и PLY) и является автором знаменитой Python Essential. Он также имеет значительный опыт работы с системным программированием на C, C++ и ассемблере.

Брайан К. Джонс – системный администратор на факультете компьютерных наук Принстонского университета.

Вступительное слово

Python – уникальный язык программирования. При желании я смогла бы научить писать на Python кого угодно за час. Однако его простота часто является и его недостатком. Чтобы создавать чистый и внятный код, требуется поучиться.

Книга, которую вы держите в руках, даст четкое понимание, как именно стоит писать приложения на Python, чтобы не пришлось раз за разом их переделывать и чтобы другие могли наслаждаться их чтением. В ней разобраны самые популярные подходы для создания легкочитаемого, оптимизированного и поддерживаемого кода. В книге также объясняется, как применять полученные знания в прикладных областях, все главы сопровождаются конкретными примерами.

Вас может смутить отсутствие поддержки самых последних стандартов языка: книга основана на версии Python 3.3, но это не должно сильно пугать вас. Основные концепты языка не меняются с самого момента его создания, и в данной работе Дэвид Бизли сконцентрировался на основополагающих моментах. Книга идеально подойдет как настольный справочник и источник вдохновения для вашей работы.

Я очень советую тратить больше времени на тренировки, чем на непосредственное чтение: именно практика позволит освоить и закрепить изученный материал. Читайте главы по мере пробуждения вашего интереса к ним.

После прочтения книги я настоятельно рекомендую вам ближе познакомиться с другими работами Дэвида Бизли. Это один из самых известных гуру Python, его доклады на конференциях покрывают все возможности языка, в том числе те, которые не закладывали создатели.

Тележная Ольга,
разработчица, Python lover,
активный контрибьютор в Git
github.com/telezhnaya

Глава 1

Структуры данных и алгоритмы

Python предоставляет широкий спектр встроенных структур данных, таких как списки, множества и словари. Использовать эти структуры по большей части просто. Однако нередко возникают вопросы, касающиеся поиска, сортировки, упорядочивания порядка элементов и фильтрации. Цель этой главы – обсудить распространенные структуры данных и алгоритмы. Также будет дано введение в структуры данных из модуля *collections*.

1.1. Распаковка последовательности в отдельные переменные

Задача

У вас есть кортеж из N элементов или последовательность, которую вы хотите распаковать в коллекцию из N переменных.

Решение

Любая последовательность (или итерируемый объект) может быть распакована в переменные с помощью простого присваивания. Единственное обязательное условие заключается в том, чтобы количество и структура переменных совпадали с таковыми у последовательности. Например:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = ['ACME', 50, 91.1, (2012, 12, 21)]
>>> name, shares, price, date = data
>>> name
```

```
'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>
```

При несовпадении количества элементов вы получите ошибку. Например:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

Обсуждение

Распаковка работает с любым итерируемым объектом, а не только с кортежами и списками. Это строки, файлы, итераторы и генераторы. Например:

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

При распаковке вы иногда можете захотеть отбраковать некоторые значения. Специального синтаксиса для этого в Python нет, но вы можете назначить переменные, которые потом отбросите. Например:

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

Но убедитесь, что вы уже не использовали где-то эту переменную.

1.2. Распаковка элементов из последовательностей произвольной длины

Задача

Вам нужно распаковать N элементов из итерируемого объекта, но этот объект может содержать больше N элементов, что вызывает исключение «too many values to unpack» («слишком много значений для распаковки»).

Решение

Для решения этой задачи можно использовать «выражения со звездочкой». Предположим, например, что вы ведете учебный курс. В конце семестра вы решаете, что не будете принимать во внимание оценки за первое и последнее домашние задания, а по остальным оценкам посчитаете среднее значение. Если у вас было четыре задания, то можно просто распаковать все четыре. Но что делать, если их 24? Выражение со звездочкой позволит легко решить эту задачу:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

Рассмотрим еще один пример: предположим, что у вас есть записи о пользователях, которые состоят из имени и адреса электронной почты, за которыми следует произвольное количество телефонных номеров. Вы можете распаковать записи так:

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

Стоит отметить, что переменная *phone_numbers* всегда будет списком – несмотря на то, сколько телефонных номеров распаковано (даже если и ни одного). Любому коду, который будет использовать переменную *phone_numbers*, не нужно учитывать возможность того, что в ней будет не список (и производить дополнительные проверки на предмет этого).

Переменная со звездочкой также может быть первой в списке. Например, у вас есть последовательность значений, представляющая продажи вашей компании за последние восемь кварталов. Если вы хотите посмотреть, как последний квартал соотносится со средним значением по первым семи, вы можете сделать так:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

Интерпретатор Python выдаст:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

Обсуждение

Расширенная распаковка отлично подходит для распаковки итерируемых объектов неизвестной или произвольной длины. Часто эти объекты имеют некоторые известные элементы или шаблоны (например, «все, что после элемента 1, является телефонным номером»). Распаковка со звездочкой позволяет программисту легко использовать эти шаблоны – вместо того чтобы исполнять акробатические трюки для извлечения нужных элементов из итерируемого объекта.

Стоит отметить, что синтаксис звездочки может быть особенно полезен при итерировании по последовательности кортежей переменной длины. Например, возможна такая последовательность кортежей с тегами:

```
records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

Распаковка со звездочкой также может быть полезна в комбинации с операциями обработки строк, такими как разрезание. Например:

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

Иногда вам может быть нужно распаковать значения и отбросить их. Вы не можете просто определить голую * при распаковке, но можно использовать обычное для отбрасывания имя переменной, такое как _ или *ign* (ignored). Например:

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

Есть некоторое сходство между распаковкой со звездочкой и обработкой списков в функциональных языках. Например, если у вас есть список, то вы можете легко разделить его на «хвост» и «голову»:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

Можно представить себе функцию, которая произведет такое разрезание с помощью хитрого рекурсивного алгоритма. Например:

```
>>> def sum(items):
...   head, *tail = items
...   return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

Однако вам следует знать, что рекурсия не относится к числу сильных сторон Python из-за внутреннего лимита на нее. Поэтому последний пример на практике оказывается просто любопытным предметом для размышления.

1.3. Оставляем N последних элементов

Задача

Вы хотите хранить ограниченную историю из нескольких последних элементов, полученных в ходе итерации или какого-то другого процесса обработки данных.

Решение

Задача хранения ограниченной истории – отличный повод применить collections.deque. Например, следующий фрагмент кода производит простое сопоставление текста с последовательностью строк, а при совпадении выдает совпавшие строки вместе с N предыдущими строками контекста:

```
from collections import deque
def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
```

```
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Пример использования
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-'*20)
```

Обсуждение

При написании программы для поиска элементов обычно используют функцию-генератор, содержащую *yield* (как и показано в вышеприведенном примере). Это отделяет процесс поиска от кода, который использует результаты. Если вы новичок в обращении с генераторами, см. **рецепт 4.3**.

Использование *deque(maxlen=N)* создает очередь фиксированной длины. Когда новые элементы добавлены и очередь заполнена, самый старый элемент автоматически удаляется. Пример:

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

Хотя вы можете вручную производить такие операции над списком (то есть добавление в конец, удаление и т. п.), очередь элегантнее и работает намного быстрее.

Обобщим: *deque* может быть использована в любом случае, когда вам нужна простая очередь. Если вы не зададите максимальную длину, то получите бесконечную очередь, которая позволит вам добавлять и удалять элементы с обоих концов. Например:

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
```

```
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

Добавление или удаление элементов в любой из концов очереди имеет сложность O(1). А вот вставка или удаление элемента в начале списка имеет сложность O(N).

1.4. Поиск N максимальных и минимальных элементов

Задача

Вы хотите создать список N максимальных или минимальных элементов коллекции.

Решение

У модуля *heapq* есть две функции, *nlargest()* и *nsmallest()*, которые делают именно то, что вам нужно. Например:

```
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums))  # Выведет [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Выведет [-4, 1, 2]
```

Обе функции также принимают параметр *key*, который позволяет использовать их с более сложными структурами данных. Например:

```
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]
cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

Обсуждение

Если вы ищете N наименьших или N наибольших элементов, причем N невелико, по сравнению с общим размером коллекции, эти функции покажут великолепную производительность. «Под капотом» они начинают работу с конвертирования данных в список, где данные упорядочены, как в куче. Например:

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
```

```
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

Самая важная возможность кучи состоит в том, что *heap[0]* всегда будет наименьшим элементом. Кроме того, последующие элементы могут быть легко найдены с помощью метода *heapq.heappop()*, который удаляет первый элемент и заменяет его следующим наименьшим элементом (это требует O(log N) операций, где N – размер кучи). Например, чтобы найти три наименьших элемента, вы могли бы сделать так:

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

Функции *nlargest()* и *nsmallest()* лучше всего подходят, если вы пытаетесь найти относительно небольшое количество элементов. Если же вы просто хотите найти один наибольший или наименьший элемент (N = 1), функции *min()* и *max()* будут быстрее. Похожим образом, если N сопоставимо с размером самой коллекции, обычно будет быстрее отсортировать их и взять срез (то есть выполнить *sorted(items)[:N]* или *sorted(items)[-N:]*). Стоит отметить, что реализация *nlargest()* и *nsmallest()* в модуле *heapq* работает гибко и выполняет некоторые из этих оптимизаций самостоятельно (например, использует сортировку, если размер N близок к размеру входных данных).

Хотя использовать этот рецепт необязательно, реализация кучи интересна и заслуживает изучения. Информацию о ней можно найти в любой приличной книге по алгоритмам и структурам данных. В документации модуля *heapq* также обсуждаются детали внутренней реализации.

1.5. Реализация очереди с приоритетом

Задача

Вы хотите реализовать очередь, которая сортирует элементы по заданному приоритету и всегда возвращает элемент с наивысшим приоритетом при каждой операции получения (удаления) элемента.

Решение

Приведенный ниже класс использует модуль *heapq* для реализации простой очереди с приоритетом.

```
import heapq

class PriorityQueue:
    def __init__(self):
```

```
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
            return heapq.heappop(self._queue)[-1]
```

А вот пример использования:

```
>>> class Item:
...   def __init__(self, name):
...       self.name = name
...   def __repr__(self):
...       return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

Первая операция *pop()* возвращает элемент с наивысшим приоритетом. Также заметьте, что два элемента с одинаковым приоритетом (*foo* и *grok*) были возвращены в том же порядке, в каком они были помещены в очередь.

Обсуждение

Суть этого рецепта – использование модуля *heapq*. Функции *heapq.heappush()* и *heapq.heappop()* вставляют и удаляют элементы из *list_queue* таким образом, что первый элемент в списке имеет наименьший приоритет (как обсуждалось в **рецепте 1.4**). Метод *heappop()* всегда возвращает «наименьший» элемент, что является ключом к тому, чтобы заставить очередь удалять правильные элементы. Кроме того, так как операции вставки и удаления имеют сложность O(log N), где N – число элементов в куче, то они вполне эффективны даже для весьма больших значений N.

В этом рецепте очередь состоит из кортежей формата *(-priority, index, item)*. Значение *priority* сделано отрицательным, чтобы заставить очередь сортировать элементы от наибольшего к наименьшему приоритету. Это противоположно обычному порядку сортировки кучи (от наименьшего к наибольшему значению).

Роль переменной *index* заключается в установлении правильного порядка элементов с одинаковым приоритетом. Поддержание постоянно увеличивающегося индекса позволяет сортировать элементы в соответствии с порядком, в каком они были вставлены. Однако индекс также играет важную роль в выполнении операций сравнения при работе с элементами с одинаковыми значениями приоритета.

Если остановиться на этом подробнее, то отметим, что экземпляры класса *Item* не могут быть упорядочены. Например:

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Если вы создаете кортежи *(priority, item)*, то их можно сравнивать до тех пор, пока приоритеты различны. Однако же если сравниваются два кортежа с равными приоритетами, то сравнение не может быть проведено (как и ранее). Например:

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Вводя дополнительный индекс и создавая кортежи *(priority, index, item)*, вы полностью обходите эту проблему, поскольку два кортежа никогда не будут иметь одинаковые значения переменной *index* (и Python никогда не будет сравнивать остальные значения в кортежах, если результат сравнения уже определен):

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

Если вы хотите использовать эту очередь для коммуникации между потоками, то должны добавить правильную блокировку и передачу сигналов (см. **рецепт 12.3**).

Документация модуля *heapq* содержит дополнительные примеры и обсуждения теории и реализации куч[1].

[1] https://docs.python.org/3.4/library/heapq.html.

1.6. Отображение ключей на несколько значений в словаре

Задача

Вы хотите создать словарь, который отображает ключи на более чем одно значение (так называемый «мультисловарь», multidict).

Решение

Словарь – это отображение, где каждый ключ отображен на единственное значение. Если вы хотите отобразить ключи на множественные значения, вам нужно хранить эти множественные значения в контейнере, таком как список или множество. Например, вы можете создавать такие словари:

```
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}

e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

Выбор, использовать либо не использовать списки или множества, зависит от того, как будет использован мультисловарь. Применяйте список, если вы хотите сохранить порядок, в котором добавлены элементы. Применяйте множество, если вы хотите устранить дубликаты (и при этом не беспокоиться о порядке элементов).

Чтобы легко создавать такие словари, вы можете использовать *defaultdict* из модуля *collections*. Особенность *defautdict* заключается в автоматической инициализации первого значения, так что вы можете сосредоточиться на добавлении элементов. Например:

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...
```

Одно предупреждение: *defaultdict* автоматически создаст записи словаря для ключей, к которым позже будет осуществлен доступ (даже если их в данный момент в словаре нет). Если такое поведение нежелательно, вы можете использовать *setdefault()* на обычном словаре. Например:

```
d = {}  # Обычный словарь
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
...
```

Однако многие программисты находят *setdefault()* несколько неестественным – и это если не учитывать тот факт, что он всегда создает новый экземпляр первоначального значения при каждом вызове (в примере это пустой список []).

Обсуждение

Конструирование словарей со множественными значениями не является чем-то сложным. Однако инициализация первого значения может быть запутанной, если вы пытаетесь сделать это самостоятельно. Например, вы можете написать что-то такое:

```
d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)
```

Использование *defaultdict* приводит к намного более чистому коду:

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

Этот рецепт связан с проблемой группировки записей в задачах обработки данных. Посмотрите, например, **рецепт 1.15**.

1.7. Поддержание порядка в словарях

Задача

Вы хотите создать словарь, который позволит контролировать порядок элементов при итерировании по нему или при сериализации.

Решение

Чтобы контролировать порядок элементов в словаре, вы можете использовать *OrderedDict* из модуля *collections*. При итерировании он в точности сохраняет изначальный порядок добавления данных. Например:

```
 from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4
```

```
# Выведет "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

OrderedDict особенно полезен, когда вы хотите создать отображение, которое в дальнейшем собираетесь сериализовать или закодировать в другой формат. Например, если вы хотите строго контролировать порядок полей, выводимых в формате JSON, вам нужно просто создать *OrderedDict* с нужными данными:

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

Обсуждение

OrderedDict внутри себя поддерживает двусвязный список, который упорядочивает ключи в соответствии с порядком добавления. Когда новый элемент вставляется впервые, он помещается в конец этого списка. Последующее связывание значения с существующим ключом не изменяет порядок.

Заметьте, что размер *OrderedDict* более чем в два раза превышает размер обычного словаря из-за содержащегося внутри дополнительного списка. А если вы собираетесь создать структуру данных, в которой будет большое число экземпляров *OrderedDict* (например, вы хотите прочитать 100 000 строк CSV-файла в список экземпляров *OrderedDict*), вам стоит изучить требования вашего приложения, чтобы решить, перевесят ли преимущества использования *OrderedDict* затраты на дополнительную память.

1.8. Вычисления со словарями

Задача

Вы хотите проводить различные вычисления (например, поиск минимального и максимального значений, сортировку) на словаре с данными.

Решение

Рассмотрим словарь, который отображает тикеры (идентификаторы акций на бирже) на цены:

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

Чтобы выполнить вычисления на содержимом словаря, часто бывает полезно обратить ключи и значения, используя функцию *zip()*. Например, вот так можно найти минимальную и максимальную цены, а также соответствующий тикер:

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price - (10.75, 'FB')

max_price = max(zip(prices.values(), prices.keys()))
# max_price - (612.78, 'AAPL')
```

Похожим образом для ранжирования данных можно использовать *zip()* с *sorted()*, как показано ниже:

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted - [(10.75, 'FB'), (37.2, 'HPQ'),
#                  (45.23, 'ACME'), (205.55, 'IBM'),
#                  (612.78, 'AAPL')]
```

Когда вы производите эти вычисления, обратите внимание, что *zip()* создает итератор, по которому можно пройти только один раз. Например, следующий фрагмент кода – неправильный:

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names))  # OK
print(max(prices_and_names))  # ValueError: max() arg is an empty sequence
```

Обсуждение

Если вы попытаетесь выполнить обычные функции обработки данных на словаре, то обнаружите, что они обрабатывают лишь ключи, но не значения. Например:

```
min(prices)  # Вернет 'AAPL'
max(prices)  # Вернет 'IBM'
```

Вероятно, это не то, чего вы хотели добиться, поскольку пытались выполнить вычисления с использованием значений словаря. Вы можете попробовать исправить это, используя метод словаря *values()*:

```
min(prices.values())  # Вернет 10.75
max(prices.values())  # Вернет 612.78
```

К несчастью, в большинстве случаев это не то, чего вы хотите. Например, вам нужно знать соответствующие ключи (т. е. понять, у каких акций самая низкая цена).

Вы можете получить ключ, соответствующий минимальному или максимальному значению, если передадите функцию в функции *min()* и *max()*. Например:

```
min(prices, key=lambda k: prices[k])  # Вернет 'FB'
max(prices, key=lambda k: prices[k])  # Вернет 'AAPL'
```

Однако чтобы получить минимальное значение, вам потребуется дополнительное обращение. Например:

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

Использование функции *zip()* решает задачу путем «обращения» словаря в последовательность пар (value, key). Когда выполняется сравнение таких кортежей, элемент *value* сравнивается первым, а *key* – следующим. Это дает вам то самое поведение, которое вы хотите, а также позволяет проводить обработки и сортировку словаря с использованием единственного выражения.

Стоит отметить, что в вычислениях с использованием пар *(value, key)* будет применен *key*, чтобы определить результат в экземплярах, где множественные записи имеют одинаковые *value*. Например, в вычислениях *min()* и *max()* запись с наименьшим или наибольшим ключом будет возвращена, если найдутся одинаковые значения. Например:

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

1.9. Поиск общих элементов в двух словарях

Задача

У вас два словаря, и вы хотите выяснить, что у них общего (одинаковые ключи, значения и т. п.).

Решение

Рассмотрим два словаря:

```
a = {
    'x' : 1,
    'y' : 2,
    'z' : 3
}

b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

Чтобы найти общие элементы, просто выполните обычный набор операций с использованием методов *keys()* и *items()*. Например:

```
# Находим общие ключи
a.keys() & b.keys()  # { 'x', 'y' }
# Находим ключи, которые есть в a, но которых нет в b
a.keys() - b.keys()  # { 'z' }
# Находим общие пары (key,value)
a.items() & b.items() # { ('y', 2) }
```

Операции такого типа также могут быть использованы для изменения или фильтрования содержимого словаря. Предположим, например, что вы хотите создать новый словарь, в котором некоторые ключи удалены. Взгляните на этот пример кода генератора словаря (dictionary comprehension):

```
# Создаем новый словарь, из которого удалены некоторые ключи
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c - это {'x': 1, 'y': 2}
```

Обсуждение

Словарь – это отображение множества ключей на множество значений. Метод словаря *keys()* возвращает объект ключей словаря (dict_keys). Малоизвестная особенность этих объектов заключается в том, что они поддерживают набор операций над множествами: объединения, пересечения, разности. Так что если вам нужно выполнить этот набор операций над ключами словаря, вы можете использовать объект ключей словаря напрямую, без предварительного конвертирования во множество.

Метод словаря *items()* возвращает объект значений словаря, состоящий из пар (key, value). Этот объект поддерживает похожий набор операций и может быть использован для выполнения таких операций, как поиск того, какие пары ключ–значение являются общими для двух словарей.

Хотя метод словаря *values()* похож на предыдущие, он не поддерживает операции над множествами, описанные выше в этом рецепте. Это происходит, в частности, по причине того, что, в отличие от ключей, значения могут и не быть уникальными. Один этот факт делает бесполезным применение к ним операций над множествами. Если же, однако, вы вынуждены выполнить такие операции, этого можно добиться простым путем предварительного конвертирования значений во множество.

1.10. Удаление дубликатов из последовательности с сохранением порядка элементов

Задача

Вы хотите исключить дублирующиеся значения из последовательности, но при этом сохранить порядок следования оставшихся элементов.

Решение

Если значения в последовательности являются хешируемыми, задача может быть легко решена с использованием множества и генератора. Например:

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

Вот пример использования этой функции:

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

Это будет работать только в том случае, если элементы последовательности хешируются. Если вы пытаетесь удалить дубликаты в последовательности из нехешируемых типов (таких как словари), то можете внести небольшое изменение в этот рецепт. Например, такое:

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

Аргумент *key* здесь нужен для определения функции, которая конвертирует элементы последовательности в хешируемый тип, подходящий для поиска дубликатов. Вот как это работает:

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

Последнее решение также отлично работает, если вам нужно удалить дубликаты, базируясь на значении одного поля или атрибута либо более крупной структуры данных.

Обсуждение

Если вы просто хотите удалить дубликаты, то часто достаточно будет создать множество. Например:

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

Однако этот поход не сохраняет какой бы то ни было порядок, поэтому результат будет перемешан. Показанные выше решения помогают избежать этого.

Использование функции-генератора в этом рецепте отражает тот факт, что вы наверняка хотите написать функцию максимально широкого назначения, а не напрямую привязанную к обработке списков. Например, если вы хотите читать файл, удаляя дублирующиеся строки, то можете сделать так:

```
with open(somefile,'r') as f:
    for line in dedupe(f):
        ...
```

Передача функции в аргументе *key* имитирует похожую возможность во встроенных функциях, таких как *sorted()*, *min()* и *max()*. См., например, **рецепты 1.8** и **1.13**.

1.11. Присваивание имен срезам

Задача

Ваша программа превратилась в нечитабельную массу индексов срезов, и вы хотите все это расчистить.

Решение

Предположим, что у вас есть код, который вытаскивает определенные поля с данными из строковых записей с фиксированным набором полей (т. е. из файла с плоской структурой или похожего формата):

```
###### 0123456789012345678901234567890123456789012345678901234567890'
record = '.....................100 .......513.25 ..........'
cost = int(record[20:32]) * float(record[40:48])
```

Вместо этого вы вполне можете присвоить срезам имена:

```
SHARES = slice(20,32)
PRICE  = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

В последнем примере вы избежали появления кучи загадочных индексов, и код стал проще и яснее.

Обсуждение

Общее правило таково: написание кода с большим количеством неоформленных индексов ведет к проблемам с читабельностью и поддерживаемостью. Например, если вы вернетесь к такому коду через год, то наверняка не сразу вспомните, как и о чем вы думали, когда все это писали. Приведенное выше решение – простой путь к более ясному обозначению того, что делает ваш код.

Встроенная функция *slice()* создает объект среза, который может быть использован везде, где применяются обычные срезы. Например:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

Если у вас есть экземпляр *slice*, сохраненный в переменной *s*, вы можете получить больше информации о нем, если посмотрите на атрибуты *s.start*, *s.stop* и *s.step*. Например:

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
>>> a.step
2
>>>
```

Также вы можете наложить срез на последовательность определенного размера, используя его метод *indices(size)*. Он возвращает кортеж *(start, stop, step)*, где все значения соответственно ограничены, чтобы вписаться в границы (дабы избежать возбуждения исключений *IndexError* при индексировании). Например:

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...   print(s[i])
...
W
r
d
>>>
```

1.12. Определение наиболее часто встречающихся элементов в последовательности

Проблема

У вас есть последовательность элементов, и вы хотите узнать, какие элементы встречаются в ней чаще всего.

Решение

Класс *collections.Counter* разработан как раз для решения подобных задач. В нем даже есть удобный метод *most_common()*, который сразу выдаст вам ответ.

Чтобы проиллюстрировать это, предположим, что у вас есть список слов, и вы хотите найти наиболее часто встречающееся. Вот как можно это сделать:

```
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]

from collections import Counter

word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Выведет [('eyes', 8), ('the', 5), ('look', 4)]
```

Обсуждение

На входе объектам класса *Counter* можно скормить любую последовательность хешируемых элементов. В основе *Counter* лежит словарь, который отображает количество вхождений элементов. Например:

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

Если вы хотите увеличить счет вручную, используйте сложение:

```
>>> morewords = ['why','are','you','not','looking','in','my','eyes']
>>> for word in morewords:
...   word_counts[word] += 1
...
>>> word_counts['eyes']
9
>>>
```

Или же вы можете использовать метод *update()*:

```
>>> word_counts.update(morewords)
>>>
```

Малоизвестная возможность экземпляров *Counter* состоит в том, что они могут быть легко скомбинированы с использованием разнообразных математических операций. Например:

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
        "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
        'my': 1, 'why': 1})

>>> # Объединяем счетчики
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
        'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
        'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Вычитаем счетчики
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
        "you're": 1, "don't": 1, 'under': 1})
>>>
```

Нет смысла упоминать, что объекты *Counter* – невероятно полезный инструмент для практически любых задач, где вам нужно перевести данные в табличную форму и посчитать их. Рекомендуем использовать этот способ, а не писать вручную решения на основе словарей.

1.13. Сортировка списка словарей по общему ключу

Задача

У вас есть список словарей, и вы хотите отсортировать записи согласно одному или более значениям.

Решение

Сортировка структур этого типа легко выполняется с помощью функции *itemgetter* из модуля *operator*. Предположим, вы выполнили запрос к таблице базы данных, чтобы получить список зарегистрированных пользователей вашего сайта, и получили в ответ вот такую структуру данных:

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

Можно достаточно легко вывести эти строки с упорядочением по любому из полей, общих для всех словарей. Например:

```
from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

Вышеприведенный код выведет следующее:

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

Функция *itemgetter()* также может принимать несколько ключей. Пример кода:

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)
```

Вышеприведенный код выведет:

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

Обсуждение

В этом примере строки передаются встроенной функции *sorted()*, которая принимает именованный аргумент *key*. Этот аргумент должен быть вызываемым объектом, который принимает один элемент из *rows* и возвращает значение, которое будет использовано в качестве основы для сортировки. Функция *itemgetter()* создает такой вызываемый объект. (Вызываемый объект – это объект, который имеет метод __call__. – *Прим. перев.*)

Функция *operator.itemgetter()* принимает в качестве аргументов индексы, которые используются для извлечения нужных значений из записей в *rows*. Это может быть ключ словаря, номер элемента в списке или любое другое значение, которое может быть скормлено методу *__getitem__()*. Если вы передадите несколько индексов функции *itemgetter()*, вызываемый объект, который она создаст, вернет кортеж со всеми элементами, и функция *sorted()* упорядочит выводимые элементы в соответствии с отсортированным порядком кортежей. Это может быть полезно, если вы хотите провести сортировку сразу по нескольким полям (в примере это имя и фамилия).

Функциональность *itemgetter()* иногда может быть заменена *lambda*-выражением. Например:

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'],r['fname']))
```

Это решение в большинстве случаев работает отлично. Однако решение с использованием *itemgetter()* обычно выполняется быстрее. Так что обратите на него внимание, если производительность в приоритете.

Последнее по порядку, но не по значению: не забудьте, что описанная в этом рецепте техника может быть применена к таким функциям, как *min()* и *max()*. Например:

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14. Сортировка объектов, не поддерживающих сравнение

Задача

Вы хотите отсортировать объекты одного класса, но они не поддерживают операции сравнения.

Решение

Встроенная функция *sorted()* принимает аргумент *key*, в котором может быть передан вызываемый объект, который будет возвращать некоторое значение из объектов, которое *sorted()* будет использовать для сравнения этих объектов. Например, если у вас в приложении есть последовательность экземпляров класса *User*

и вы хотите отсортировать их по атрибуту *user_id*, то вы могли бы предоставить вызываемый объект, который принимает экземпляр класса *User* и возвращает атрибут *user_id*. Например:

```
>>> class User:
...  def __init__(self, user_id):
...      self.user_id = user_id
...  def __repr__(self):
...      return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
[User(3), User(23), User(99)]
>>>
```

Вместо *lambda* можно применить альтернативный подход с использованием *operator.attrgetter()*:

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

Обсуждение

Использовать или не использовать *lambda* или *attrgetter()* – вопрос личных предпочтений. Однако *attrgetter()* часто оказывается немного быстрее, а также позволяет одновременно извлекать несколько полей. Это аналогично использованию *operator.itemgetter()* для словарей (см. **рецепт 1.13**). Например, если экземпляры класса User также имеют атрибуты *first_name* и *last_name*, вы можете выполнить вот такую сортировку:

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

Стоит отметить, что использованный в этом рецепте прием может быть применен к таким функциям, как *min()* и *max()*. Например:

```
>>> min(users, key=attrgetter('user_id')
User(3)
>>> max(users, key=attrgetter('user_id')
User(99)
>>>
```

1.15. Группирование записей на основе полей

Задача

У вас есть последовательность словарей или экземпляров, и вы хотите итерировать по данным, сгруппированным по значению конкретного поля (например, по дате).

Решение

Функция *itertools.groupby()* особенно полезна для такого типа группирования данных. Предположим, что у вас есть список словарей:

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

Предположим также, что вы хотите проитерировать по группам данных, объединенных общей датой. Проведем сортировку по нужному полю (в данном случае по дате), а потом применим *itertools.groupby()*:

```
from operator import itemgetter
from itertools import groupby

# Сначала сортируем по нужным полям
rows.sort(key=itemgetter('date'))

# Итерируем в группах
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

Вывод будет таким:

```
07/01/2012
    {'date': '07/01/2012', 'address': '5412 N CLARK'}
    {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
    {'date': '07/02/2012', 'address': '5800 E 58TH'}
    {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
    {'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
    {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
    {'date': '07/04/2012', 'address': '5148 N CLARK'}
    {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

Обсуждение

Функция *groupby()* работает так: сканирует последовательность и ищет последовательные «партии» одинаковых значений (или значений, возвращенных переданной через *key* функцией). В каждой итерации функция возвращает значение вместе с итератором, который выводит все элементы в группу с одинаковым значением.

Важным предварительным шагом тут является сортировка данных по интересующему нас полю. Поскольку *groupby()* проверяет только последовательные элементы, без предварительной сортировки группировка записей выполнена не будет.

Если ваша цель – просто сгруппировать данные вместе в крупную структуру данных с произвольным доступом, то вам больше поможет функция *defaultdict()*, которая создает «мультисловарь», как было описано в **рецепте 1.6**. Например:

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

Это позволяет легко получить доступ к записям для каждой даты:

```
>>> for r in rows_by_date['07/01/2012']:
...   print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

В последнем примере предварительная сортировка записей не обязательна. Но если вы не заботитесь о потреблении памяти, то может оказаться быстрее сделать это с помощью предварительной сортировки и итерирования с использованием *groupby()*.

1.16. Фильтрование элементов последовательности

Задача

У вас есть данные внутри последовательности, и вы хотите извлечь значения или сократить последовательность по какому-либо критерию.

Решение

Самый простой способ фильтрования последовательности – использовать генератор списка (list comprehension). Например:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

Потенциальная проблема с использованием генераторов списков заключается в том, что они могут создать большой результат, если размер входных данных тоже большой. Если это вас беспокоит, вы можете использовать выражения-генераторы для итеративного возврата отфильтрованных значений. Например:

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
```

```
>>> for x in pos:
...   print(x)
...
1
4
10
2
3
>>>
```

Иногда критерий фильтрования не может быть легко выражен в форме генератора списка или выражения-генератора. Предположим, например, что процесс фильтрования включает обработку исключений или какой-то другой сложный момент. Чтобы справиться с этим, поместите фильтрующий код в функцию и используйте встроенную функцию *filter()*. Например:

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']

def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False

ivals = list(filter(is_int, values))
print(ivals)
# Выведет ['1', '2', '-3', '4', '5']
```

filter() создает итератор, так что если вы хотите получить список результатов, не забудьте использовать *list()*, как показано выше.

Обсуждение

Генераторы списков и выражения-генераторы часто являются самым легким и прямым способом фильтрования простых данных. Но у них также есть дополнительная способность – одновременно изменять данные. Например:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

Одна из разновидностей фильтрования включает замену значений, которые не подходят под определенный критерий, другими значениями (вместо отбраковки неподходящих). Например, вместо простого поиска положительных значений вы также хотите обрезать «плохие» значения, чтобы они попадали в определенный диапазон. В большинстве случаев это легко сделать с помощью перемещения критерия фильтрования в условное выражение:

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
```

```
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

Другой важный инструмент для фильтрации – *itertools.compress()*, который принимает итерируемый объект вместе с последовательностью-селектором из булевых значений. На выходе функция выдает все элементы итерируемого объекта, для которых совпадающий элемент в селекторе – True. Это может быть полезно, если вы пытаетесь применить результаты фильтрования одной последовательности к другой связанной последовательности. Например, у вас есть две колонки данных:

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK',
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

Теперь предположим, что вы хотите создать список всех адресов, где соответствующие значения из *counts* больше 5. Вот как это можно сделать:

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

Ключевой момент – сначала создать последовательность булевых значений, которые будут указывать, какие элементы удовлетворяют заданному условию. Далее функция *compress()* выберет элементы, соответствующие значениям True.

Как и *filter()*, функция *compress()* возвращает итератор. Поэтому если вы хотите на выходе получить список, вам придется использовать *list()*.

1.17. Извлечение подмножества из словаря

Задача

Вы хотите создать словарь, который будет подмножеством другого словаря.

Решение

Эту задачу можно легко решить с помощью генератора словаря (dictionary comprehension). Например:

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}

# Создать словарь всех акций с ценами больше 200
p1 = { key:value for key, value in prices.items() if value > 200 }

# Создать словарь акций технологических компаний
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:value for key,value in prices.items() if key in tech_names }
```

Обсуждение

Большую часть того, что можно сделать с помощью генераторов словарей, можно осуществить путем создания последовательности кортежей и передачи их в функцию *dict()*. Например:

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

Однако решение на основе генератора словаря яснее и работает немного быстрее (в рассмотренном выше примере генератор отработал в два раза быстрее).

Иногда существует множество путей решить задачу. К слову, второй пример можно переписать так:

```
# Создать словарь акций технологических компаний
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

Однако подсчет времени выполнения открывает нам, что это решение почти в 1,6 раза медленнее, чем первое. Если производительность для вас важна, обычно стоит потратить немного времени на изучение таких вопросов. См. **рецепт 14.13**, чтобы получить детальную информацию о подсчете времени и профилировании.

1.18. Отображение имен на последовательность элементов

Задача

У вас есть код, который осуществляет доступ к элементам в списке или кортеже по позиции. Однако такой подход часто делает программу нечитабельной. Также вы можете захотеть уменьшить зависимость от позиции в структуре данных путем перехода к принципу доступа к элементам по имени.

Решение

collections.namedtuple() предоставляет такую возможность, добавляя лишь минимальные затраты по сравнению с использованием обычного кортежа. *collections.namedtuple()* – это фабричный метод, который возвращает подкласс стандартного типа Python *tuple* (кортеж). Вы скармливаете этому методу имя типа и поля, которые он должен иметь. Он возвращает класс, который может порождать экземпля-

ры с полями, вами определенными, а также значениями этих полей, которые вы передадите при создании. Например:

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

Хотя экземпляр *namedtuple* (именованного кортежа) выглядит так же, как и обычный экземпляр класса, он взаимозаменяем с кортежем и поддерживает все обычные операции кортежей, такие как индексирование и распаковка. Например:

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

Самый частый случай использования именованного кортежа – отвязка вашего кода от работы с позициями элементов, которыми он манипулирует. Скажем, если вы получаете большой список кортежей в ответ на запрос к базе данных, а потом манипулируете ими через позиционное обращение к элементам, ваш код может сломаться, если вы, например, добавите новую колонку в таблицу. Этого можно избежать, если вы сначала превратите полученные кортежи в именованные кортежи.

Чтобы проиллюстрировать это, приведем пример кода, использующего обычные кортежи:

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

Использование позиционного обращения к элементам часто делает код немного менее выразительным и более зависимым от структуры записей. А вот версия с использованием именованного кортежа:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

Естественно, вы можете избежать явной конвертации в именованный кортеж *Stock*, если последовательность *records* из примера уже содержит такие экземпляры.

Обсуждение

Возможное использование именованного кортежа – замена словаря, который требует больше места для хранения. Так что если создаете крупные структуры данных с использованием словарей, применение именованных кортежей будет более эффективным. Однако не забудьте, что именованные кортежи неизменяемы (в отличие от словарей). Например:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Если вам нужно изменить любой из атрибутов, это может быть сделано с помощью метода *_replace()*, которым обладают экземпляры именованных кортежей. Он создает полностью новый именованный кортеж, в котором указанные значения заменены. Например:

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

Тонкость использования метода *_replace()* заключается в том, что он может стать удобным способом наполнить значениями именованный кортеж, у которого есть опциональные или отсутствующие поля. Чтобы сделать это, создайте прототип кортежа, содержащий значения по умолчанию, а затем применяйте *_replace()* для создания новых экземпляров с замененными значениями. Например:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Создание экземпляра прототипа
stock_prototype = Stock('', 0, 0.0, None, None)

# Функция для преобразования словаря в Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

Вот пример работы этого кода:

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

Последнее, но важное замечание: стоит отметить, что если вашей целью является создание эффективной структуры данных, которая позволяет менять различные атрибуты экземпляров, использование именованных кортежей – не лучший вариант. Вместо них стоит определить класс с использованием __slots__ (см. **рецепт 8.4**).

1.19. Одновременное преобразование и сокращение (свертка) данных

Задача

Вам нужно выполнить функцию сокращения (т. е. sum(), min(), max()), но сначала необходимо преобразовать или отфильтровать данные.

Решение

Есть весьма элегантное решение для объединения сокращения (свертки) и преобразования данных – выражение-генератор в аргументе. Например, если вы хотите подсчитать сумму квадратов, попробуйте следующее:

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

Вот еще несколько примеров:

```
# Определяем, есть ли файлы .py в каталоге
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')

# Выводит кортеж как CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))

# Сокращение (reduction) данных по полям в структуре данных
portfolio = [
    {'name':'GOOG', 'shares': 50},
    {'name':'YHOO', 'shares': 75},
    {'name':'AOL', 'shares': 20},
    {'name':'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)
```

Обсуждение

Решение демонстрирует тонкий синтаксический аспект выражений-генераторов, связанный с передачей их как единственного аргумента в функцию: повторяющиеся скобки не нужны. Например, следующие инструкции эквивалентны:

```
s = sum((x * x for x in nums))  # Передаем выражение-генератор
                                # в качестве аргумента
s = sum(x * x for x in nums)    # Более элегантный синтаксис
```

Использование аргумента-генератора часто будет более эффективным и элегантным, нежели предварительное создание временного списка. Например, если вы не используете выражение-генератор, то можете склониться к такой реализации:

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])
```

Это работает, но вводит лишний шаг и создает лишний список. Для небольшого списка из примера это не имеет значения, но если nums был огромным, вы получите крупную временную структуру данных, которая будет использована только один раз, а потом выброшена. Решение с генератором обрабатывает данные итеративно и потому намного более эффективно с точки зрения использования памяти.

Некоторые функции сокращения (свертки), такие как *min()* и *max()*, принимают аргумент *key*, что может оказаться полезным в ситуациях, когда вы склоняетесь к использованию генератора. Например, в этом примере вы можете попробовать альтернативный подход:

```
# Изначальный: возвращает 20
min_shares = min(s['shares'] for s in portfolio)

# Альтернативный: возвращает {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20. Объединение нескольких отображений в одно

Задача

У вас есть много словарей или отображений, которые вы хотите логически объединить в одно отображение, чтобы выполнить некоторые операции, такие как поиск значений или проверка существования ключей.

Решение

Предположим, у вас есть два словаря:

```
a = {'x': 1, 'z': 3 }
b = {'y': 2, 'z': 4 }
```

А теперь предположим, что вы хотите провести поиски, в ходе которых вам нужно проверить оба словаря (то есть сначала проверить в словаре *a*, а потом в *b*, если в первом словаре искомое не найдено). Простой способ сделать это – использовать класс *ChainMap* из модуля *collections*. Например:

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x'])  # Выводит 1 (из a)
print(c['y'])  # Выводит 2 (из b)
print(c['z'])  # Выводит 3 (из a)
```

Обсуждение

ChainMap принимает несколько отображений и делает их логически единым целым. Однако в буквальном смысле они не сливаются. Вместо этого *ChainMap* просто содержит список отображений и переопределяет обычные операции над словарями для сканирования данного списка. Большинство операций работает. Например:

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

В случае появления одинаковых ключей будут использованы значения из первого словаря. Например, *c['z']* в примере всегда будет ссылаться на значение из словаря *a*, а не из *b*.

Операции, которые изменяют отображение, всегда действуют на первое отображение в списке. Например:

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

ChainMap особенно полезны для работы со значениями, принадлежащими областям видимости, такими как переменные языка программирования (глобальные, локальные и т. п.). На самом деле даже существуют методы, которые все упрощают:

```
>>> values = ChainMap()
>>> values['x'] = 1
>>> # Добавляем новое отображение
>>> values = values.new_child()
>>> values['x'] = 2
>>> # Добавляем новое отображение
>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Удаляем последнее отображение
>>> values = values.parents
>>> values['x']
```

```
2
>>> # Удаляем последнее отображение
>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>
```

В качестве альтернативы *ChainMap* вы можете обдумать слияние словарей с использованием метода update(). Например:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

Это работает, но требует от вас создания полностью нового объекта словаря (или необратимого изменения одного из существующих). В этом случае при изменении одного из первоначальных словарей изменения не затронут новый объект объединенного словаря. Например:

```
>>> a['x'] = 13
>>> merged['x']
1
```

ChainMap использует первоначальные словари, поэтому не подвержен такому поведению. Например:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x']  # Изменение происходит и в объединенных словарях
42
>>>
```