Чарльз Р. Северанс

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Чарльз Р. Северанс

Python для всех

Python for Everybody

Exploring Data Using Python 3

Dr. Charles R. Severance

Python для всех

Обработка данных с использованием Python 3

Чарльз Р. Северанс

Москва, 2022

УДК 004.94
ББК **32.972**
С28

Северанс Ч. Р.
С28 Python для всех / пер. с англ. А. В. Снастина. – М.: ДМК Пресс, 2021. – 262 с.: ил.

ISBN 978-5-93700-104-7

Данная книга поможет освоить с нуля язык программирования Python и перейти к решению практических задач анализа данных.

Краткий и четкий стиль изложения позволяет быстро усвоить основные принципы программирования. Ознакомившись с базовыми функциональными свойствами языка Python, читатель перейдет к изучению тонких приемов его применения. В многочисленных примерах и упражнениях показана реализация часто применяемых алгоритмов, шаблонов программирования и разнообразных структур данных. Особое внимание уделяется методам обработки сетевых данных и взаимодействия с реляционными базами данных. Почти в каждой главе приводятся полезные советы по отладке программ – обнаружению и исправлению ошибок.

Издание предназначено для широкого круга читателей, которые, не являясь профессиональными программистами, тем не менее хотели бы освоить Python и использовать его в своей области деятельности.

УДК 004.94
ББК 32.972

ISBN 978-1-530-05112-0 (англ.)
ISBN 978-5-93700-104-7 (рус.)

Содержание

От издательства

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 на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и 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 на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Другая редакция книги под свободной лицензией

Для преподавателей и научных сотрудников вузов, которые постоянно твердят «публикуйся или исчезни», вполне естественным является желание создавать с нуля собственные оригинальные творения. Эта книга – эксперимент, но не начатый с нуля, а представляющий собой «другую редакцию» книги «Think Python: How to Think Like a Computer Scientist», написанной Алленом Б. Дауни (Allen B. Downey), Джеффом Элкнером (Jeff Elkner) и другими авторами.

В декабре 2009 г. я готовился к преподаванию курса SI502 – Networked Programming в Мичиганском университете в пятом семестре подряд и решил, что наступило время написать учебник по языку Python, в котором главное внимание было бы сосредоточено на обработке данных, а не на понимании алгоритмов и абстракций. Моя цель в курсе SI502 – научить людей навыкам обработки данных с использованием Python, которые оставались бы полезными в течение всей жизни. Лишь некоторые из моих студентов планировали стать профессиональными программистами. Но другие предполагали работать библиотекарями, менеджерами, юристами, биологами, экономистами и т. д., которые способны умело использовать (информационные) технологии в выбранной ими области.

Мне никогда не удавалось найти совершенную книгу, ориентированную на обработку данных на языке Python, для своего курса, поэтому я просто решил написать такую книгу. К счастью, за три недели до того, как я начал работу над своей новой книгой с нуля, на собрании преподавателей факультета доктор Атул Пракаш (Atul Prakash) показал мне книгу «Think Python», которую он использовал при чтении своего курса по языку Python в этом семестре. Оказалось, что это отлично написанный текстовый материал по информационным технологиям, уделяющий главное внимание коротким четким описаниям и легкий для изучения.

Общая структура этой книги была изменена для перехода к решению задач анализа данных как можно быстрее, и в нее был включен ряд работающих примеров и упражнений по анализу данных для самых начинающих.

Главы 2–10 похожи на книгу «Think Python», но и в них были внесены существенные изменения. Примеры и упражнения на вычисления были заменены на упражнения, ориентированные на обработку данных. Темы представлены в порядке, необходимом для последовательного формирования решений по анализу данных с постепенно увеличивающейся сложностью. Некоторые темы, такие как применение `try` и `except`, вынесены в более ранний материал и представлены как часть главы об условных конструкциях. Функциям уделено весьма немного внимания до тех пор, пока они не становятся необходимыми для снижения сложности программы, т. е. они не описываются как ранний пример абстракции. Почти все функции, определяемые пользователем, удалены из исходных кодов примеров и упражнений,

встречающихся до главы 4. Слово «рекурсия» вообще не появляется в этой книге (разумеется, за исключением этой строки).

В главах 1 и 11–16 весь материал принципиально новый, сфокусированный на использовании в реальной жизни методов анализа данных, приведены соответствующие примеры на Python, включая регулярные выражения для поиска и синтаксического разбора (парсинга), автоматизации задач на вашем компьютере, извлечения данных из сетевой среды, скрейпинг вебстраниц для получения данных, объектно-ориентированного программирования, использования веб-сервисов, синтаксического разбора (парсинга) данных в форматах XML и JSON, создания и использования баз данных с применением языка Structured Query Language (SQL) и визуализации данных.

Самая главная цель этих изменений – смещение фокуса с компьютерных технологий на информатику и включение в первый курс по компьютерным технологиям только тех тем, которые могут быть полезными, даже если студент не намерен становиться профессиональным программистом.

Студенты, которые найдут эту книгу интересной и захотят узнать больше, должны обратиться к книге «Think Python» Аллена Б. Дауни. Поскольку между этими двумя книгами очень много общего, студенты быстро овладеют навыками и умениями в дополнительных областях технического программирования и алгоритмического мышления, которые подробно рассматриваются в книге «Think Python». А с учетом того, что книги написаны в похожем стиле, изучение «Think Python» потребует лишь минимальных усилий.

Как обладатель прав на «Think Python» Аллен разрешил мне изменить лицензию, касающуюся материала из его книги, который включен в мою книгу, с GNU Free Documentation License на более новую лицензию Creative Commons Attribution – Share Alike. Это является следствием всеобщего перехода на лицензии, защищающие свободно распространяемую (открытую) документацию, – от лицензии GFDL к лицензии CC-BY-SA (например, «Википедия»). Использование лицензии CC-BY-SA сохраняет строгие традиции книжного «копилефта», но при этом лицензия становится еще более понятной для новых авторов при повторном использовании этого материала, если они находят его подходящим для своих работ.

Я считаю, что эта книга послужит примером, объясняющим, почему свободно распространяемые материалы так важны для будущего образования, и хочу поблагодарить Аллена Б. Дауни и издательство Cambridge University Press за их дальновидное решение сделать эту книгу доступной под открытым авторским правом. Надеюсь, что они довольны результатами моих усилий, а также надеюсь, что вам, моим читателям, понравится наш совместный труд.

Я хотел бы поблагодарить Аллена Б. Дауни (Allen B. Downey) и Лорен Каулз (Lauren Cowles) за их помощь, терпение и руководство в совместной работе и решении вопросов об авторских правах, касающихся этой книги.

Чарльз Северанс (Charles Severance)
www.dr-chuck.com
Анн-Арбор (шт. Мичиган), США
9 сентября 2013 г.

Чарльз Северанс является практикующим адъюнкт-профессором в Информационной школе Мичиганского университета.

Глава 1

Почему вы должны учиться писать программы

Написание программ (или программирование) – это весьма креативная и плодотворная деятельность. Вы можете писать программы по многим причинам, посвятив всю свою жизнь решению трудных задач анализа данных, или получать удовольствие, помогая кому-нибудь другому решить трудную задачу. Эта книга предполагает, что каждый должен знать, как написать программу, а как только вы узнаете, как написать программу, то определите, что именно вы хотите сделать с помощью новоприобретенных умений и навыков.

В повседневной жизни мы окружены компьютерами со всех сторон – от ноутбуков до мобильных телефонов. Мы можем считать эти компьютеры своими «личными помощниками», которые могут позаботиться о многих вещах вместо нас. В сущности, аппаратура в современных компьютерах создана так, чтобы непрерывно спрашивать нас: «Что я должен делать дальше?»

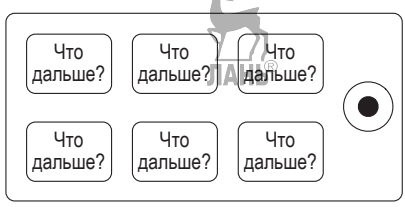


Рис. 1.1 ❖ Личный цифровой помощник

Программисты добавляют операционную систему и набор приложений к аппаратуре, и в итоге мы получаем личный цифровой помощник, который весьма полезен и способен помочь нам выполнить множество разнообразных дел.

Наши компьютеры быстры, обладают огромным объемом памяти и могли бы оказаться чрезвычайно полезными, если бы мы только знали язык, на котором могли бы объяснить компьютеру, что именно он «должен делать дальше». Если бы мы знали такой язык, то могли бы приказать компьютеру выполнять для нас многократно повторяющиеся (рутинные) задачи. Любопытно, что компьютеры лучше всего выполняют те задачи, которые нам, людям, кажутся скучными и утомительно-нудными.

Например, посмотрите на первые три абзаца этой главы и скажите, какое слово чаще всего используется в них и сколько раз это слово там встречается. Несмотря на то что вы способны читать и понимать слова за несколько секунд, их подсчет становится почти мучительной проблемой, потому что это не тот тип задач, для решения которых предназначен человеческий мозг. Для компьютера совсем наоборот – чтение и понимание текста с листа бумаги являются трудной задачей, но подсчет слов и вывод сообщения для нас – сколько раз самое часто встречающееся слово было использовано в заданном тексте – это очень простая задача:

```
python words.py
Enter file:words.txt
to 16
```

Наш «личный помощник по анализу информации» быстро сообщает, что слово «to» использовалось шестнадцать раз в первых трех абзацах этой главы.

Тот простой факт, что компьютеры лучше справляются с теми делами, в которых люди не преуспевают, является причиной, почему вам необходимо приобрести навыки общения на «языке компьютера». Как только вы изучите этот новый язык, то сразу сможете передать рутинные задачи своему партнеру (компьютеру), освобождая больше времени для тех дел, для которых вы лучше приспособлены. В такое партнерство вы привносите креативность, интуицию и изобретательность.

1.1. Креативность и мотивация

Хотя эта книга и не предназначена для профессиональных программистов, профессиональное программирование может быть чрезвычайно плодотворным делом как в финансовом, так и в личном плане. Создание полезных, изящных и интеллектуальных программ для использования другими людьми – это весьма креативная деятельность. Ваш компьютер или личный цифровой помощник (PDA – Personal Digital Assistant) обычно содержит множество разнообразных программ многих различных групп программистов, конкурирующих в борьбе за ваше внимание и заинтересованность. Они пытаются наилучшим образом удовлетворить ваши потребности и предоставить наибольшее удобство как пользователю в этом процессе. В некоторых ситуациях, когда вы выбираете часть программного обеспечения, программисты напрямую вознаграждаются благодаря вашему выбору.

Если считать программы креативным результатом работы группы программистов, то, возможно, на рис. 1.2 представлена более рациональная версия нашего личного цифрового помощника.

В данный момент наша главная мотивация – не зарабатывание денег и не удовлетворение потребностей конечных пользователей, а повышение эффективности при обработке данных и информации, которая встречается в повседневной жизни. В самом начале вы будете и программистом, и конечным пользователем своих программ. Повышая свое мастерство программис-

та, вы почувствуете, что программирование становится более креативным занятием, и тогда ваши мысли обратятся в сторону разработки программ для других людей.

Рис. 1.2 ❖ Программисты разговаривают с вами

1.2. Аппаратная архитектура компьютера

Прежде чем начать изучение языка, на котором передаются инструкции компьютерам для разработки программного обеспечения, необходимо немного познакомиться с тем, как устроены компьютеры. Если бы вы разобрали компьютер или мобильный телефон и заглянули поглубже внутрь, то увидели бы его составные части, показанные на рис. 1.3.

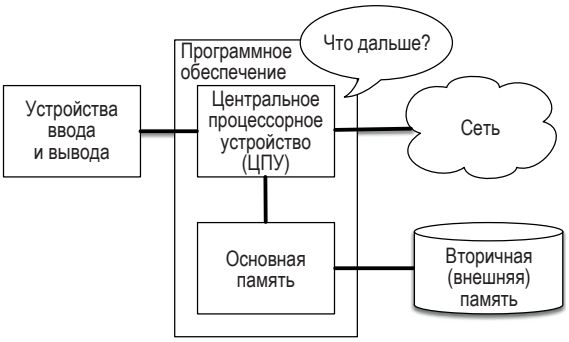


Рис. 1.3 ❖ Аппаратная архитектура

Ниже приведены определения высокого уровня этих составных частей:

- центральное процессорное устройство (ЦПУ), или просто центральный процессор (ЦП) (Central Processing Unit – CPU) – это та часть компьютера, которая одержима вопросом «что дальше?». Если ваш компьютер определен как имеющий тактовую частоту 3.0 ГГц, это значит, что ЦПУ будет спрашивать «что дальше?» три миллиарда раз в секунду. Вы должны научиться говорить быстро, чтобы успевать за ЦПУ;
- основная память (main memory) используется для хранения информации, которая необходима ЦПУ в срочном порядке. Основная память почти так же быстра, как ЦПУ. Но информация, хранящаяся в основной памяти, бесследно исчезает при выключении компьютера;

- вторичная (внешняя) память (secondary memory) также используется для хранения информации, но она намного медленнее основной памяти. Преимущество вторичной памяти состоит в том, что она может хранить информацию, даже когда компьютер выключен. Примерами вторичной (внешней) памяти являются дисковые накопители и устройства флеш-памяти (обычно в составе USB-накопителей и мобильных музыкальных плееров);
- устройства ввода и вывода – это просто экран дисплея, клавиатура, мышь, микрофон, динамик, тачпад и т. п. Все это – способы взаимодействия с компьютером;
- в наше время большинство компьютеров также имеют устройство сетевого соединения (network connection) для получения информации по сети. Можно считать сеть очень медленным «устройством» для хранения и извлечения данных, которое, возможно, не всегда «готово к работе». Так что в некотором смысле сеть – это более медленная и временами ненадежная форма вторичной (внешней) памяти.

Большинство подробностей о том, как работают эти компоненты, лучше оставить производителям компьютеров, но все же полезно знать, что означают некоторые термины, чтобы мы могли говорить о некоторых таких составных частях компьютера при написании программ.

Ваша работа как программиста состоит в использовании и регулировании каждого из этих ресурсов для решения поставленной перед вами задачи и анализа данных, получаемых из этого решения. Как программист вы в основном будете «разговаривать» с ЦПУ и сообщать ему, что делать дальше. Иногда вы будете приказывать ЦПУ использовать основную память, вторичную (внешнюю) память, сеть или устройства ввода/вывода.

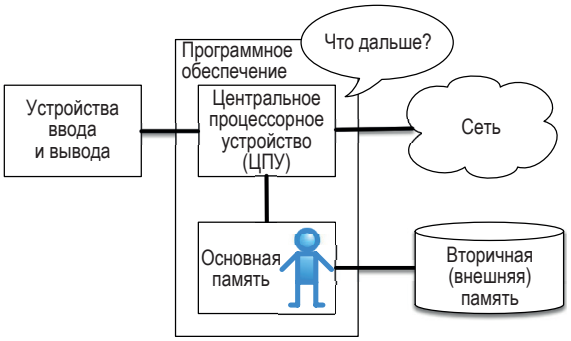


Рис. 1.4 ❖ Где находитесь вы

Вы должны быть тем человеком, который отвечает на вопрос ЦПУ «что дальше?». Но уменьшение размеров вашего тела до 5 мм с последующим перемещением внутрь компьютера, чтобы вы смогли подавать три миллиарда команд в секунду, может оказаться чрезвычайно неудобным. Так что вместо этого вы должны написать свои инструкции заранее. Мы называем эти сохраненные инструкции программой, а запись инструкций и обеспечение их корректности – программированием.

1.3. Изучение программирования

В остальной части книги мы попытаемся превратить вас в такого человека, который хорошо владеет искусством программирования. После прочтения книги вы станете программистом, возможно, непрофессиональным, но, по крайней мере, вы получите навыки для определения задачи анализа данных/информации и разработки программы для решения этой задачи.

В той или иной степени вам необходимы два навыка (умения), чтобы стать программистом:

- во-первых, вы должны знать язык программирования (Python) – требуются знания словаря и грамматики. Вы обязаны правильно произносить (т. е. использовать) слова этого нового языка и знать, как составлять правильно сформулированные «предложения» на этом новом языке;
- во-вторых, необходимо умение «рассказывать истории». Записывая рассказ, вы объединяете слова и предложения, чтобы донести главную мысль до читателя. Существует умение и искусство составления рассказа, а умение написать рассказ улучшается постоянной практикой написания текстов и получением некоторой обратной связи. В программировании наша программа – это «рассказ», а задача, которую вы пытаетесь решить, – это «основная мысль».

После того как вы освоите один из языков программирования, например Python, вы обнаружите, что гораздо проще изучить второй язык, такой как JavaScript или C++. Новый язык программирования имеет совершенно другой словарь и грамматику, но умение решать задачи останется одним и тем же при использовании всех языков программирования.

Вы изучите «словарь» и «предложения» языка Python достаточно быстро. Больше времени займет приобретение способности написать логически связную программу для решения качественно новой задачи. Мы изучаем программирование во многом так же, как учимся писать. Начинаем с чтения и объяснения программ, затем пишем простые программы, после этого пишем программы, сложность которых постепенно увеличивается со временем. В некоторый момент вы чувствуете, что «нашли свою музу», знаете, как создать собственные шаблоны, и более естественным для вас становится понимание, как сформулировать задачу и написать программу для ее решения. Когда вы достигаете этого момента, программирование становится весьма приятным и креативным процессом.

Мы начинаем со словаря и структуры программы на языке Python. Будьте терпеливы, так как простые примеры напомнят вам то время, когда вы начинали учиться читать.

1.4. Слова и предложения

В отличие от человеческих языков, словарь Python действительно невелик. Этот словарь мы называем «зарезервированными словами» (reserved words). Эти слова обладают особым значением в языке Python. Когда Python видит

эти слова в программе, для него они имеют один и только один смысл. Позже, когда вы начнете писать программы, то будете создавать собственные слова, имеющие особый смысл лично для вас, – они называются переменными (variables). Вы получите большую свободу выбора имен для своих переменных, но в качестве таких имен не сможете использовать ни одно из зарезервированных слов языка Python.

Когда вы дрессируете собаку, то используете специальные слова, например «сидеть», «тубо» и «апорт». Когда вы говорите с собакой, но не используете какое-либо из этих зарезервированных слов, она просто вопросительно смотрит на ваше лицо до тех пор, пока вы не произнесете зарезервированное слово. Например, если вы говорите: «Хотел бы я, чтобы больше людей могли гулять для улучшения своего здоровья», – то большинство собак слышат: «бла, бла, бла, гулять, бла, бла, бла», потому что «гулять» – это зарезервированное слово на собачьем языке. Возможно, многие предполагают, что в языке общения между людьми и кошками нет таких зарезервированных слов[1].

Зарезервированные слова в языке, на котором люди разговаривают с Python, перечислены в табл. 1.1.

Таблица 1.1. Зарезервированные слова языка Python

and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	
class	finally	is	return	
continue	for	lambda	try	
def	from	nonlocal	while	

Вот и все слова, но, в отличие от собаки, Python уже полностью выдрессирован. Если вы говорите «try», то Python будет пытаться (выполнить определенное действие) каждый раз, когда вы произносите это слово без ошибки.

Мы выучим все эти зарезервированные слова и научимся правильно их использовать в нужное время, но сейчас необходимо сосредоточиться на аналоге «разговора» с Python (в человеческо-собачьем языке). В диалоге с Python хорошо то, что мы можем даже сообщить, что именно нужно сказать, передавая ему сообщение в (одиночных) кавычках:

```
print('Hello world!')
```

Мы только что написали свое первое синтаксически правильное предложение на языке Python. Это предложение начинается с функции `print`, за которой следует выбранная нами строка текста, заключенная в одиночные кавычки. Строки в инструкциях вывода всегда заключаются в кавычки. Одиночные и двойные кавычки выполняют одну и ту же функцию. Большинство людей используют одиночные кавычки, за исключением тех случаев, когда одиночная кавычка (которая также является символом апострофа) содержится в строке.

[1] http://xkcd.com/231/.

1.5. Диалог с Python

Теперь, когда нам известно слово и простое предложение в Python, необходимо узнать, как начать диалог с Python, чтобы проверить наши новые познания в языке.

Прежде чем начать диалог с Python, сначала необходимо установить соответствующее программное обеспечение на ваш компьютер и научиться запускать Python. Но это требует описания слишком многих подробностей в данной главе, поэтому я предлагаю обратиться на сайт www.py4e.com, где приведены подробные инструкции и деморолики по установке и запуску Python в системах Macintosh и Windows. В некоторый момент вы окажетесь в окне терминала или командной строки, где нужно будет ввести команду `python`, чтобы запустить интерпретатор языка Python, который выполняется в интерактивном режиме и выглядит приблизительно так:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Промпт (приглашение) >>> – это способ интерпретатора Python спросить вас «Что прикажете делать дальше?». Python готов начать диалог с вами. Все, что вам нужно знать, – как разговаривать на языке Python.

Например, предположим, что вы пока еще не знали даже самых простых слов и предложений языка Python. Вероятно, вы воспользуетесь стандартной фразой, которую применяют астронавты при посадке на далекой планете и попытке поговорить с ее обитателями:

```
>>> I come in peace, please take me to your leader
#   Я пришел с миром, пожалуйста, отведите меня к вашему предводителю.
File "<stdin>", line 1
I come in peace, please take me to your leader
     ^
SyntaxError: invalid syntax
>>>
```

Получилось не очень удачно. Если вы быстро не придумаете что-то еще, то обитатели этой планеты, вероятнее всего, проткнут вас копьями, насадят на вертел, поджарят и съедят на обед.

К счастью, в свои путешествия вы прихватили с собой экземпляр этой книги, открыли ее как раз на этой странице и попытались снова:

```
>>> print('Hello world!')
Hello world!
```

Это выглядит намного лучше, и вы пробуете продолжить общение:

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
# Должно быть, вы бог из легенд, пришедший с неба
```

```
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
# Мы очень долго ждали вас
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
# Наши легенды гласят, что вы будете очень вкусны с горчицей
>>> print 'We will have a feast tonight unless you say
# Мы будем пировать нынешней ночью, если вы не скажете
File "<stdin>", line 1
print 'We will have a feast tonight unless you say
                                                  ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

Диалог продолжался неплохо некоторое время, но затем вы сделали крошечную ошибку в использовании языка Python, и Python снова направил на вас копья.

В этот момент вы должны также понять: несмотря на то что Python чрезвычайно сложен и мощен, а также весьма требователен к синтаксису, используемому для общения с ним, тем не менее Python не обладает разумом. В действительности вы только что вели диалог с самим собой, но использовали корректный синтаксис.

В определенном смысле, когда вы используете программу, написанную кем-то другим, в диалоге между вами и этими другими программистами Python действует как посредник. Python предоставляет создателям программ способ, позволяющий определить, как предположительно происходит такой диалог. И уже в нескольких следующих главах вы станете одним из этих программистов, использующих Python для беседы с пользователями своей программы.

Прежде чем мы прекратим наш первый диалог с интерпретатором Python, вы должны узнать, как правильно попрощаться при контакте с обитателями планеты Python:

```
>>> good-bye
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
File "<stdin>", line 1
if you don't mind, I need to leave
         ^
SyntaxError: invalid syntax
>>> quit()
```

Обратите внимание: ошибки различны в первых двух неправильных попытках. Вторая ошибка отличается от первой, потому что `if` – зарезервированное слово, Python заметил его и посчитал, что мы пытаемся что-то сказать, но синтаксис этого предложения оказался некорректным.

Правильный способ попрощаться с Python – ввод `quit()` после интерактивного промпта >>>. Вероятно, вам потребовалось бы некоторое время, чтобы найти этот способ, так что наличие книги под рукой оказывается полезным.

1.6. Терминология: интерпретатор и компилятор

Python – язык высокого уровня, предназначенный для относительно простого чтения и записи команд человеком и чтения и обработки команд компьютерами. К другим языкам высокого уровня относятся Java, C++, PHP, Ruby, Basic, Perl, JavaScript и многие другие. В действительности аппаратура внутри центрального процессорного устройства (ЦПУ) не понимает ни один из этих языков.

ЦПУ понимает только язык, который мы называем машинным языком. Машинный язык очень прост и, откровенно говоря, весьма утомителен для записи, потому что представлен исключительно нулями и единицами:

```
00101000111010010010101000001111
11100110000011101010010101101101
...
```

Машинный язык на первый взгляд кажется чрезвычайно простым, потому что использует только нули и единицы, но его синтаксис даже сложнее и намного замысловатее, чем синтаксис Python. Поэтому лишь немногие программисты пишут на машинном языке. Вместо этого мы создаем разнообразные трансляторы, позволяющие программистам писать программы на языках высокого уровня, таких как Python или JavaScript, а эти трансляторы преобразуют программы в машинный язык для их реального выполнения ЦПУ.

Поскольку машинный язык напрямую связан с аппаратурой компьютера, он не является переносимым между различными типами аппаратного обеспечения. Программы, написанные на языках высокого уровня, можно переносить на различные компьютеры, используя соответствующий интерпретатор на новом компьютере или перекомпилируя исходный код для создания версии программы на машинном языке для нового компьютера.

Трансляторы языков программирования делятся на две общие категории: (1) интерпретаторы и (2) компиляторы.

Интерпретатор (interpreter) считывает исходный код программы в том виде, как он написан программистом, выполняет синтаксический анализ (парсинг) исходного кода и интерпретирует (выполняет) инструкции сразу же, «на лету». Python – это интерпретатор, и когда мы запускаем его в интерактивном режиме, то можем вводить строку на его языке, и Python немедленно обрабатывает и выполняет ее, после чего снова готов к приему другой строки, которую мы можем ввести.

Некоторые строки сообщают Python, что мы хотим запомнить некоторое значение на будущее. Необходимо выбрать имя для запоминания такого значения, и мы можем использовать это символьное имя для извлечения этого значения в дальнейшем. Мы применяем термин переменная (variable) для обозначения меток, используемых для обращения к таким сохраненным данным.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

В этом примере мы попросили Python запомнить числовое значение шесть и использовали метку `x`, чтобы можно было получить это значение позже. Мы проверили, действительно ли Python запомнил значение, воспользовавшись функцией `print`. Затем мы попросили Python извлечь значение `x` и умножить его на семь и запомнить новое вычисленное значение в `y`. После этого попросили Python вывести текущее значение, хранящееся в `y`.

Даже если мы вводим все эти команды в одной строке одновременно, Python определяет их как упорядоченную последовательность инструкций, при этом более поздние инструкции могут извлекать данные, созданные в более ранних инструкциях. Мы пишем наш первый простой абзац с четырьмя предложениями в логическом и разумном порядке.

Для интерпретатора вполне естественной является способность вести интерактивный диалог, как показано выше. Компилятор (compiler) требует обработки всей программы в целом в файле, затем он запускает процесс преобразования исходного кода на языке высокого уровня в машинный язык, после чего помещает полученный в результате машинный язык (машинный код) в файл для последующего выполнения.

Если вы пользуетесь системой Windows, то чаще всего эти выполняемые программы на машинном языке имеют суффикс (расширение) «.exe» или «.dll», обозначающие соответственно «executable» (выполняемый файл) и «dynamic link library» (динамически связываемая библиотека). В системах Linux и Macintosh нет специальных суффиксов (расширений), однозначно определяющих файл как выполняемый.

Если открыть выполняемый файл в текстовом редакторе, то его содержимое выглядит абсолютно бессмысленно и нечитабельно:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

Очень не просто читать или писать на машинном языке, но, к счастью, существуют интерпретаторы и компиляторы, позволяющие нам писать на языках высокого уровня, таких как Python или C.

В этот момент обсуждения компиляторов и интерпретаторов у вас должен возникнуть небольшой вопрос о самом интерпретаторе Python. На каком языке он написан? На компилируемом? Когда мы вводим команду `python`, что происходит на самом деле?

Интерпретатор Python написан на языке высокого уровня C. Исходный код интерпретатора Python можно посмотреть на сайте www.python.org и даже

самостоятельно поработать с этим исходным кодом. Так что Python сам по себе является программой, скомпилированной в машинный код. Когда вы устанавливаете Python на свой компьютер (или его устанавливает поставщик компьютера), то копируете экземпляр программы Python в машинном коде в свою систему. В Windows выполняемый машинный код интерпретатора Python, вероятнее всего, находится в файле с именем:

```
C:\Python35\python.exe
```

Это даже больше, чем вам в действительности необходимо знать, чтобы стать программистом на Python, но иногда стоит ответить на эти небольшие назойливые вопросы в самом начале.

1.7. Написание программы

Ввод команд в интерпретаторе Python – это отличный способ поэкспериментировать с функциональными возможностями этого языка, но такой подход не рекомендуется для решения более сложных задач.

Если нужно написать программу, то мы используем текстовый редактор, чтобы записать инструкции Python в файл, называемый скриптом (script). По принятому соглашению скрипты Python имеют имена с расширением `.py`.

Для выполнения скрипта вы должны сообщить интерпретатору Python имя соответствующего файла. В окне команд нужно ввести команду `python hello.py`, как показано ниже:

```
$ cat hello.py
print('Hello world!')
$ python hello.py
Hello world!
```

Символ «$» – это промпт (приглашение) операционной системы, а команда `cat hello.py` показывает нам, что файл *hello.py* содержит однострочную программу на языке Python для вывода строки.

Мы вызываем интерпретатор Python и приказываем прочитать этот исходный код из файла *hello.py* вместо вывода приглашения для ввода строк кода Python в интерактивном режиме.

Вы наверняка заметите, что нет необходимости вводить `quit()` в конце программы на языке Python, записанной в файле. Когда Python читает исходный код из файла, то знает, что нужно остановиться при достижении конца этого файла.

1.8. Что такое программа

Самое обобщенное определение программы – это последовательность инструкций на языке Python, предназначенная для выполнения каких-либо действий. Даже наш простой скрипт *hello.py* является программой. Эта одно-

строчная программа не слишком полезна, но по самому строгому определению это все же программа на языке Python.

Возможно, проще понять, что такое программа, если подумать о задаче, для решения которой может быть создана программа, а затем рассмотреть программу, которая должна решить эту задачу.

Например, предположим, что вы проводите исследование активности в соцсетях по постам в Фейсбуке и интересуетесь, какое слово чаще всего используется в некоторой последовательности постов. Вы могли бы вывести поток постов в Фейсбуке и сосредоточенно изучать их тексты в поисках наиболее часто употребляемого слова, но это отняло бы слишком много времени, а кроме того, очень высока вероятность сделать ошибку. Если бы вы были настолько умелыми, чтобы написать программу на Python для быстрого выполнения этой задачи, то наверняка смогли бы посвятить выходные чему-то более интересному.

Например, рассмотрим следующий текст о клоуне и автомобиле. Прочтите текст и определите самое часто используемое в нем слово и сколько раз оно встречается.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

А теперь представьте, что вы выполняете эту задачу, просматривая миллионы строк текста. Откровенно говоря, быстрее изучить Python и написать на нем программу подсчета слов, чем искать слова вручную.

Еще более приятная новость – я предлагаю вам уже готовую простую программу поиска наиболее часто встречающегося слова в текстовом файле. Я написал и протестировал ее и теперь передаю вам, чтобы вы использовали эту программу, сэкономив некоторое время.

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Исходный код: http://www.py4e.com/code3/words.py
```

Вам даже не нужно знать Python, чтобы пользоваться этой программой. Потребуется прочитать весь материал до главы 10 этой книги, чтобы полностью

понять те превосходные методики языка Python, которые использовались при написании данной программы. Сейчас вы – конечный пользователь, вы просто используете программу, восхищаетесь ее способностями и тем, что она избавила вас от тяжелого ручного труда. Просто введите этот исходный код в файл с именем *words.py* и запустите его или скачайте файл исходного кода здесь: http://www.py4e.com/code3/ – и запустите его.

Это хороший пример того, как интерпретатор и язык Python действуют в качестве посредника между вами (конечным пользователем) и мною (программистом). Для нас Python – способ обмена последовательностями полезных инструкций (т. е. программами) на общем языке, которым может пользоваться каждый, установивший Python на свой компьютер. Так что никто из нас не разговаривает с Python, вместо этого мы общаемся друг с другом через Python.

1.9. Структурные элементы программы

В нескольких следующих главах мы узнаем больше о словаре, структуре предложений, структуре абзаца и структуре рассказа Python. Мы познакомимся с мощными возможностями языка Python и со способами объединения этих возможностей для создания полезных программ.

Существуют некоторые концептуальные шаблоны низкого уровня, которые мы используем для создания программ. Эти конструкции присутствуют не только в программах на языке Python, они являются частью каждого языка программирования – от машинного языка до языков высокого уровня.

- Ввод (input) – получение данных из «внешнего мира». Это может быть чтение данных из файла или даже некоторый вид сенсорного устройства, например микрофон или GPS. В наших первых программах ввод будет выполняться пользователем, печатающим данные на клавиатуре.
- Вывод (output) – показ результатов выполнения программы на экране, или сохранение их в файле, или, возможно, передача в устройство, такое как динамик, для воспроизведения музыки или текста.
- Последовательное выполнение (sequential execution) – поочередное выполнение инструкций (одна за другой) в том порядке, в котором они записаны в скрипте.
- Условное выполнение (conditional execution) – проверка определенных условий, затем выполнение или пропуск некоторой последовательности инструкций.
- Повторяющееся выполнение (repeated execution) – многократное выполнение некоторой группы инструкций, обычно с некоторым изменением.
- Повторное использование (reuse) – однократная запись группы инструкций с присвоением ей имени, затем многократное использование этих инструкций, когда это необходимо в программе.

Это выглядит слишком просто, чтобы быть правдой, но, разумеется, эти конструкции никогда не бывают настолько простыми. Это похоже на фразу,

утверждающую, что прогулка – это просто «перемещение одной ноги перед другой». «Искусство» написания программы – это многократное объединение и сплетение этих базовых элементов для получения того, что окажется полезным для пользователей.

Приведенная выше программа подсчета слов использует все эти шаблоны, за исключением одного.

1.10. Что могло бы пойти не так

Как мы видели в наших предыдущих диалогах с Python, необходима абсолютно точная передача информации при вводе кода Python. Малейшее отклонение или ошибка приводит к тому, что Python отказывается рассматривать нашу программу.

Начинающие программисты часто приводят тот факт, что Python не оставляет возможности для совершения ошибок как доказательство того, что Python придирчивый, злобный и жестокий. Хотя Python для каждого выглядит по-разному, он выделяет таких программистов среди прочих и сохраняет свое недоброжелательное отношение к ним. Из-за такого недоброжелательного отношения Python принимает наши безупречно написанные программы или отвергает их как «непригодные» только для того, чтобы помучить нас.

```
>>> primt 'Hello world!'
File "<stdin>", line 1
  primt 'Hello world!'
                     ^
SyntaxError: invalid syntax
>>> primt ('Hello world')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined
>>> I hate you Python!
File "<stdin>", line 1
  I hate you Python!
       ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
File "<stdin>", line 1
  if you come out of there, I would teach you a lesson
           ^
SyntaxError: invalid syntax
>>>
```

В споре с Python почти ничего нельзя добиться. Это же просто инструмент. У него нет эмоций, он всегда доволен и готов служить вам, когда это необходимо. Его сообщения об ошибках выглядят неприятно, но это всего лишь просьба о помощи от Python. Он посмотрел, что вы набрали на клавиатуре, и просто не может понять, что же вы ввели.

Python очень похож на собаку, которая любит вас безгранично, понимает несколько ключевых слов, преданно смотрит вам прямо в глаза (>>>) и ждет, когда вы скажете то, что она понимает. Когда Python говорит «SyntaxError: invalid syntax», он просто виляет хвостом и как бы хочет сказать: «Похоже, ты что-то сказал мне, но я совсем не понимаю, что это значит, но, пожалуйста, поговори со мной еще (>>>)».

Когда ваши программы будут постепенно становиться более сложными, вы встретитесь с тремя основными типами ошибок:

- синтаксические ошибки (syntax errors) – это самые первые ошибки, которые вы будете совершать, и их проще всего исправить. Синтаксическая ошибка означает, что вы нарушили правила «грамматики» языка Python. Python делает все возможное, чтобы указать прямо на строку и символ, который вызвал его недоумение. Единственная сложность с синтаксическими ошибками заключается в том, что ошибка, требующая исправления, в действительности находится в программе раньше того места, в котором Python обнаружил непонятный для него символ. Поэтому строка и символ, на которые указывает Python в сообщении о синтаксической ошибке, могут оказаться лишь начальным пунктом для вашего расследования;
- логические ошибки (logic errors) – если ваша программа соблюдает правильный синтаксис, но инструкции расположены в неправильном порядке или имеется ошибка в том, как инструкции связаны друг с другом. Вот, возможно, неплохой пример логической ошибки: «выпейте воды из бутылки, положите бутылку в рюкзак, идите в библиотеку, затем закройте бутылку крышкой»;
- семантические (смысловые) ошибки (semantic errors) – смысловые ошибки возникают, когда ваше описание шагов выполнения синтаксически безупречно и расположено в верном порядке, но в программе просто есть ошибка. Программа абсолютно корректна, но она не делает то, что вы предполагали сделать с ее помощью. Можно привести такой простой пример – вы объясняете человеку дорогу в ресторан и говорите: «…когда дойдете до перекрестка с автозаправкой, поверните налево, пройдите милю и слева увидите красное здание – это ресторан». Поздно вечером ваш друг звонит вам и говорит, что их компания находится на ферме, бродит вокруг амбара и не видит никаких признаков ресторана. Тогда вы спрашиваете: «Так ты повернул налево или направо около автозаправки?», а он отвечает: «Я в точности следовал твоим указаниям, я записал их. Ты сказал – повернуть налево и пройти милю к автозаправке». Тогда вы говорите: «Мне очень жаль, потому что хотя мои инструкции были синтаксически корректными, к сожалению, они содержали небольшую, но необнаруженную семантическую ошибку».

Однако и в случаях обнаружения всех трех типов ошибок Python просто пытается сделать все возможное, чтобы выполнить именно то, что вы попросили.

1.11. Отладка

Если Python выдает сообщение об ошибке или даже если вы получили результат, отличающийся от предполагаемого, то начинается охота с целью выяснения причины ошибки. Отладка (debugging) – это процесс поиска причины ошибки в вашем исходном коде. При отладке программы, особенно если вы пытаетесь устранить трудно обнаруживаемую ошибку, следует попробовать четыре приема:

- чтение – внимательно исследуйте свой исходный код, читайте его снова и снова и следите за тем, чтобы в нем говорилось именно то, что вы хотели сказать;
- прогон (running) – экспериментируйте, внося изменения и запуская различные версии. Часто если вы наблюдаете правильный результат в нужном месте программы, то проблема становится очевидной, но иногда требуется некоторое время, чтобы создать вспомогательные средства;
- размышление – подумайте в течение некоторого времени. К какому типу относится ошибка – синтаксическая, времени выполнения, семантическая? Какую информацию можно получить из сообщений об ошибках или из вывода программы? Какой именно тип ошибки мог бы привести к наблюдаемой проблеме? Какое самое последнее изменение вы внесли, прежде чем возникла проблема?
- отступление назад – в некоторый момент самое лучшее, что можно сделать, – это отступать назад, отменяя недавно внесенные изменения до тех пор, пока не получите программу, которая работает, и вы понимаете, как она работает. Затем можно начать вносить изменения, снова продвигаясь вперед.

Начинающие программисты иногда зацикливаются на одном из этих приемов и забывают о других. Поиск трудно обнаруживаемой ошибки требует чтения кода, прогона, размышления и иногда отступления назад. Если вы зашли в тупик, применяя один из описанных выше приемов, то попробуйте другие. Каждому приему соответствует свойственный ему характер неисправности.

Например, чтение исходного кода может помочь, если возникла проблема из-за опечатки, но не из-за принципиального непонимания. Если вы не понимаете, что именно делает ваша программа, то можете прочесть код 100 раз, но никогда не найдете ошибку, потому что она в вашей голове.

Эксперименты с прогоном могут помочь, особенно если вы выполняете небольшие простые тесты. Но если вы экспериментируете с прогонами без размышлений и чтения кода, то можете скатиться до приема, который я называю «программированием методом случайного блуждания», т. е. до процесса внесения случайно выбранных изменений до тех пор, пока программа не начнет работать правильно. И без слов понятно, что программирование методом случайного блуждания может отнять много времени.

Необходимо выделить время для размышлений. Отладка похожа на экспериментальную науку. У вас должна быть хотя бы одна гипотеза (предпо-

ложение) о том, в чем заключается проблема. Если существуют два и более возможных вариантов, то попробуйте подумать о тесте, который исключит один из вариантов (или все, кроме одного).

Небольшой перерыв помогает продуктивному размышлению. Необходимо поговорить. Если объяснить возникшую проблему кому-то другому (или даже себе), то иногда ответ находится даже до того, как вы закончите задавать вопрос.

Но даже самые наилучшие методики отладки бессильны, если в программе слишком много ошибок или если код, который вы пытаетесь исправить, слишком велик и сложен. Иногда наилучшим вариантом становится отступление назад («откат к предыдущей версии»), упрощение программы до тех пор, пока не получится работающая версия, и вы понимаете, как она работает.

Начинающие программисты часто весьма неохотно отступают назад, потому что не могут смириться с удалением строки кода (даже если она неправильная). Если это успокоит вас, то скопируйте свою программу в другой файл, прежде чем начать удаление строк. Тогда вы сможете постепенно возвращать фрагменты кода в работающую программу.

1.12. Процесс обучения

При дальнейшем чтении этой книги не беспокойтесь, если излагаемые концепции и принципы не выглядят полностью согласованными друг с другом в первое время. Когда вы учились говорить, в первые несколько лет не возникало проблем из-за того, что вы издавали лишь забавное гуканье. И все было в норме, если за шесть месяцев вы переходили от простого словаря к простым предложениям, и еще пять-шесть лет требовалось для перехода от предложений к абзацам. А еще через несколько лет вы научились писать собственные интересные, полностью законченные короткие рассказы.

Надеюсь, мы будем изучать Python гораздо быстрее, поскольку обучимся всему вышеперечисленному во время прочтения нескольких следующих глав. Но это похоже на изучение нового (естественного) языка, когда требуется определенное время для восприятия и понимания, прежде чем вы почувствуете, что можете свободно говорить на нем. Это приводит к некоторой путанице, когда мы вновь и вновь возвращаемся к темам, стараясь позволить вам увидеть общую картину, пока мы определяем крошечные фрагменты, составляющие ее. Книга написана по линейной схеме, и если вы изучаете курс, то будете продвигаться по этой линейной схеме, но все же не стесняйтесь и становитесь «весьма нелинейными», свободно перемещаясь между различными частями книги. Заглядывайте вперед и возвращайтесь назад, читайте без напряжения. Бегло просматривая более сложный материал без полного понимания подробностей, вы сможете лучше понять ответ на вопрос «почему?» о программировании. Просматривая ранее изученный материал и даже повторно выполняя ранее завершенные упражнения, вы обнаружите, что действительно усвоили огромный объем знаний, даже если в текущий момент упорно штудируете материал, выглядящий слегка непостижимым.

Обычно при изучении самого первого языка программирования возникает несколько восхитительных моментов озарений «Ага!», в которые вы способны отколоть фрагмент каменной глыбы с помощью молотка и зубила, отойти на шаг и увидеть, что вы, несомненно, создаете прекрасную скульптуру.

Если что-то кажется особенно трудным, то обычно нет никакого смысла просиживать над этим ночами, не отрывая взгляда. Сделайте перерыв, вздремните, перекусите, объясните, в чем заключается ваша проблема, кому-нибудь (возможно, даже своей собаке), а затем возвращайтесь к решению проблемы со свежей головой. Заверяю вас, что как только вы изучите концепции и принципы программирования по этой книге, то, оглянувшись назад, вы поймете, что все было действительно просто и понятно, требовалось только лишь немного времени, чтобы усвоить эти знания.

1.13. Словарь терминов

- Ошибка, «баг» (bug) – ошибка в программе.
- Центральное процессорное устройство (ЦПУ) (central processor unit – CPU) – сердце любого компьютера. Это устройство, которое выполняет программы, которые мы пишем. Его также называют центральным процессором (ЦП) или просто процессором.
- Компиляция (compile) – перевод (преобразование) всей программы в целом, написанной на языке высокого уровня, на язык низкого уровня (машинный язык), чтобы подготовить программу для последующего выполнения.
- Язык высокого уровня (high-level language) – язык программирования, например Python, предназначенный для того, чтобы людям легче было читать и писать исходный код программ.
- Интерактивный режим (interactive mode) – способ использования интерпретатора языка Python для ввода команд и выражений в текстовой строке после промпта (приглашения).
- Интерпретация (interpret) – выполнение программы на языке высокого уровня с помощью последовательного преобразования (перевода) строк исходного кода (по одной строке за шаг выполнения).
- Язык низкого уровня (low-level language) – язык программирования, предназначенный для выполнения компьютером в упрощенном виде; его также называют «машинным кодом» или «языком ассемблера»[1].
- Машинный код (machine code) – язык самого низкого уровня для программного обеспечения, который непосредственно выполняется центральным процессорным устройством (ЦПУ).
- Основная память (main memory) – хранит программы и данные. При выключении компьютера основная память теряет всю хранимую в ней информацию.

[1] Здесь автор не совсем точен: машинный код состоит из нулей и единиц, а в языке ассемблера используются символьные мнемонические коды (хотя он действительно является языком низкого уровня). – *Прим. перев.*

- Синтаксический анализ (парсинг) (parse) – исследование программы и анализ ее синтаксической структуры.
- Переносимость (portability) – свойство программы выполняться на разнообразных типах компьютеров.
- Функция вывода `print` – инструкция, позволяющая интерпретатору Python выводить значения на экран.
- Решение задачи (problem solving) – процесс формулирования задачи, поиска ее решения и оформления решения в конечном виде.
- Программа (program) – набор инструкций, которые определяют процесс вычисления.
- Промпт (приглашение) (prompt) – появляется, когда программа выводит некоторое сообщение и делает паузу, чтобы пользователь мог ввести некоторые данные в программу.
- Вторичная (внешняя) память (secondary memory) – хранит программы и данные и сохраняет эту информацию даже после выключения компьютера. Как правило, медленнее основной памяти. Примеры вторичной памяти: дисковые накопители и флеш-память на USB-накопителях.
- Семантика (semantics) – смысл программы.
- Семантическая (смысловая) ошибка (semantic error) – ошибка в программе, заставляющая ее делать что-то, не предусмотренное программистом.
- Исходный код (source code) – программа, записанная на языке высокого уровня.

1.14. Упражнения

УПРАЖНЕНИЕ 1 Какую функцию выполняет вторичная (внешняя) память в компьютере?

a) Выполняет все вычисления и логику программы.
b) Извлекает веб-страницы из интернета.
c) Сохраняет информацию в течение длительного времени, даже после выключения компьютера.
d) Принимает входные данные от пользователя.

УПРАЖНЕНИЕ 2 Что такое программа?

УПРАЖНЕНИЕ 3 В чем различие между компилятором и интерпретатором?

УПРАЖНЕНИЕ 4 Какой из следующих объектов содержит машинный код?

a) Интерпретатор языка Python.
b) Клавиатура.
c) Файл исходного кода на языке Python.
d) Документ программы обработки текста.

УПРАЖНЕНИЕ 5 Что неправильно в следующем коде:

```
>>> primt 'Hello world!'
File "<stdin>", line 1
```

```
  primt 'Hello world!'
                    ^
SyntaxError: invalid syntax
>>>
```

УПРАЖНЕНИЕ 6 В какой части компьютера сохраняется переменная, например «x», после того как Python выполнит следующую строку (инструкцию)?

```
x = 123
```

a) В центральном процессорном устройстве.
b) В основной памяти.
c) Во вторичной памяти.
d) В устройствах ввода.
e) В устройствах вывода.

УПРАЖНЕНИЕ 7 Что выведет следующая программа:

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) Сообщение об ошибке, потому что x = x + 1 невозможно с точки зрения математики.

УПРАЖНЕНИЕ 8 Опишите каждую из перечисленных ниже частей компьютера, используя пример человеческих возможностей:

1) центральное процессорное устройство;
2) основная память;
3) вторичная память;
4) устройство ввода;
5) устройство вывода.

Например: «что у человека является аналогом центрального процессорного устройства?»

УПРАЖНЕНИЕ 9 Как исправить синтаксическую ошибку «Syntax Error»?

Глава 2

Переменные, выражения и инструкции

2.1. Значения и типы

Значение (value) – это одно из базовых понятий, с которыми работает программа, как буква или число. До сих пор мы встречались с такими значениями, как 1, 2 и «Hello, World!».

Эти значения принадлежат к различным типам (types): 2 – целое число (integer), а «Hello, World!» – строка (string), названная так, потому что содержит последовательность букв[1]. Вы (и интерпретатор) можете идентифицировать строки, потому что они заключены в кавычки.

Инструкция `print` также работает и с целыми числами. Мы используем команду `python` для запуска интерпретатора.

```
python
>>> print(4)
4
```

Если вы не уверены в том, к какому типу относится значение, то интерпретатор может подсказать вам.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Неудивительно, что строки принадлежат к типу `str`, а целые числа – к типу `int`. Менее очевидно, что числа с десятичной точкой принадлежат к типу `float`, потому что такие числа представлены в специальном формате с плавающей точкой (floating point).

---

[1] В английском языке (в частности, в программировании и обработке данных) различаются термины string (вереница, ряд, серия) – последовательность символов и line – строка (текста, кода программы) как единое целое. В русском языке термин один – строка. – *Прим. перев.*

```
>>> type(3.2)
<class 'float'>
```

А что можно сказать о таких значениях, как `"17"` и `"3.2"`? Они выглядят как числа, но заключены в кавычки как строки.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

И все-таки это строки.

При вводе большого целого числа может появиться соблазн использовать запятые между группами из трех цифр, например 1,000,000. Такая форма записи не является правильным целым числом в Python, тем не менее она допустима:

```
>>> print(1,000,000)
1 0 0
```

Ну, это совсем не то, чего мы ожидали. Python интерпретировал 1,000,000 как разделенную запятыми последовательность целых чисел и вывел их с разделением пробелами.

Это первый наблюдаемый нами пример семантической (смысловой) ошибки: код выполняется без каких-либо сообщений об ошибках, но не делает «правильные» вещи.

2.2. Переменные

Одной из самых мощных функциональных возможностей любого языка программирования является способность работать с переменными. Переменная (variable) – это имя, которое ссылается (указывает) на некоторое значение.

Инструкция присваивания (assignment statement) создает новые переменные и связывает с ними значения:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

В этом примере выполняются три присваивания. Первое присваивает строку новой переменной с именем `message`. Второе присваивает целое число 17 переменной `n`. Третье присваивает (приблизительное) значение числа π переменной `pi`.

Для вывода значения переменной можно воспользоваться инструкцией `print`:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

Тип переменной – это тип значения, на которое она указывает.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3. Имена переменных и ключевые слова

Программисты (почти) всегда выбирают для переменных имена, которые являются осмысленными и объясняющими, для чего предназначена конкретная переменная.

Имена переменных могут иметь произвольную длину. Они могут содержать буквы и цифры, но не могут начинаться с цифр. Допускается использование букв в верхнем регистре, но лучше все-таки начинать имя переменной с буквы в нижнем регистре (позже вы узнаете, почему).

В имя можно включать символ подчеркивания (_). Его часто используют в именах, состоящих из нескольких слов, таких как `my_name` или `airspeed_of_unladen_swallow`. Имена переменных могут начинаться с символа подчеркивания, но в общем случае мы так делать не будем, если только не пишем код библиотеки для других пользователей.

Если вы даете переменной недопустимое имя, то получите сообщение о синтаксической ошибке:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

Имя `76trombones` недопустимо, потому что начинается с числа. Имя `more@` недопустимо, потому что содержит неразрешенный символ @. А что не так с именем `class`?

Оказывается, что `class` – это одно из ключевых слов (keywords) языка Python. Интерпретатор использует ключевые слова для распознавания структуры программы, поэтому их нельзя использовать как имена переменных.

В языке Python зарезервировано 35 ключевых слов, перечисленных в табл. 2.1.

Таблица 2.1. Ключевые слова языка Python

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

Возможно, потребуется всегда держать этот список под рукой. Если интерпретатор жалуется на одно из имен ваших переменных и вы не понимаете, почему, то посмотрите, не находится ли имя в этом списке.

2.4. Инструкции

Инструкция (statement) – это элемент кода, который интерпретатор Python может выполнить. Мы уже видели два типа инструкций: вывод (`print`), являющийся инструкцией-выражением, и присваивание.

Когда вы вводите инструкцию в интерактивном режиме, интерпретатор выполняет ее и выводит результат, если есть что выводить.

Скрипт обычно содержит последовательность инструкций. Если имеется более одной инструкции, то их результаты появляются поочередно, по мере выполнения инструкций.

Например, скрипт


```
print(1)
x = 2
print(x)
```

выводит

```
1
2
```

Инструкция присваивания не выводит никакого результата.

2.5. Операторы и операнды

Операторы (operators) – это специальные символы, представляющие вычислительные операции, например сложение и умножение. Значения, к которым применяется оператор, называются операндами (operands).

Операторы +, –, *, / и ** выполняют соответственно сложение, вычитание, умножение, деление и возведение в степень, как показано в следующих примерах:


```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

Действие оператора деления изменилось в версии Python 3.x по сравнению с версией Python 2.x. В версии Python 3.x результатом такого деления является число с плавающей точкой:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

Оператор деления в версии Python 2.0 должен разделить два целых числа и выполнить усечение результата до целого числа:

```
>>> minute = 59
>>> minute/60
0
```

Чтобы получить такой результат в версии Python 3.0, используйте оператор целочисленного деления с округлением в меньшую сторону (//).

```
>>> minute = 59
>>> minute//60
0
```


В версии Python 3.0 результаты операций целочисленного деления в большей степени похожи на те, что ожидаются при вводе такого же выражения в калькулятор.

2.6. Выражения

Выражение (expression) – это совокупность значений, переменных и операторов. Все значения и переменные сами по себе считаются выражениями, поэтому все приведенные ниже строки кода являются допустимыми выражениями (предполагается, что переменной x было присвоено значение):


```
17
x
x + 17
```

Если вы вводите выражение в интерактивном режиме, то интерпретатор вычисляет его и выводит результат:

```
>>> 1 + 1
2
```

Но в скрипте выражение само по себе ничего не делает. Для начинающих зачастую это становится обескураживающим фактом.

УПРАЖНЕНИЕ 1 Введите следующие инструкции в интерпретаторе Python и понаблюдайте, что происходит:

```
5
x = 5
x + 1
```

2.7. Порядок выполнения операций

Если в выражении содержится более одного оператора, то порядок вычислений зависит от правил приоритета (rules of precedence). Для математических операций Python соблюдает математические соглашения о приоритетах. Английский акроним PEMDAS[1] является удобным способом запоминания этих правил:

- скобки (Parentheses) имеют наивысший приоритет и могут использоваться для определения необходимого вам порядка вычислений. Поскольку выражения в скобках вычисляются в первую очередь, `2 * (3-1)` равно `4`, а `(1+1)**(5-2)` равно `8`. Кроме того, можно использовать скобки для того, чтобы выражения легче читались, например `(minute * 100) / 60`, даже если скобки не изменяют результат;
- возведение в степень (Exponentiation) имеет следующий наивысший приоритет, поэтому `2**1+1` равно `3`, а не `4`, и `3*1**3` равно `3`, а не `27`;
- умножение (Multiplication) и деление (Division) имеют одинаковый приоритет, который выше приоритета сложения (Addition) и вычитания (Subtraction), которые также имеют одинаковый приоритет. Поэтому `2*3-1` равно `5`, а не `4`, и `6+4/2` равно `8`, а не `5`;
- операторы с одинаковым приоритетом вычисляются слева направо. Поэтому выражение `5-3-1` равно `1`, а не `3`, потому что `5-3` вычисляется в первую очередь, а затем `1` вычитается из `2`.

Если сомневаетесь, то всегда используйте скобки в выражениях для уверенности в том, что вычисления будут выполняться в нужном порядке.

2.8. Оператор деления по модулю

Оператор деления по модулю (или взятия остатка) (modulus operator) работает с целыми числами и возвращает остаток от деления первого операнда на второй. В языке Python оператор деления по модулю обозначается знаком процента (`%`). Синтаксис такой же, как и для других операторов:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

То есть при делении 7 на 3 получаем 2 с остатком 1.

---

[1] К сожалению, удачный русскоязычный акроним подобрать не удается, потому что Скобки и Сложение, Возведение в степень и Вычитание начинаются с одинаковых букв. Что можно запомнить по акрониму СВУДСВ? Да и сами правила приоритета вычислений достаточно просты и легко запоминаются. – *Прим. перев.*

Оператор деления по модулю неожиданно оказывается весьма полезным. Например, вы можете проверить, делится ли нацело одно число на другое: если выражение `x % y` равно нулю, то `x` делится нацело на `y`.

Также можно извлекать правую крайнюю цифру (самый младший разряд) или несколько цифр (разрядов) из любого целого числа. Например, выражение `x % 10` дает правую крайнюю цифру числа `x` (в системе счисления с основанием 10). Аналогично выражение `x % 100` дает две последние цифры этого числа.

2.9. Операции со строками

Оператор + работает со строками, но это не сложение в математическом смысле. Вместо этого он выполняет их объединение (сцепление – concatenation), т. е. соединение конца одной строки с началом другой. Например:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

Оператор * также работает со строками, умножая их содержимое на целое число, т. е. повторяя содержимое строки заданное целое число раз. Например:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

2.10. Запрос ввода от пользователя

Иногда желательно получить значение для переменной от пользователя, который вводит его с клавиатуры. Python предоставляет встроенную функцию `input`, принимающую ввод с клавиатуры[1]. При вызове этой функции программа останавливается и ждет, пока пользователь не введет что-нибудь с клавиатуры. Когда пользователь нажимает клавишу **Ввод** (Enter или Return), программа возобновляет выполнение, а функция `input` возвращает все, что ввел пользователь, в виде строки.

```
>>> inp = input()
Some silly stuff
```

[1] В версии Python 2.0 эта функция называлась `raw_input`.

```
>>> print(inp)
Some silly stuff
```

Прежде чем получить ввод от пользователя, неплохо было бы вывести приглашение, сообщающее пользователю, что именно он должен ввести. Вы можете передать строку в функцию `input`, чтобы эта строка была показана пользователю перед паузой в ожидании ввода:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

Последовательность символов `\n` в конце строки приглашения представляет условное обозначение символа перехода на новую строку (newline), т. е. специального символа, заставляющего программу «оборвать» текущую строку и перейти на новую. Поэтому ввод пользователя появляется под приглашением.

Если от пользователя ожидается ввод целого числа, то вы можете попытаться преобразовать возвращаемое значение в тип `int`, используя функцию `int()`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
#            'Какова...скорость относительно воздуха ласточки в свободном полете?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

Но если пользователь вводит что-то, отличающееся от строки цифр, то будет выведено сообщение об ошибке:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
# Какую ласточку вы имеете в виду – африканскую или европейскую?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

Позже мы увидим, как обрабатывать этот тип ошибок.

2.11. Комментарии

Когда программа увеличивается в размере и становится более сложной, ее труднее читать. Формальные языки слишком лаконичны, поэтому зачастую трудно изучать некоторый фрагмент кода и понять, что он делает и почему.

По этой причине правильной практикой является добавление примечаний в исходный код для объяснения на естественном языке, что именно делает эта программа (или фрагмент ее кода). Эти примечания называются комментариями (comments), и в Python комментарии начинаются с символа #:

```
# Вычисление процента прошедшей части текущего часа.
percentage = (minute * 100) / 60
```

В этом примере комментарий размещен на отдельной строке. Также можно размещать комментарии в конце строки кода:

```
percentage = (minute * 100) / 60              # Процент от часа.
```

Все, начиная с символа # до конца строки, игнорируется интерпретатором. В программе эта часть строки не выполняет никаких действий и не дает никакого эффекта.

Комментарии наиболее полезны, если они документируют неочевидные функциональные свойства и возможности кода. Разумно предположить, что читатель может определить, что именно делает данный код, но гораздо полезнее объяснить, почему он это делает.

Приведенный ниже комментарий просто дублирует исходный код, поэтому он избыточен и бесполезен:

```
v = 5           # Присваивание 5 переменной v.
```

Следующий комментарий содержит полезную информацию, которой нет в исходном коде:

```
v = 5           # Скорость в м/с.
```

Правильно выбранные имена переменных могут снизить потребность в комментариях, но слишком длинные имена могут сделать сложные выражения трудными для чтения, поэтому необходимо находить некоторый компромисс.

2.12. Выбор легко запоминаемых имен переменных

Если вы соблюдаете простые правила именования переменных и избегаете использования зарезервированных слов, то вам предоставлен огромный выбор для имен переменных. На начальном этапе этот выбор может оказаться непонятным и при чтении, и при написании программ. Например, следующие три программы одинаковы в плане того, что они делают, но совершенно различны, когда вы читаете и пытаетесь понять их.

```
a = 35.0
b = 12.50
c = a * b
```

```
print(c)

hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

Интерпретатор Python рассматривает эти три программы как абсолютно одинаковые, но люди видят и понимают их совершенно по-разному. Люди быстрее всего поймут предназначение второй программы, поскольку программист выбрал имена переменных, отображающих их назначение с учетом данных, хранящихся в каждой переменной.

Такие разумно выбранные имена переменных мы называем мнемоническими (легко запоминаемыми) именами переменных. Слово «мнемонический» (mnemonic)[1] означает «помогающий запоминанию». Мы выбираем мнемонические имена переменных, чтобы лучше запомнить, почему изначально были созданы именно эти переменные.

Все это выглядит великолепно, и использование мнемонических имен переменных является весьма правильной методикой, тем не менее мнемонические имена переменных могут помешать начинающему программисту развивать способность синтаксически анализировать и понимать исходный код. Причина в том, что начинающие программисты пока еще не запомнили все зарезервированные слова (их всего 33), и иногда переменные со слишком информативными именами начинают казаться им частью языка, а не правильно выбранными именами переменных.

Рассмотрим следующий небольшой фрагмент кода Python, который выполняет в цикле проход по некоторым данным. Циклы мы будем рассматривать очень скоро, но сейчас попробуйте просто поломать голову над тем, что это означает:

```
for word in words:
    print(word)
```

Что здесь происходит? Какие из элементов (for, word, in и т. д.) являются зарезервированными словами, а какие – просто именами переменных? Понимает ли Python на базовом уровне трактовку слов? Начинающие программисты испытывают сильное затруднение при отделении тех частей кода, которые обязательно должны остаться теми же, что и в примере, от частей кода, имена которых просто выбраны программистом.

Следующий код равнозначен приведенному выше:

```
for slice in pizza:
    print(slice)
```

[1] Подробное описание этого слова см. здесь: https://en.wikipedia.org/wiki/Mnemonic; https://ru.wikipedia.org/wikiМнемоника.

Для начинающего программиста в этом коде проще узнать, какие части являются зарезервированными словами, определенными языком Python, а какие – просто именами переменных, выбранными программистом. Абсолютно очевидно, что Python не понимает на базовом уровне значение слов pizza и slice и не осознает тот факт, что пицца (pizza) состоит из одного или нескольких ломтиков (slices).

Но если программа действительно считывает данные и ищет определенные слова в этих данных, то `pizza` и `slice` – это совершенно не мнемонические имена переменных. Выбор таких имен переменных отвлекает внимание от смысла программы.

По прошествии достаточно короткого интервала времени вы будете помнить наиболее часто используемые зарезервированные слова, и они сразу будут бросаться в глаза.

Части исходного кода, определенные в языке Python (for, in, print и :), выделяются цветом и начертанием шрифта, а выбранные программистом имена переменных никак не выделяются. Многим текстовым редакторам знаком синтаксис языка Python, и они подсвечивают разными цветами зарезервированные слова, чтобы вы могли отличать их от своих переменных. Через некоторое время вы начнете свободно читать код Python и быстро определять, где переменная, а где зарезервированное слово.

2.13. Отладка

На этом этапе наиболее вероятно, что вы получаете сообщения о синтаксических ошибках, сделанных из-за недопустимых имен переменных, таких как `class` и `yield`, являющихся ключевыми словами, или `odd-job` и `US$`, которые содержат неразрешенные символы.

Если в имя переменной вы вставите пробел, то Python подумает, что это два операнда без оператора между ними:

```
>>> bad name = 5
SyntaxError: invalid syntax

>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

Сообщения о синтаксических ошибках не очень-то помогают. Чаще всего выводятся сообщения `SyntaxError: invalid syntax` и `SyntaxError: invalid token` – оба не слишком информативные.

Наиболее часто совершаемой ошибкой времени выполнения является использование до определения («use before def;»), т. е. попытка использования переменной, прежде чем вы присвоили ей значение. Это может произойти, если вы неправильно ввели имя переменной:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Имена переменных чувствительны к регистру букв, поэтому `LaTeX` – это не то же самое, что `latex`.

На этом этапе наиболее вероятной причиной семантической (смысловой) ошибки является порядок выполнения операторов. Например, для вычисления значения 1/2π, возможно, покажется, что надо записать

```
>>> 1.0 / 2.0 * pi
```

Но деление выполняется первым, и вы получите значение π/2, т. е. не то, что вам нужно. Нет никакого способа дать понять Python, что именно вы имеете в виду в такой записи, и в этом случае вы не получите сообщение об ошибке, просто результат будет неверным.

2.14. Словарь терминов

- Присваивание (assignment) – инструкция, которая присваивает значение переменной.
- Конкатенация (concatenation) – соединение конца одного операнда с началом другого.
- Комментарий (comment) – информация в программе, предназначенная для других программистов (или для любого человека, читающего исходный код), которая не оказывает никакого воздействия на выполнение программы.
- Вычисление (evaluate) – упрощение выражения посредством выполнения операций, чтобы получить единственное значение.
- Выражение (expression) – совокупность переменных, операторов и значений, которая представляет одно итоговое значение – результат выражения.
- Число с плавающей точкой (floating point) – тип данных, представляющий числа с дробными частями.
- Целое число (integer) – тип данных, представляющий целые числа.
- Ключевое слово (keyword) – зарезервированное слово, используемое компилятором (и транслятором) для синтаксического анализа программ. В качестве имен переменных нельзя использовать ключевые слова, такие как `if`, `def`, `while` и т. п.
- Мнемоника (mnemonic) – помощь в запоминании. Мы часто даем переменным мнемонические имена, чтобы лучше запомнить, что хранится в конкретной переменной.
- Оператор деления по модулю (modulus operator) – оператор, обозначенный символом процента (`%`), который работает с целыми числами и выдает остаток от деления одного целого числа на другое.
- Операнд (operand) – одно из значений, с которыми работает оператор.

- Оператор (operator) – специальный символ, представляющий простое вычисление, например сложение, умножение или конкатенацию (соединение) строк.
- Правила приоритета (rules of precedence) – набор правил, управляющих порядком вычислений в выражениях, содержащих несколько операторов и операндов.
- Инструкция (statement) – часть исходного кода, которая представляет команду или действие. До сих пор мы видели только инструкции присваивания и вывода выражений.
- Строка (string) – тип данных, представляющий последовательность символов.
- Тип (данных) (type) – категория значений. До сих пор мы встречались с типами данных: целые числа (`int`), числа с плавающей точкой (`float`) и строки (`str`).
- Значение (value) – один из основополагающих элементов данных, например число или строка, который обрабатывает программа.
- Переменная (variable) – имя, которое указывает (ссылается) на некоторое значение.

2.15. Упражнения

УПРАЖНЕНИЕ 2 Написать программу, которая использует `input` для вывода приглашения (промпта), предлагающего пользователю ввести свое имя, а затем приветствует его по имени.

```
Enter your name: Chuck
Hello Chuck
```

УПРАЖНЕНИЕ 3 Написать программу, предлагающую пользователю ввести количество часов и оплату в час для вычисления общей суммы выплаты.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

Здесь мы не обращаем внимания на обеспечение точности двух разрядов после десятичной точки. Если хотите, можете поэкспериментировать со встроенной в Python функцией round для правильного округления итоговой выплаты до двух цифр после десятичной точки.

УПРАЖНЕНИЕ 4 Предположим, что мы выполняем следующие инструкции присваивания:

```
width = 17
height = 12.0
```

Для каждого из приведенных ниже выражений напишите значение, полученное при его вычислении, и тип (значения выражения).

1. `width//2`.
2. `width/2.0`.
3. `height/3`.
4. `1 + 2 * 5`.

Используйте интерпретатор Python, чтобы проверить правильность своих ответов.

УПРАЖНЕНИЕ 5 Написать программу, которая предлагает пользователю ввести температуру по шкале Цельсия, выполняет преобразование в температуру по шкале Фаренгейта и выводит преобразованное значение температуры.

Глава 3

Условное выполнение

3.1. Логические выражения

Логическое выражение (boolean expression) – это выражение, которое является либо истинным, либо ложным. В приведенных ниже примерах используется оператор ==, который сравнивает два операнда и дает результат `True` (истина), если они равны, иначе результатом является `False` (ложь).

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` и `False` – это специальные значения, принадлежащие классу `bool`, которые не являются строками:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Оператор == является одним из операторов сравнения. Ниже приведены примеры использования других операторов сравнения:

```
x != y               # Значение x не равно значению y.
x > y                # Значение x больше значения y.
x < y                # Значение x меньше значения y.
x >= y               # Значение x больше или равно y.
x <= y               # Значение x меньше или равно y.
x is y               # Значение x то же самое, что и значение y.
x is not y           # Значение x не то же самое, что и значение y.
```

Хотя эти операторы, вероятно, уже знакомы вам, символы языка Python отличаются от математических символов, используемых для тех же операций. Самой распространенной ошибкой является использование одного знака равенства (=) вместо удвоенного (==). Необходимо запомнить, что = – это оператор присваивания, а == – оператор сравнения. Это не то же самое, что =< или =>.

3.2. Логические операторы

В языке Python существуют три логических оператора: `and`, `or` и `not`. Семантика (смысл) этих операторов аналогична их смыслу в английском языке. Например:

```
x > 0 and x < 10
```

истинно, только если `x` больше нуля и меньше 10.

```
n%2 == 0 or n%3 == 0
```

истинно, если любое из этих условий истинно, т. е. если число n делится нацело на 2 или на 3.

Наконец, оператор `not` отрицает значение логического выражения, поэтому `not (x > y)` истинно, если выражение `x > y` ложно, т. е. если значение `x` меньше или равно `y`.

Строго говоря, операндами логических операторов должны быть логические выражения, но Python не очень строг. Любое ненулевое выражение интерпретируется как «истина».

```
>>> 17 and True
True
```

Такая гибкость может оказаться полезной, но существуют некоторые тонкие моменты, которые могут привести к путанице. Возможно, вам придется избегать их до тех пор, пока вы не будете абсолютно уверены в том, что делаете.

3.3. Условное выполнение

Для того чтобы писать полезные программы, нам почти всегда необходима возможность проверять условия и соответствующим образом изменять поведение программы. Условные инструкции (conditional statements) предоставляют нам эту возможность. Ниже приведена простейшая форма инструкции `if` (если):

```
if x > 0 :
    print('x is positive')
```

Логическое выражение после оператора `if` называется условием (condition). Инструкция `if` завершается символом двоеточия (:), а строка (или несколько строк) после инструкции проверки условия сдвигается вправо.

Если логическое условие истинно, то выполняется сдвинутая вправо инструкция. Если логическое условие ложно, то сдвинутая вправо инструкция пропускается.

Инструкции проверки условия `if` имеют такую же структуру, как и определения функций или циклы `for`[1]. Инструкция состоит из строки заголовка, за-

[1] Функции мы будем рассматривать в главе 4, а циклы – в главе 5.

вершающейся символом двоеточия (:), а за этой строкой следует сдвинутый вправо блок. Подобные инструкции называются составными инструкциями, потому что они занимают более одной строки.

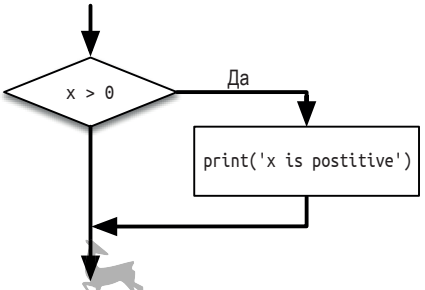


Рис. 3.1 ❖ Логическая схема проверки условия (if)

Количество инструкций, которые могут находиться в теле, не ограничено, но обязательно должна существовать хотя бы одна инструкция. Иногда полезно иметь тело без инструкций (обычно как зарезервированное место для кода, который пока еще не написан). В этом случае можно воспользоваться инструкцией pass, которая ничего не делает.

```
if x < 0 :
    pass          # Здесь необходимо обрабатывать отрицательные значения.
```

Если вы вводите инструкцию if в интерпретаторе Python, то промпт изменит вид с >>> на три точки (...), подсказывая, что вы находитесь внутри блока инструкций, как показано ниже:

```
>>> x = 3
>>> if x < 10:
...     print('Small')
...
Small
>>>
```

При использовании интерпретатора Python необходимо обязательно оставить пустую строку в конце блока инструкций, иначе Python выведет сообщение об ошибке:

```
>>> x = 3
>>> if x < 10:
...     print('Small')
...     print('Done')
  File "<stdin>", line 3
    print('Done')
        ^
SyntaxError: invalid syntax
```

Пустая строка в конце блока инструкций не является необходимой при написании и выполнении скрипта, но может сделать исходный код более удобным для чтения.

3.4. Альтернативная последовательность выполнения

Второй формой инструкции `if` является альтернативная последовательность выполнения (alternative execution), в которой имеются две возможности и условие, определяющее выбор одной из них для выполнения. Синтаксис показан ниже:

```
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

Если остаток при делении x на 2 равен 0, то мы знаем, что x – четное число, и программа выводит соответствующее сообщение об этом. Если условие ложно, то выполняется второй блок инструкций.

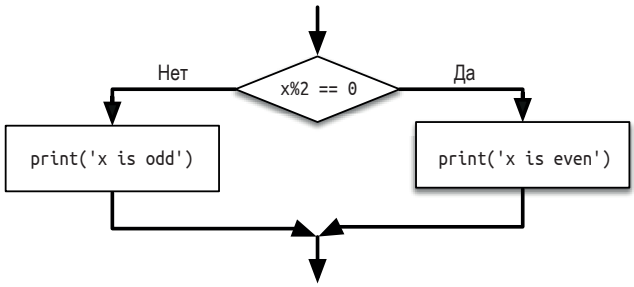


Рис. 3.2 ❖ Логическая схема конструкции `if-then-else`

Поскольку условие непременно должно быть либо истинным, либо ложным, будет выполняться один и только один из альтернативных вариантов. Эти альтернативные варианты называются ветвями (branches), потому что они разветвляют поток выполнения программы.

3.5. Цепочечные условные инструкции

Иногда существует более двух возможностей, и нам требуется более двух ветвей. Одним из способов записи подобного вычисления является цепочечная условная инструкция (chained conditional):

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

Инструкция elif является аббревиатурой словосочетания else if (иначе если). И в этом случае будет выполнена только одна ветвь.

Количество инструкций elif не ограничено. Если имеется вариант else, то он непременно должен быть самым последним, но его присутствие не обязательно.

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

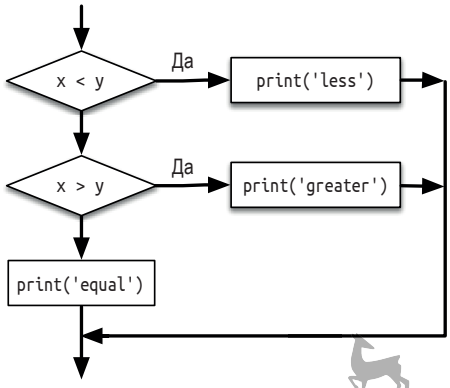


Рис. 3.3 ❖ Логическая схема if-then-elseif

Условия проверяются в том порядке, в котором они записаны. Если первое условие ложно, то проверяется следующее и т. д. Если одно из условий истинно, то выполняется соответствующая ветвь, и инструкция завершается. Даже если несколько условий являются истинными, выполняется только ветвь, соответствующая первому истинному условию.

3.6. Вложенные условные инструкции

Одна условная инструкция также может быть вложенной в другую. Пример с тремя ветвями условного выполнения можно было бы записать следующим образом:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

Внешняя условная инструкция содержит две ветви. Первая ветвь содержит простую инструкцию. Вторая – другую инструкцию `if`, которая, в свою очередь, содержит две ветви. Обе эти ветви являются простыми инструкциями, хотя они могли бы быть и условными инструкциями.

Несмотря на то что сдвиг вправо инструкций в теле формирует явно видимую структуру, вложенные условные инструкции очень быстро становятся трудными для чтения. В общем случае лучше по возможности избегать подобных конструкций.

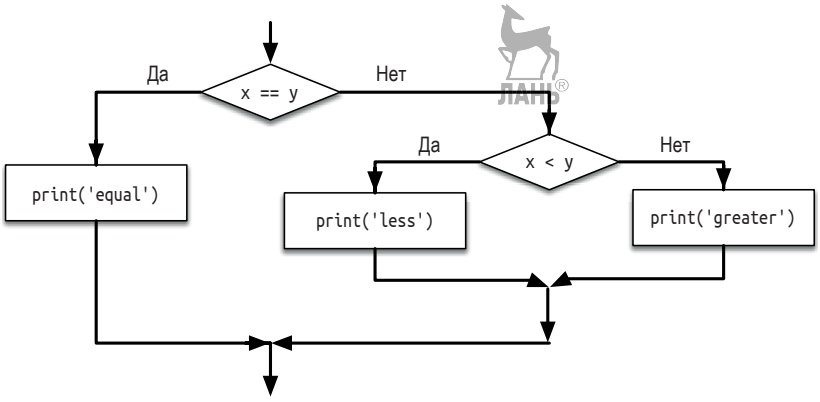


Рис. 3.4 ❖ Вложенные условные инструкции `if`

Логические операторы часто предоставляют возможность упрощения вложенных условных инструкций. Например, приведенный ниже код можно переписать, используя одну условную инструкцию:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

Инструкция `print` выполняется, только если успешно выполняются оба условия, поэтому можно получить тот же результат с помощью оператора `and`:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

3.7. Перехват исключений с использованием ключевых слов `try` и `except`

Ранее мы видели фрагмент кода, в котором использовались функции `input` и `int` для чтения и синтаксического анализа (парсинга) целого числа, введенного пользователем. Но мы также видели, каким коварным может оказаться это действие:

```
>>> prompt = "What is the air velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What is the air velocity of an unladen swallow?
# Какова скорость свободного полета ласточки относительно воздушной среды?
What do you mean, an African or a European swallow?
# Вы имеете в виду африканскую или европейскую ласточку?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

При выполнении этих инструкций в интерпретаторе Python мы получаем новый промпт от интерпретатора, мысленно говорим «ой» и переходим к следующей инструкции.

Но если записать этот код в скрипт Python, то при этой ошибке выполнение скрипта немедленно останавливается для обратной трассировки. Следующая инструкция не выполняется.

Ниже приведен пример программы для преобразования температуры по шкале Фаренгейта в температуру по шкале Цельсия:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Исходный код: http://www.py4e.com/code3/fahren.py
```

Если выполнить этот код и задать недопустимое входное значение, то программа просто завершится аварийно и выдаст недружественное сообщение об ошибке:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

Существует встроенная в Python структура условного выполнения для обработки подобных типов ожидаемых и неожидаемых ошибок, которая называется try/except. Основной принцип работы `try` и `except` состоит в том, что вы знаете, что в некоторой последовательности инструкций может возникнуть проблема, поэтому необходимо добавить некоторые инструкции, которые должны выполняться, если возникает ошибка. Эти дополнительные инструкции (в блоке исключения except) игнорируются, если ошибка не возникает.

Можно считать функциональные возможности `try` и `except` в Python «страховым полисом» для последовательности инструкций.

Программу преобразования температур можно переписать следующим образом:

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Исходный код: http://www.py4e.com/code3/fahren2.py
```

Python начинает с выполнения последовательности инструкций в блоке `try`. Если все идет хорошо, то блок `except` пропускается, и выполнение продолжается. Если в блоке `try` возникает исключение (ошибка), то Python немедленно покидает блок `try` и выполняет последовательность инструкций в блоке `except`.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Обработка исключения (ошибки) с помощью инструкции `try` называется перехватом исключения. В этом примере в части `except` выводится сообщение об ошибке и подсказка о правильном вводе. В общем случае перехват исключения дает вам шанс устранить проблему, или повторить попытку, или, по крайней мере, завершить программу аккуратно.

3.8. Вычисление логических выражений по сокращенной схеме

Когда Python обрабатывает логическое выражение, такое как `x >= 2 and (x/y) > 2`, то вычисляет такое выражение слева направо. По определению оператора `and`, если `x` меньше 2, то выражение `x >= 2` имеет значение `False`, следовательно, все выражение в целом равно `False` вне зависимости от того, вычисляется ли для `(x/y) > 2` значение `True` или `False`.

Когда Python обнаруживает, что дальнейшее вычисление оставшейся части логического выражения ничего не дает для конечного результата, то он останавливает обработку и не выполняет никаких вычислений в оставшейся части логического выражения. Если вычисление логического выражения останавливается, потому что его общее значение уже известно, это называется сокращенной (или укороченной) схемой вычисления (short-circuiting evaluation).

Это может выглядеть хорошим свойством, и сокращенная схема вычислений приводит к хитроумной методике, называемой шаблон-защитник (guardian pattern). Рассмотрим следующий фрагмент кода в интерпретаторе Python:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Третье вычисление завершилось с критической ошибкой, потому что Python вычислял `(x/y)`, а `y` равен нулю, что привело к ошибке времени выполнения. Но первое и второе вычисления не привели к критической ошибке, потому что в первом вычислении `y` не равен нулю, а во втором первая часть логических выражений `x >= 2` дает результат `False`, поэтому выражение `(x/y)` даже не вычислялось по правилу сокращенной схемы, и ошибка не возникла.

Мы можем сформировать логическое выражение для стратегически обоснованного размещения защитного вычисления прямо перед вычислением, которое может привести к критической ошибке, как показано ниже:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

В первом логическом выражении `x >= 2` равно `False`, поэтому вычисление останавливается на операторе `and`. Во втором логическом выражении условие `x >= 2` равно `True`, но условие `y != 0` равно `False`, поэтому мы никогда не переходим к вычислению `(x/y)`.

В третьем логическом выражении условие `y != 0` находится после вычисления `(x/y)`, поэтому выражение приводит к критической ошибке.

Во втором выражении мы говорим, что условие `y != 0` действует как защитник, обеспечивающий возможность вычисления `(x/y)`, только если `y` не равно нулю.

3.9. Отладка

Обратная трассировка Python выводится при возникновении ошибки и содержит весьма большой объем информации, который может оказаться ошеломляющим. Обычно наиболее полезны следующие части вывода:

- какого типа ошибка возникла;
- где она возникла.

Синтаксические ошибки обычно легко найти, но существует несколько хитростей. Ошибки, связанные с пробельными символами, могут оказаться трудно выявляемыми, потому что пробелы и табуляции невидимы, и мы привыкли не обращать на них внимания.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

В этом примере проблема заключается в том, что вторая строка сдвинута вправо на один пробел. Но сообщение об ошибке указывает на `y`, что неверно и сбивает с толку. В общем случае сообщения об ошибках сообщают, где проблема была обнаружена, но настоящая ошибка может находиться раньше в коде, иногда в предыдущей строке.

Итак, в общем случае сообщения об ошибках говорят, где была обнаружена проблема, но зачастую не то место, где она возникла.

3.10. Словарь терминов

- Тело (body) – последовательность инструкций в составной инструкции.
- Логическое выражение (boolean expression) – выражение, значение которого равно либо `True` (истина), либо `False` (ложь).
- Ветвь (branch) – одна из альтернативных последовательностей инструкций в условной инструкции.
- Цепочечная условная инструкция (chained conditional) – условная инструкция с последовательностью альтернативных ветвей.
- Оператор сравнения (comparison operator) – один из операторов, которые сравнивают свои операнды: `==`, `!=`, `>`, `<`, `>=` и `<=`.
- Условная инструкция (conditional statement) – инструкция, управляющая потоком выполнения в зависимости от некоторого условия.

- Условие (condition) – логическое выражение в условной инструкции, определяющее, какая ветвь будет выполняться.
- Составная инструкция (compound statement) – инструкция, которая состоит из заголовка и тела. Строка заголовка завершается символом двоеточия (:). Строки тела сдвигаются вправо относительно строки заголовка.
- Шаблон-защитник (guardian pattern) – создается при формулировании логического выражения и использует дополнительные операции сравнения, чтобы получить преимущества от применения сокращенной схемы вычислений.
- Логический оператор (logical operator) – один из операторов, объединяющих логические выражения: `and`, `or` и `not`.
- Вложенная условная инструкция (nested conditional) – условная инструкция, находящаяся в одной из ветвей другой условной инструкции.
- Обратная трассировка (traceback) – список выполняемых (в данный момент) функций, который выводится при возникновении исключения (ошибки).
- Сокращенная (или укороченная) схема вычислений (short circuit) – поведение Python в процессе вычисления логического выражения с остановкой этого вычисления, потому что Python уже знает его конечное значение без необходимости вычисления оставшейся части выражения.

3.11. Упражнения

УПРАЖНЕНИЕ 1 Переписать программу вычисления выплаты, чтобы повысить почасовую оплату в 1.5 раза для тех, кто работал более 40 часов.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

УПРАЖНЕНИЕ 2 Переписать программу вычисления выплаты, используя `try` и `except`, чтобы программа аккуратно обрабатывала нечисловые входные данные, выводя сообщение и завершая свою работу. Ниже показаны два примера обработки исключений в этой программе:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input
# Ошибка, введите числовое значение

Enter Hours: forty
Error, please enter numeric input
```

УПРАЖНЕНИЕ 3 Написать программу, запрашивающую ввод оценки от 0.0 до 1.0. Если введена оценка вне этого диапазона, то вывести сообщение об ошибке. Если введена оценка между 0.0 и 1.0, то вывести соответствующий символьный балл, используя следующую таблицу:

```
 Score   Grade
#Оценка  Балл
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
 < 0.6     F

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Выполнить программу несколько раз, как показано выше, чтобы протестировать различные отличающиеся друг от друга значения для ввода.

Глава 4

Функции

4.1. Вызовы функций

С точки зрения программирования функция (function) – это именованная последовательность инструкций, выполняющих некоторое вычисление. Когда вы определяете функцию, то задаете ее имя и последовательность инструкций. В дальнейшем вы можете «вызвать» функцию по имени. Ранее мы уже видели пример вызова функции (function call):

```
>>> type(32)
<class 'int'>
```

Здесь имя функции `type`. Выражение в скобках называется аргументом (argument) функции. Аргумент – это значение или переменная, которая передается как входные данные для функции. Результат выполнения функции `type` – тип аргумента.

Обычно говорят, что функция «принимает» аргумент и «возвращает» результат. Результат называется возвращаемым значением (return value).

4.2. Встроенные функции

Python предоставляет ряд важных встроенных функций, которые можно использовать без их предварительного определения. Создатели языка Python написали набор функций для решения наиболее часто встречающихся задач и включили этот набор в Python, чтобы мы могли ими воспользоваться.

Функции max и min вычисляют соответственно наибольшее и наименьшее значения в любом списке:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

Функция `max` сообщает нам «наибольший символ» в строке (которым оказывается буква «w»), а функция `min` показывает наименьший символ (это пробел).

Другая часто используемая встроенная функция `len` возвращает количество элементов в переданном ей аргументе. Если аргументом `len` является строка, то она возвращает число символов в этой строке.

```
>>> len('Hello world')
11
>>>
```

Действие этих функций не ограничивается строками. Они могут работать с любым множеством значений, как мы увидим в следующих главах.

Имена встроенных функций следует воспринимать как зарезервированные слова (т. е. избегать использования слов max, min, len и т. п. в качестве имен переменных).

4.3. Функции преобразования типов

Python также предоставляет встроенные функции, которые выполняют преобразование значения из одного типа в другой. Функция `int` принимает любое значение и преобразовывает его в целое число, если это возможно, а в противном случае выдает сообщение об ошибке:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

Функция `int` может преобразовывать значения с плавающей точкой в целые числа, но не округляет их корректно, а просто отбрасывает дробную часть:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Функция `float` выполняет преобразование целых чисел и строк в числа с плавающей точкой:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Функция `str` преобразовывает свой аргумент в строку:

```
>>> str(32)
```

```
'32'
>>> str(3.14159)
'3.14159'
```

4.4. Математические функции

В Python имеется модуль `math`, который предоставляет широко известные математические функции. Чтобы получить возможность воспользоваться этим модулем, нужно сначала импортировать его:

```
>>> import math
```

Эта инструкция создает объект модуля (module object) с именем `math`. Если передать в функцию вывода `print` объект модуля, то получим некоторую информацию о нем:

```
>>> print(math)
<module 'math' (built-in)>
```

Этот объект модуля содержит функции и переменные, определенные в модуле `math`. Для доступа к одной из таких функций необходимо указать имя модуля и имя функции, разделенные точкой. Такой формат называется точечной записью (записью через точку) (dot notation).

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

В первом примере вычисляется логарифм по основанию 10 отношения сигнал/шум. В модуле `math` также представлена функция `log`, вычисляющая натуральный логарифм по основанию e.

Второй пример находит значение синуса угла, заданного в радианах. Имя переменной `radians` является напоминанием того, что `sin` и другие тригонометрические функции (`cos`, `tan` и т. д.) принимают аргументы в радианах. Чтобы преобразовать градусы в радианы, необходимо значение разделить на 360 и умножить на 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

Выражение `math.pi` извлекает переменную `pi` из модуля `math`. Значением этой переменной является приближение числа π с точностью до 15 знаков.

Если вы хорошо знаете тригонометрию, то можете проверить предыдущий результат, сравнив его с квадратным корнем из двух, разделенным на два:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.5. Случайные числа

Если заданы одни и те же входные данные, то большинство компьютерных программ при каждом выполнении генерируют одинаковые выходные данные, поэтому такие программы называются детерминированными (deterministic). Детерминизм обычно является положительным свойством, поскольку мы ожидаем, что одинаковые вычисления дают одинаковые результаты. Но для некоторых приложений необходимо, чтобы компьютер стал непредсказуемым. Очевидный пример – игры, но существуют и другие примеры.

Заставить программу стать действительно недетерминированной – не такое простое дело, но существуют методы, позволяющие сделать так, чтобы программа, по крайней мере, выглядела недетерминированной. Одним из таких методов является использование алгоритмов, генерирующих псевдослучайные числа (pseudorandom numbers). Псевдослучайные числа не являются действительно случайными, потому что генерируются с помощью детерминированного вычисления, но если рассматривать эти числа непредвзято, то их практически невозможно отличить от случайных.

Модуль `random` предоставляет функции, генерирующие псевдослучайные числа (далее я буду называть эти числа просто «случайными»).

Функция `random` возвращает случайное число с плавающей точкой в диапазоне от 0.0 до 1.0 (включая 0.0, но не 1.0). При каждом вызове `random` вы получаете следующее число в длинной последовательности. Чтобы увидеть пример такой последовательности, выполните следующий цикл:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Эта программа генерирует приведенный ниже список из 10 случайных чисел от 0.0 до 1.0, не включая последнее число.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

УПРАЖНЕНИЕ 1 Написать программу в своей системе и посмотреть, какие числа вы получили. Выполнить программу еще раз и посмотреть, какие числа вы получили.

Функция `random` – всего лишь одна из многих функций, работающих со случайными числами. Функция `randint` принимает параметры `low` и `high` и возвращает случайное целое число между `low` и `high` (включая оба значения).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Для выбора элемента из последовательности случайным образом можно воспользоваться функцией `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Кроме того, модуль `random` предоставляет функции для генерации случайных значений из непрерывных распределений, включая гауссово, экспоненциальное, гамма и др.

4.6. Добавление новых функций

До сих пор мы использовали только функции, которые входят в состав языка Python, но существует еще и возможность добавления новых функций. Определение функции (function definition) задает имя новой функции и последовательность инструкций, которые выполняются при вызове этой функции. После определения функции мы можем многократно использовать ее в программе.

Пример определения функции:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

Здесь `def` – ключевое слово, означающее, что это определение функции. Имя этой функции `print_lyrics`. Правила для имен функций точно такие же, как и для имен переменных: буквы, цифры и некоторые знаки препинания допускаются, но первым символом не может быть цифра. Запрещено использовать ключевые слова как имена функций, и вы должны избегать присваивания одинакового имени переменной и функции.

Пустые круглые скобки после имени функции означают, что эта функция не принимает никакие аргументы. В дальнейшем мы будем создавать функции, принимающие аргументы как входные данные.

Первая строка определения функции называется заголовком (header), остальная часть называется телом (body) функции. Строка заголовка обязательно должна завершаться символом двоеточия, а строки тела непременно должны быть сдвинуты вправо. По соглашению строки тела всегда сдвигаются вправо при помощи четырех пробелов. Тело может содержать любое число инструкций.

Если вы вводите определение функции в интерактивном режиме, то интерпретатор вместо обычного промпта выводит многоточие (...), чтобы подсказать вам, что определение не завершено:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

Для завершения определения функции необходимо ввести пустую строку (в скрипте это не обязательно).

Определение функции создает переменную с тем же именем.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

Значением `print_lyrics` является объект функции (function object), который имеет тип `function`.

Синтаксис вызова новой функции точно такой же, как и для встроенных функций:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

После определения функции ее можно использовать внутри другой функции. Например, для повторения приведенного выше припева можно написать функцию с именем `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

Затем вызвать функцию `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Но на самом деле так песни не пишут и не поют.

4.7. Определение и использование

Если собрать вместе все фрагменты исходного кода из предыдущего раздела, то вся программа в целом выглядит следующим образом:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Исходный код: http://www.py4e.com/code3/lyrics.py
```


Эта программа содержит два определения функций: `print_lyrics` и `repeat_lyrics`. Определения функций выполняются так же, как и другие инструкции, но результатом является создание объектов функций. Инструкции внутри функции не начинают выполняться до тех пор, пока функция не будет вызвана, и определение функции не генерирует никакого вывода.

Как можно предположить, функцию необходимо создать до того, как вы сможете выполнить ее. Другими словами, определение функции должно быть выполнено до ее первого вызова.

УПРАЖНЕНИЕ 2 Переместить самую последнюю строку приведенной выше программы в начало исходного кода, чтобы вызов функции выполнялся перед определениями. Запустите программу и посмотрите, какое сообщение об ошибке будет выведено.

УПРАЖНЕНИЕ 3 Переместить вызов функции обратно в конец исходного кода, потом переместить определение функции print_lyrics после определения функции repeat_lyrics. Что происходит при запуске этого варианта программы?

4.8. Поток выполнения

Чтобы обеспечить определение функции до ее первого использования, необходимо знать порядок выполнения инструкций, который называется потоком выполнения (flow of execution).

Выполнение всегда начинается с первой инструкции программы. Инструкции выполняются поочередно по одной сверху вниз.

Определения функций не изменяют поток выполнения программы, но следует запомнить, что инструкции внутри функции не выполняются до тех пор, пока функция не будет вызвана.

Вызов функции можно считать изменением маршрута в потоке выполнения. Вместо перехода к следующей инструкции поток перемещается в тело

функции, выполняет в нем все инструкции, а затем снова возвращается в то место, которое он покинул.

Это выглядит достаточно просто, пока вы не вспомните, что одна функция может вызывать другую. Находясь в середине одной функции, программа может потребовать выполнение инструкций в другой функции. Но при выполнении этой новой функции может возникнуть необходимость выполнения еще одной функции.

К счастью, Python ведет себя правильно, постоянно отслеживая, где он находится, поэтому при каждом завершении функции программа возвращается в то место, которое она покинула, в функции, вызвавшей только что завершившуюся функцию. Когда достигается конец программы, она завершается.

Какова же мораль этой унылой истории? При чтении программы не обязательно всегда читать ее строго сверху вниз от начала до конца. Иногда полезнее следовать потоку выполнения.

4.9. Параметры и аргументы

Для некоторых из встроенных функций, которые мы видели ранее, требуются аргументы. Например, при вызове `math.sin` в качестве аргумента передается число. Некоторые функции принимают более одного аргумента: `math.pow` принимает два – основание и степень.

Внутри функции аргументы присваиваются переменным, которые называются параметрами (parameters). Ниже приведен пример функции, определенной пользователем, которая принимает аргумент:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Эта функция присваивает переданный ей аргумент параметру с именем `bruce`. При вызове функции она выводит значение этого параметра (чем бы оно ни было) дважды.

Функция работает с любым значением, которое она может вывести.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

Правила сочетания и комбинирования, применяющиеся к встроенным функциям, также применимы и к функциям, определяемым пользователем, поэтому мы можем использовать любой тип выражения как аргумент для `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Аргумент вычисляется до вызова функции, поэтому в приведенных примерах выражения `'Spam '*4` и `math.cos(math.pi)` вычисляются только один раз.

В качестве аргумента также можно использовать переменную:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Имя переменной, передаваемой как аргумент (`michael`), никак не воздействует на имя параметра (`bruce`). Не важно, как это значение было названо в вызывающей функции, потому что в теле `print_twice` мы все значения называем `bruce`.

4.10. Продуктивные и пустые функции

Некоторые из используемых нами функций, например математические, дают результаты. За неимением лучшего имени я называю их продуктивными функциями (fruitful functions). Другие функции, например `print_twice`, выполняют некоторое действие, но не возвращают значение. Они называются пустыми функциями (void functions).

При вызове продуктивной функции вы почти всегда намерены что-либо сделать с ее результатом, например можно присвоить его переменной или использовать как часть выражения:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

При вызове функции в интерактивном режиме Python выводит ее результат:

```
>>> math.sqrt(5)
2.23606797749979
```

Но в скрипте, если вы вызываете продуктивную функцию и не сохраняете ее результат в переменной, возвращаемое значение рассеивается как туман и исчезает бесследно.

```
math.sqrt(5)
```

Скрипт вычисляет квадратный корень из пяти, но поскольку результат не сохраняется в переменной и не выводится, это не очень полезное действие.

Пустые функции могут выводить что-либо на экран или осуществлять некоторое другое воздействие (побочный эффект), но они не имеют возвращаемого значения. Если вы попытаетесь присвоить переменной результат пустой функции, то получите специальное значение None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

Значение None – это не то же самое, что строка «None». Это специальное значение, которое имеет собственный особый тип:

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

Для возврата результата из своих функций мы используем инструкцию return. Например, можно было бы создать весьма простую функцию addtwo, которая выполняет сложение двух чисел и возвращает результат.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

# Исходный код: http://www.py4e.com/code3/addtwo.py
```

При выполнении этого скрипта инструкция print выведет число 8, потому что функция addtwo вызвана с аргументами 3 и 5. Внутри функции параметрам a и b присваиваются значения 3 и 5 соответственно. Функция вычисляет сумму этих двух чисел и помещает ее в локальную для этой функции переменную added. Затем используется инструкция return для передачи вычисленного значения обратно в вызывающий код как результат выполнения функции. Результат присваивается переменной x и выводится на экран.

4.11. Зачем нужны функции

Возможно, вам не совсем понятно, почему стоит беспокоиться о разделении программы на функции. Ниже перечислено несколько причин:

- создание новой функции дает возможность именовать группу инструкций, а это делает программу более простой и удобной для чтения, понимания и отладки;
- функции могут уменьшить размер программы, устраняя повторяющиеся фрагменты кода. В дальнейшем, когда вы будете вносить изменения, потребуется сделать это только в одном месте;

- разделение длинной программы на функции позволяет отлаживать ее части по отдельности, а затем собирать все части в единое работающее целое;
- правильно спроектированные функции зачастую весьма полезны для многих программ. После того как вы напишете и отладите функцию, в дальнейшем сможете многократно ее использовать.

В остальной части книги мы часто будем использовать определение функции для описания некоторой концепции. Один из навыков создания и использования функций состоит в том, чтобы функция надлежащим образом определяла основную идею, например «поиск наименьшего значения в списке». В дальнейшем будет продемонстрирован код, который ищет наименьшее значение в списке, и мы представим его как функцию с именем `min`, принимающую список значений как аргумент, возвращающую наименьшее значение в этом списке.

4.12. Отладка

Если вы используете текстовый редактор для написания скриптов, то, возможно, встречаетесь с проблемами, возникающими из-за пробелов и табуляций. Наилучший способ избежать подобных проблем – использовать только пробелы (никаких табуляций). Большинство текстовых редакторов, знакомых с Python, делают это по умолчанию, но некоторые не обладают таким свойством.

Табуляции и пробелы обычно невидимы, поэтому их трудно отлаживать, так что найдите текстовый редактор, который автоматически управляет сдвигом строк исходного кода.

Кроме того, не забывайте сохранять свою программу в файле перед ее запуском. Некоторые среды разработки делают это автоматически, но не все. В этом случае программа, на которую вы смотрите в текстовом редакторе, – это не та же программа, которую вы запускаете.

Отладка может занять очень много времени, если вы продолжаете запускать одну и ту же неправильную программу снова и снова.

Необходимо убедиться, что код, который вы видите в редакторе, – это тот же код, который вы выполняете. Если вы не уверены в этом, то поместите что-нибудь вроде `print("hello")` в самое начало программы и запустите ее еще раз. Если не увидите вывод слова `hello`, значит, вы запускаете не ту версию программы.

4.13. Словарь терминов

- Алгоритм (algorithm) – общий процесс решения некоторой категории задач.

- Аргумент (argument) – значение, передаваемое в функцию при ее вызове. Это значение присваивается соответствующему параметру в функции.
- Возвращаемое значение (return value) – результат выполнения функции. Если вызов функции используется как выражение, то возвращаемое значение является значением этого выражения.
- Вызов функции (function call) – инструкция, которая выполняет функцию. Вызов состоит из имени функции, за которым следует список аргументов в скобках.
- Детерминированный (deterministic) – свойство программы выполнять одно и то же действие при каждом ее выполнении с одинаковыми входными данными.
- Заголовок (header) – первая строка определения функции.
- Инструкция импорта (import instruction) – инструкция, которая считывает файл модуля и создает объект модуля.
- Комбинирование (composition) – использование выражения как части более крупного выражения или инструкции как части более крупной инструкции.
- Объект модуля (module object) – значение, созданное инструкцией импорта `import`, которое предоставляет доступ к данным и коду, определенным в модуле.
- Объект функции (function object) – значение, созданное определением функции. Имя функции – это переменная, которая указывает (ссылается) на объект функции.
- Определение функции (function definition) – инструкция, создающая новую функцию, определяющая ее имя, параметры и выполняемые ею инструкции.
- Параметр (parameter) – имя, используемое внутри функции для указания (ссылки) на значение, переданное как аргумент.
- Поток выполнения (flow of execution) – порядок, в котором выполняются инструкции во время работы программы.
- Продуктивная функция (fruitful function) – функция, которая возвращает значение.
- Псевдослучайные числа (pseudorandom) – последовательность чисел, которые выглядят как случайные, но генерируются детерминированной программой.
- Пустая функция (void function) – функция, которая не возвращает значение.
- Тело (body) – последовательность инструкций внутри определения функции.
- Точечная запись (dot notation) – синтаксис вызова функции, находящейся в другом модуле, – записывается имя модуля, затем точка и имя функции.
- Функция (function) – именованная последовательность инструкций, выполняющих некоторую полезную операцию. Функции могут иметь или не иметь аргументы и возвращать или не возвращать результат.

4.14. Упражнения

УПРАЖНЕНИЕ 4 Для чего нужно ключевое слово `def` в языке Python?

a) Это сленговое выражение, означающее «следующий код действительно крутой».
b) Оно обозначает начало функции.
c) Оно означает, что следующий за ним выровненный вправо блок кода должен быть сохранен для использования в дальнейшем.
d) Оба варианта b) и c) правильны.
e) Ни один из приведенных выше вариантов не является правильным.

УПРАЖНЕНИЕ 5 Что выводит приведенная ниже программа на языке Python?


```
def fred():
    print("Zap")

def jane():
    print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane.
b) Zap ABC Zap.
c) ABC Zap jane.
d) ABC Zap ABC.
e) Zap Zap Zap.

УПРАЖНЕНИЕ 6 Перепишите программу вычисления почасовой оплаты с дополнительной выплатой за сверхурочную работу и создайте функцию `computepay`, которая принимает два параметра (`hours` и `rate`).


```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

УПРАЖНЕНИЕ 7 Перепишите программу из предыдущей главы, используя функцию `computegrade`, которая принимает оценку как параметр и возвращает символьный балл как строку.

```
  Score   Grade
# Оценка  Балл
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
 < 0.6     F

Enter score: 0.95
```

```
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Выполните эту программу несколько раз, чтобы протестировать ее с различными значениями входных данных.

Глава 5

Итерации

5.1. Обновление переменных

Общеизвестным шаблоном для операций присваивания является инструкция присваивания, которая обновляет переменную, при этом новое значение этой переменной зависит от старого.

```
x = x + 1
```

Это означает: «Взять текущее значение x, прибавить 1, затем обновить переменную x, присвоив ей полученное новое значение».

Если вы попытаетесь обновить несуществующую переменную, то получите сообщение об ошибке, потому что Python вычисляет правую часть выражения перед присваиванием значения переменной x:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Прежде чем вы сможете обновить переменную, вы должны инициализировать ее, обычно с помощью простого присваивания:

```
>>> x = 0
>>> x = x + 1
```

Обновление переменной посредством прибавления 1 называется инкрементом (increment), а посредством вычитания 1 – декрементом (decrement).

5.2. Инструкция while

Компьютеры часто используются для автоматизации повторяющихся задач. Повторное выполнение одинаковых или почти одинаковых задач без совершения ошибок – это то, что компьютеры делают хорошо, а люди хуже. Поскольку процедура итерации (iteration) так часто встречается, Python предоставляет несколько функциональных возможностей для их упрощения.

Одной из форм итеративной процедуры в Python является инструкция `while`. Ниже приведена простая программа, выполняющая обратный отсчет от пяти, а затем выводящая «Blastoff!» («Пуск!»).

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

Вы сразу же можете прочитать инструкцию `while`, как если бы это была фраза на английском языке. Она означает: «Пока `n` больше 0, вывести его значение, затем уменьшить значение `n` на 1. Когда получится 0, выйти из инструкции `while` и вывести слово `Blastoff!`».

Ниже приведено более формальное описание потока выполнения инструкции `while`:

1) вычислить условие и получить `True` или `False`;
2) если условие ложно, то выйти из инструкции `while` и продолжить выполнение со следующей инструкции;
3) если условие истинно, то выполнить тело инструкции `while`, затем вернуться к шагу 1.

Этот тип потока выполнения называется циклом (loop), потому что третий шаг возвращает процесс выполнения обратно к первому шагу («зацикливает» выполнение). Каждое отдельное выполнение тела этого цикла называется итерацией (iteration). О приведенном выше цикле можно было бы сказать: «Он выполнил пять итераций». Это означает, что тело цикла выполнено пять раз.

Тело цикла должно изменять значение одной или нескольких переменных, чтобы в конце концов условие стало ложным и цикл завершился. Переменные, изменяющиеся при каждом выполнении тела цикла и управляющие его завершением, называются итерационными переменными (iteration variable). При отсутствии итерационной переменной цикл будет выполняться вечно, т. е. становится бесконечным циклом (infinite loop).

5.3. Бесконечные циклы

Неисчерпаемым источником веселья для программистов является инструкция по использованию шампуня «намылить (до появления пены), смыть, повторить», которая представляет собой бесконечный цикл, поскольку здесь нет итерационной переменной, указывающей, сколько раз нужно выполнить этот цикл.

В случае с `countdown` (обратный отсчет) мы можем доказать, что цикл завершается, потому что нам известно, что значение n конечно, и мы можем видеть, что значение n уменьшается при каждом проходе через цикл, так что в конце концов получится 0. В других случаях цикл очевидно является бесконечным, так как в нем вообще нет итерационных переменных.

Иногда мы не знаем, что наступило время для завершения цикла, до тех пор, пока не пройдем половину пути в его теле. В этом случае можно преднамеренно написать бесконечный цикл, а затем использовать инструкцию `break` для немедленного выхода из него.

Приведенный ниже цикл явно является бесконечным, так как логическое выражение в инструкции while представлено просто логической константой `True`:

```
n = 10
while True:
  print(n, end=' ')
  n = n - 1
print('Done!')
```

Если вы совершите ошибку и выполните этот код, то быстро узнаете, как остановить выполняющийся процесс Python в вашей системе, или найдете кнопку отключения электропитания на своем компьютере. Эта программа будет выполняться вечно или пока не сядет батарея, потому что логическое выражение в заголовке цикла всегда истинно в силу того непреложного факта, что этим выражением является постоянное значение `True`.

Хотя это неработоспособный бесконечный цикл, мы все же можем продолжать использовать этот шаблон для создания полезных циклов, если аккуратно добавим в тело цикла код для явного выхода из него с помощью ключевого слова `break`, когда будет выполнено условие выхода.

Например, предположим, что необходимо принимать входные данные от пользователя до тех пор, пока он не введет слово `done`. В этом случае можно написать:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')

# Исходный код: http://www.py4e.com/code3/copytildone1.py
```

Условием цикла является специальное значение `True`, которое всегда истинно, поэтому цикл повторяется многократно до тех пор, пока не сработает инструкция `break`.

На каждой итерации выводится приглашение для пользователя в виде угловой скобки. Если пользователь вводит слово `done`, то инструкция `break` выполняет выход из этого цикла. Иначе программа повторно выводит введенную пользователем строку и возвращается в начало цикла. Ниже приведен пример выполнения:

```
> hello there
hello there
> finished
finished
```

```
> done
Done!
```

Этот способ записи циклов `while` весьма часто применяется, потому что можно проверить условие в любом месте цикла (не только в строке заголовка), и вы можете выразить условие остановки утвердительно («остановиться, когда это произойдет»), а не отрицательно («продолжать выполнение, пока это не произойдет»).

5.4. Завершение отдельных итераций с помощью инструкции `continue`

Иногда вы находитесь внутри итерации некоторого цикла и хотите завершить текущую итерацию и немедленно перейти к следующей. В этом случае можно воспользоваться инструкцией `continue` для перехода к следующей итерации без завершения тела цикла в текущей итерации.

Ниже приведен пример цикла, который копирует вводимые данные до тех пор, пока пользователь не введет слово `done`, но считает, что строки, начинающиеся с хеш-символа (#), выводить не нужно (это что-то вроде комментариев языка Python).

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Исходный код: http://www.py4e.com/code3/copytildone2.py
```

Вот пример выполнения этой новой программы с добавлением инструкции `continue`.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

Выводятся все введенные строки, за исключением одной, которая начинается с хеш-символа, потому что при выполнении инструкции `continue` немедленно завершается текущая итерация и происходит переход обратно к инструкции `while` для начала новой итерации, следовательно, пропускается инструкция `print` внутри цикла.

5.5. Определение циклов с использованием инструкции for

Иногда необходимо пройти в цикле по множеству (набору) объектов, например по списку слов, по строкам в файле или по списку чисел. При наличии списка объектов, перебираемых в цикле, можно создать определенный цикл, воспользовавшись инструкцией `for`. Инструкцию `while` мы называем бесконечным циклом, потому что она просто зацикливает выполнение до тех пор, пока некоторое условие не станет ложным (`False`), тогда как инструкция `for` позволяет пройти в цикле по известному множеству элементов, так что выполняется столько итераций, сколько элементов в этом множестве.

Синтаксис цикла `for` похож на синтаксис цикла `while` в том, что имеется строка инструкции `for` и тело цикла:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

В терминах языка Python переменная `friends` – это список[1] из трех строк, а цикл `for` проходит по этому списку и выполняет тело по одному разу для каждой из этих строк. В результате получаем следующий вывод:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Перевод этого цикла `for` на естественный язык не так очевиден, как в случае `while`, но если считать имена друзей множеством, то можно сказать приблизительно так: «Выполнять инструкции в теле цикла `for` по одному разу для каждого имени друга в этом множестве».

Рассмотрим подробнее цикл: `for` и `in` – зарезервированные слова языка Python, а `friend` и `friends` – переменные.

```
for friend in friends:
    print('Happy New Year:', friend)
```

В частности, `friend` – это итерационная переменная для цикла `for`. Значение переменной `friend` изменяется на каждой итерации и управляет завершением цикла `for`. Итерационная переменная успешно проходит по трем строкам, хранящимся в переменной `friends`.

[1] Мы будем рассматривать списки более подробно в одной из следующих глав.

5.6. Шаблоны цикла

Циклы for и while часто используются для прохода по списку элементов или по содержимому файла, и мы ищем что-то, например наибольшее или наименьшее значение в просматриваемых в цикле данных.

Эти циклы в обобщенном случае создаются следующим образом:

- инициализация одной или нескольких переменных перед началом цикла;
- выполнение некоторого вычисления с каждым элементом в теле цикла, возможно, с изменением переменных непосредственно в теле цикла;
- просмотр полученных в результате значений переменных, когда цикл завершается.

Для демонстрации концепций и конструкций этих шаблонов цикла мы будем использовать список чисел.

5.6.1. Циклы подсчета и суммирования

Например, для подсчета количества элементов в списке можно было бы написать следующий цикл for:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

Перед началом выполнения цикла значение переменной count устанавливается равным нулю, затем мы пишем цикл for для прохода по списку чисел. Итерационная переменная называется itervar, и хотя мы не используем ее напрямую в теле, тем не менее она управляет циклом и заставляет его тело выполняться по одному разу для каждого значения из заданного списка.

В теле цикла мы прибавляем 1 к текущему значению count для каждого числового значения в списке. Во время выполнения цикла значением count является количество значений, которое мы «увидели до текущего момента».

После завершения цикла значением count является общее количество элементов. Это общее количество «падает прямо нам в руки» в конце работы цикла. Мы создаем цикл так, чтобы получить требуемое значение сразу после завершения цикла.

Другой похожий цикл вычисляет общую сумму множества чисел, как показано ниже:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

В этом цикле мы напрямую используем итерационную переменную. Вместо простого прибавления единицы к счетчику count, как в предыдущем цик-

ле, мы прибавляем реальные числа из списка (3, 41, 12 и т. д.) к текущему значению переменной `total` на каждой итерации цикла. Если повнимательнее присмотреться к переменной `total`, то можно заметить, что она содержит «текущую сумму всех ранее пройденных значений». Перед началом цикла значение `total` равно нулю, потому что мы пока еще не видели никаких значений, в процессе выполнения цикла в `total` постепенно накапливается общая сумма, а после окончательного завершения цикла `total` содержит общую сумму всех значений в списке.

При выполнении цикла total накапливает (аккумулирует) сумму элементов, поэтому переменную, используемую таким способом, иногда называют аккумулятором.

Но цикл подсчета и цикл суммирования в действительности не слишком полезны на практике, потому что существуют встроенные функции `len()` и `sum()`, которые вычисляют количество элементов и общую сумму элементов в списке соответственно.

5.6.2. Циклы вычисления максимума и минимума

Для поиска наибольшего значения в списке или последовательности мы создаем следующий цикл:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)
```

Итоговый вывод результата выполнения программы показан ниже:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

Переменную `largest` удобнее всего считать «наибольшим значением из тех, которые мы видели до настоящего момента». Перед началом цикла для `largest` устанавливается значение константы `None`. `None` – это специальное постоянное значение, которое можно хранить в переменной, чтобы пометить ее как «пустую».

До начала выполнения цикла наибольшим значением, которое мы видели до настоящего момента, является `None`, поскольку мы еще не встречали ни одного значения. В процессе выполнения цикла, если `largest` равно `None`, то мы берем первое рассматриваемое значение как максимальное на теку-

щий момент. В первой итерации можно видеть, что когда значение `itervar` равно 3, то поскольку `largest` равно `None`, мы немедленно устанавливаем для `largest` значение 3.

После первой итерации значение `largest` уже не равно `None`, поэтому вторая часть составного логического выражения, проверяющая условие `itervar > largest`, срабатывает, только если мы видим значение, превышающее «наибольшее в текущий момент». Когда мы находим новое, «еще большее» значение, то устанавливаем его для `largest`. При выводе результатов этой программы можно видеть, как значение `largest` постепенно изменяется с 3 на 41, затем на 74.

В конце цикла просканированы все значения, и переменная `largest` теперь действительно содержит максимальное значение в списке.

Для вычисления наименьшего значения код почти такой же, но с одним небольшим изменением:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

И в этом случае `smallest` – «наименьшее значение из тех, что мы видели до настоящего момента» до, во время и после выполнения цикла. После полного завершения цикла `smallest` содержит минимальное значение из заданного списка.

Как и для циклов подсчета и суммирования, встроенные функции `max()` и `min()` делают практически бесполезным написание циклов, продемонстрированных в этом подразделе.

Ниже показана упрощенная версия встроенной в Python функции `min()`:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

В этой версии функции с минимальным исходным кодом удалены все инструкции `print`, чтобы она стала почти равнозначной функции `min()`, уже встроенной в Python.

5.7. Отладка

Когда вы начинаете писать более крупные программы, то, возможно, обнаружите, что тратите больше времени на отладку. Больший объем кода означает увеличение вероятности совершения ошибки и количества мест, где они скрываются.

Одним из способов сокращения времени на отладку является «отладка методом бисекции (деления пополам)». Например, если в вашей программе 100 строк кода и вы проверяете их поочередно по одной, то для этого потребуется 100 шагов.

Вместо этого попробуйте разделить программу пополам. Обратите внимание на середину программы или на близкое к середине место для рассмотрения промежуточного значения, которое можно проверить. Добавьте инструкцию `print` (или что-то другое, позволяющее проверить значение) и запустите программу.

Если проверка в средней точке оказалась некорректной, то проблема непременно находится в первой половине программы. Если результат проверки корректен, то проблема во второй половине.

При каждом выполнении проверки, подобной описанной выше, вы делите пополам число строк, в которых будет выполняться поиск. После шести шагов (это намного меньше, чем 100) вы должны прийти к одной или двум строкам исходного кода, по крайней мере теоретически.

На практике не всегда понятно, что такое «середина программы», и не всегда возможно ее проверить. Не имеет смысла считать строки и искать абсолютную среднюю точку. Вместо этого подумайте о тех местах в программе, где возможны ошибки и которые легко поддаются проверке. Затем выберите небольшой фрагмент, в котором, по вашему мнению, существуют почти равные вероятности того, что ошибка находится до или после инструкций проверки.

5.8. Словарь терминов

- Аккумулятор (accumulator) – переменная, используемая в цикле для суммирования или накопления (аккумуляции) результата.
- Бесконечный цикл (infinite loop) – цикл, в котором условие завершения никогда не выполняется или в котором вообще нет условия завершения.
- Декремент (decrement) – обновление, которое уменьшает значение переменной.
- Инициализация (initialization) – присваивание, которое устанавливает начальное значение переменной, которое будет обновляться в дальнейшем.
- Инкремент (increment) – обновление, которое увеличивает значение переменной (часто на единицу).
- Итерация (iteration) – повторяющееся выполнение группы инструкций, использующее функцию, вызывающую саму себя, или цикл.
- Счетчик (counter) – переменная, используемая в цикле для подсчета количества некоторых событий. Мы инициализируем счетчик нулем, а затем увеличиваем его значение каждый раз, когда нужно что-то «посчитать».

5.9. Упражнения

УПРАЖНЕНИЕ 1 Написать программу, которая многократно считывает числа до тех пор, пока пользователь не введет слово `done`. После ввода `done` выводится общая сумма, количество и среднее арифметическое всех введенных чисел. Если пользователь вводит что-то, отличающееся от числа, то обнаружить эту ошибку с использованием инструкций `try` и `except`, вывести сообщение об ошибке и перейти к вводу следующего числа.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.333333333333333
```


УПРАЖНЕНИЕ 2 Написать другую программу, которая предлагает пользователю ввести список чисел, как в программе из упражнения 1, а в конце выводит максимальное и минимальное числа из этого списка вместо среднего арифметического.

Глава 6

Строки

6.1. Строка – это последовательность

Строка – это последовательность символов. Вы можете получить доступ к одному из символов строки с помощью оператора квадратных скобок:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

Вторая инструкция извлекает символ, индекс которого обозначает позицию 1, из переменной `fruit` и присваивает этот символ переменной `letter`.

Выражение в квадратных скобках называется индексом (index). Индекс определяет, какой именно символ из последовательности вам нужен (отсюда и название).

Но, возможно, вы получили не то, что ожидали:

```
>>> print(letter)
a
```

Для большинства людей первой буквой слова «banana» является «b», а вовсе не «a». Но в Python индекс – это смещение от начала строки, поэтому смещение первой буквы равно нулю.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

Так что «b» – это нулевая буква слова «banana», «a» – первая буква, а «n» – вторая.

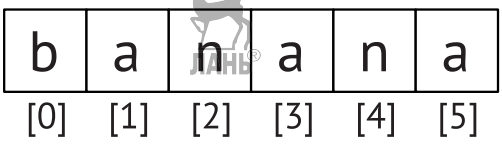


Рис. 6.1 ❖ Индексы строки

В качестве индекса можно использовать любое выражение, включающее переменные и операторы, но значение индекса обязательно должно быть целым числом. Иначе вы получите сообщение об ошибке:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

6.2. Получение длины строки с помощью функции len

Встроенная функция `len` возвращает число символов в строке:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

Для получения самой последней буквы строки, возможно, возникает искушение сделать что-то подобное:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

Причина возникновения ошибки `IndexError` заключается в том, что в слове «banana» нет буквы с индексом 6. Поскольку подсчет начинается с нуля, эти шесть букв нумеруются от 0 до 5. Чтобы получить последний символ, необходимо вычесть 1 из `length`:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

Другой вариант: можно использовать отрицательные индексы, которые позволяют вести отсчет от конца строки. Выражение `fruit[-1]` дает последнюю букву, `fruit[-2]` – предпоследнюю, т. е. вторую с конца, и т. д.

6.3. Проход по строке с использованием цикла

Весьма многие процедуры вычисления предполагают последовательную обработку строки по одному символу за одну итерацию. Часто подобные процедуры устанавливаются на начало строки, перебирают по очереди каждый символ, что-то делают с текущим символом и продолжают выполнение до конца строки. Этот шаблон обработки называется проходом (traversal). Одним из способов написания последовательного прохода является цикл `while`:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

Этот цикл проходит по заданной строке и выводит каждый символ на отдельной строке экрана. Условием цикла является `index < len(fruit)`, поэтому когда `index` равен длине строки, условие становится ложным, и тело цикла не выполняется. Последним доступным является символ с индексом `len(fruit)-1`, т. е. последний символ в этой строке.

УПРАЖНЕНИЕ 1 Написать цикл `while`, который начинает выполнение с последнего символа в заданной строке и продвигается в обратном направлении к первому символу строки, выводя каждый символ в отдельной строке экрана.

Другой способ последовательного прохода – использование цикла `for`:

```
for char in fruit:
    print(char)
```

На каждой итерации этого цикла очередной символ в строке присваивается переменной `char`. Цикл продолжается до тех пор, пока не останется ни одного символа.

6.4. Вырезки строк

Сегмент (часть) строки называется вырезкой (slice). Выбор вырезки похож на выбор символа:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

Оператор квадратных скобок возвращает часть строки, начиная с «n-го» символа и заканчивая «m-м» символом, включая первый, но исключая последний.

Если первый индекс (перед двоеточием) пропущен, то вырезка выполняется с начала строки. Если пропущен второй индекс, то вырезка выполняется до конца строки:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

Если первый индекс больше или равен второму, то результатом будет пустая строка, представленная двумя символами кавычек:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

Пустая строка не содержит символов и имеет длину 0, но во всем остальном это точно такая же строка, как и любая другая.

УПРАЖНЕНИЕ 2 Предположим, что `fruit` – это строка, тогда что означает `fruit[:]`?

6.5. Строки неизменяемы

Возникает соблазн использовать оператор квадратных скобок в левой части инструкции присваивания с намерением изменить какой-либо символ в строке. Например:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

В этом случае «объектом» (object) является строка, а «элементом» (item) – символ, который вы попытались присвоить. В данный момент объект – это то же самое, что и значение, но немного позже мы дадим более точное и полное определение. Элемент (item) – это одно из значений в последовательности.

Причина этой ошибки заключается в том, что строки являются неизменяемыми (immutable), т. е. вы не можете изменить существующую строку. Самое лучшее, что можно сделать в этом случае, – создать новую строку, являющуюся вариантом исходной:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

В этом примере выполняется конкатенация (сцепление) новой первой буквы с вырезкой из строки `greeting`. Это не оказывает никакого воздействия на исходную строку.

6.6. Работа в цикле и подсчет

Приведенная ниже программа подсчитывает число появлений буквы «a» в заданной строке:

```
word = 'banana'
```

```
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

Эта программа демонстрирует еще один шаблон вычислений, называемый счетчиком (counter). Переменная `count` инициализируется нулем, а затем инкрементируется при каждом обнаружении буквы «a». После выхода из цикла `count` содержит результат: общее число найденных букв «a».

УПРАЖНЕНИЕ 3 Включить (инкапсулировать) приведенный выше код в функцию `count` и обобщить ее определение так, чтобы функция принимала строку и букву как аргументы.

6.7. Оператор `in`

Ключевое слово `in` – это логический оператор, который принимает две строки и возвращает `True`, если первая присутствует как подстрока во второй:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

6.8. Сравнение строк

Операторы сравнения работают со строками. Чтобы убедиться, что две строки равны:

```
if word == 'banana':
    print('All right, bananas.')
```

Другие операторы сравнения полезны для размещения слов в алфавитном порядке:

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python не обрабатывает буквы верхнего и нижнего регистров так, как это делают люди. Все буквы в верхнем регистре (прописные) по порядку располагаются перед всеми буквами в нижнем регистре (строчными), поэтому:

```
Your word, Pineapple, comes before banana.
```

Общепринятым способом решения этой проблемы является преобразование строк в единый стандартный формат, например всех букв в нижний регистр, перед выполнением сравнения. Помните об этом, когда приходится защищаться от человека, вооруженного «лимонкой»[1].

6.9. Методы строк

Строки представляют собой пример объектов (objects) языка Python. Объект содержит данные (сами строки) и методы (methods) – эффективные функции, встроенные в объект и доступные любому экземпляру (instance) этого объекта.

В Python есть функция `dir`, выводящая список методов, доступных для какого-либо объекта. Функция `type` показывает тип объекта, а функция `dir` – его доступные методы.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
>>>
```

Хотя функция `dir` выводит список всех методов и вы можете использовать команду `help` для получения некоторой простой документации по любому методу, все же более подробный источник документации по методам строк находится здесь: https://docs.python.org/library/stdtypes.html#string-methods.

Вызов метода похож на вызов функции (метод принимает аргументы и возвращает значение), но синтаксис отличается. Мы вызываем метод, добавляя его имя к имени переменной и используя точку как разделитель.

---

[1] В оригинальном тексте – Pineapple – и ананас (фрукт), и «лимонка» (ручная граната). – *Прим. перев.*

Например, метод `upper` принимает строку и возвращает новую строку, в которой все буквы переведены в верхний регистр. Здесь вместо синтаксиса вызова функции `upper(word)` используется синтаксис вызова метода `word.upper()`:

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

Эта форма точечной записи (записи через точку) определяет имя метода `upper` и имя строки, к которой применяется этот метод, `word`. Пустые круглые скобки означают, что этот метод не принимает какие-либо аргументы.

Обращение к методу называется вызовом (invocation). В рассматриваемом здесь примере следует сказать, что мы вызвали метод `upper` для строки `word`.

Например, существует метод `find`, выполняющий поиск позиции одной строки внутри другой:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

В этом примере мы вызываем метод `find` для строки `word` и передаем в него искомую букву как параметр.

Метод `find` может искать не только отдельные символы, но и подстроки:

```
>>> word.find('na')
2
```

Кроме того, метод `find` может принимать как второй аргумент индекс, с которого нужно начать поиск:

```
>>> word.find('na', 3)
4
```

Одной из часто встречающихся задач является удаление пробельных символов (пробелов, табуляций или символов перехода на новую строку) из начала и конца строки при помощи метода `strip`:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Некоторые методы, например `startswith`, возвращают логические значения.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

Вероятно, вы заметили, что `startswith` требует совпадения не только символа, но и регистра, поэтому иногда перед проверкой мы выполняем преобразование букв строки в нижний регистр с помощью метода `lower`.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
>>> line.lower().startswith('h')
True
```

В последнем примере вызывается метод `lower`, затем мы используем метод `startswith`, чтобы узнать, начинается ли полученная строка с буквами в нижнем регистре с буквы «h». Если внимательно следить за порядком, то можно выполнять несколько вызовов методов в одном выражении.

УПРАЖНЕНИЕ 4. Для строк существует метод `count`, похожий на функцию из предыдущего упражнения. Изучите документацию по методу `count` здесь:

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

Написать вызов, который подсчитывает, сколько раз встречается заданная конкретная буква в слове «banana».

6.10. Синтаксический разбор (парсинг) строк

Часто необходимо рассмотреть некоторую строку и найти в ней определенную подстроку. Например, если бы нам предоставили последовательность строк, отформатированных следующим образом:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

и при этом требуется извлечь только вторую часть адреса (т. е. `uct.ac.za`) из каждой строки, то мы можем сделать это, используя метод `find` и вырезку из строки.

Сначала найдем позицию символа «коммерческое at» (@) в строке. Затем найдем позицию первого пробела после символа @. Далее воспользуемся операцией вырезки подстроки для извлечения той части строки, которую мы ищем.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
```

```
uct.ac.za
>>>
```

Мы используем версию метода `find`, позволяющую задать позицию в строке, с которой необходимо начать поиск. При выполнении вырезки мы извлекаем символы, начиная «со следующего за символом @ и до первого найденного пробельного символа, не включая этот символ».

Документация по методу `find` доступна здесь:

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

6.11. Оператор формата

Оператор формата % позволяет формировать строки, заменяя их определенные части на значения данных, хранящихся в некоторых переменных. В применении к целым числам % является оператором деления по модулю. Но если первым операндом является строка, то % становится оператором формата.

Первый операнд – строка формата (format string), содержащая одну или несколько последовательностей формата, которые определяют, как форматируется второй операнд. Результатом является строка.

Например, последовательность формата `%d` означает, что второй операнд должен быть отформатирован как целое число (`d` обозначает десятичное (decimal) целое число):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

Результатом является строка `'42'`, которую не следует путать с целочисленным значением 42.

Последовательность формата может находиться в любом месте строки, поэтому можно включать значение переменной в предложение:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```


Если в строке содержится более одной последовательности формата, то второй аргумент должен быть кортежем[1]. Каждой последовательности формата соответствует элемент кортежа, по порядку.

В следующем примере используется `%d` для форматирования целого числа, `%g` для форматирования числа с плавающей точкой (не спрашивайте, почему) и `%s` для форматирования строки:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

[1] Кортеж (tuple) – это последовательность значений, разделенных запятыми, внутри пары скобок. Кортежи будут рассматриваться в главе 10.

Число элементов в кортеже должно в точности соответствовать числу последовательностей формата в строке. Типы элементов также должны соответствовать заданным последовательностям формата:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

В первом примере недостаточное число элементов в кортеже, во втором – элемент имеет неправильный тип.

Оператор формата обладает мощными возможностями, но его практическое применение может быть связано с определенными трудностями. Более подробно об операторе формата можно прочитать здесь:

https://docs.python.org/library/stdtypes.html#printf-style-string-formatting.

6.12. Отладка

Навык, который вы должны развивать при написании программ, – это привычка постоянно спрашивать себя: «Что здесь может быть неправильным?» или немного по-другому: «Какое безумное действие может выполнить пользователь, чтобы нарушить работу нашей превосходной (как мы считаем) программы?»

Например, рассмотрим программу, которая использовалась для демонстрации работы цикла `while` в главе об итерациях (глава 5):

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Исходный код: http://www.py4e.com/code3/copytildone2.py
```

Посмотрим, что происходит, если пользователь вводит пустую строку:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

Этот код работает нормально до тех пор, пока не введена пустая строка. И поскольку здесь нет нулевого символа, мы получаем обратную трассировку. Существуют два решения этой проблемы, позволяющих сделать третью строку «безопасной», даже если введена пустая строка.

Один возможный вариант – просто воспользоваться методом `startswith`, который возвращает `False`, если строка пустая.

```
if line.startswith('#'):
```

Другой способ – написать защищенную инструкцию `if` с использованием шаблона guardian (защитник), проверить и убедиться в том, что второе логическое выражение вычисляется только в том случае, если в строке есть хотя бы один символ:

```
if len(line) > 0 and line[0] == '#':
```

6.13. Словарь терминов

- Вызов (invocation) – инструкция, которая вызывает метод.
- Вырезка (slice) – часть строки, определяемая диапазоном индексов.
- Индекс (index) – целочисленное значение, используемое для выбора элемента в последовательности, например символа в строке.
- Метод (method) – функция, связанная с объектом и вызываемая с использованием точечной записи.
- Неизменяемость (immutable) – свойство последовательности, элементам которой нельзя присваивать другие значения.
- Объект (object) – сущность, на которую может ссылаться переменная. В настоящий момент вы можете использовать термины «объект» и «значение» как синонимы.
- Оператор формата (format operator) – оператор %, который принимает строку формата и кортеж, затем генерирует строку, включающую элементы кортежа, отформатированные по содержимому строки формата.
- Поиск (search) – шаблон процедуры прохода, который останавливается, когда находит заданный искомый элемент.
- Последовательность (sequence) – упорядоченное множество, т. е. множество значений, в котором каждое значение идентифицируется по целочисленному индексу.
- Последовательность формата (format sequence) – последовательность символов в строке формата, например `%d`, которая определяет, как должно быть отформатировано значение.
- Проход (traverse) – итеративный проход по элементам в последовательности с выполнением одной и той же операции для каждого элемента.
- Пустая строка (empty string) – строка без символов длиной 0, представленная двумя символами кавычек.
- Строка формата (format string) – строка, используемая с оператором формата, которая содержит последовательности формата.

- Счетчик (counter) – переменная, используемая для подсчета чего-либо, обычно инициализируется нулем, а затем инкрементируется.
- Флаг (flag) – логическая переменная, используемая для определения, является ли условие истинным или ложным.
- Элемент (item) – одно из значений в последовательности.

6.14. Упражнения

УПРАЖНЕНИЕ 5 Задан следующий код Python, сохраняющий строку в переменной:

```
str = 'X-DSPAM-Confidence:0.8475'
```

Использовать метод `find` и вырезку строки для извлечения части строки после символа двоеточия, затем применить функцию `float` для преобразования извлеченной строки в число с плавающей точкой.

УПРАЖНЕНИЕ 6 Изучить документацию по методам строк, представленную здесь:

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

Возможно, потребуются эксперименты с некоторыми методами, чтобы быть уверенным в том, что вы понимаете, как они работают. Особенно полезны методы `strip` и `replace`.

В документации используется синтаксис, который может показаться не совсем понятным. Например, в описании метода `find(sub[, start[, end]])` квадратные скобки обозначают необязательные аргументы. То есть `sub` – обязательный аргумент, но `start` – необязательный. А если вы используете `start`, то `end` – необязательный аргумент.

Глава 7

Файлы

7.1. Длительное хранение данных

К настоящему моменту мы научились писать программы и сообщать свои намерения центральному процессорному устройству (ЦПУ), используя условное выполнение, функции и итерации. Мы узнали, как создавать и использовать структуры данных в основной памяти. ЦПУ и память – это устройства, в которых работают и выполняются наши программы. Именно здесь происходит весь процесс «мышления».

Но следует вспомнить из описания архитектуры аппаратного обеспечения, что после выключения электропитания все, что хранится в ЦПУ и в основной памяти, стирается безвозвратно. Так что до сих пор наши программы были всего лишь кратковременными забавными упражнениями для изучения языка Python.

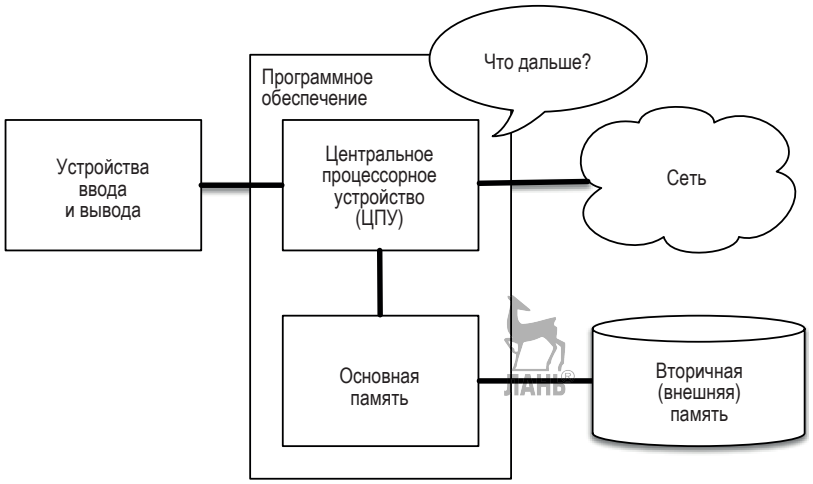


Рис. 7.1 ❖ Вторичная (внешняя) память

В этой главе мы начнем работать с вторичной (внешней) памятью (т. е. с файлами). Вторичная память не стирается при выключении электропита-

ния компьютера. Или в случае использования USB-флеш-накопителя данные, записанные из наших программ, могут быть удалены из системы и перенесены в другую систему.

Основное внимание будет сосредоточено на чтении и записи текстовых файлов, точно таких же, как мы создаем в текстовом редакторе. Позже мы увидим, как работать с файлами баз данных, которые являются двоичными (бинарными) файлами, специально предназначенными для чтения и записи с помощью программного обеспечения систем управления базами данных.

7.2. Открытие файлов

Если необходимо прочитать или записать файл (например, на жесткий диск), то сначала обязательно нужно открыть (open) файл. Операция открытия файла взаимодействует с операционной системой, которая знает, где хранятся данные каждого файла. При открытии файла вы просите операционную систему найти файл по имени и убедиться в том, что этот файл действительно существует. В приведенном ниже примере мы открываем файл *mbox.txt*, который должен храниться в той же папке (каталоге), из которой вы запустили интерпретатор Python. Этот файл можно скачать здесь: www.py4e.com/code3/mbox.txt.

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

Если функция `open` выполнена успешно, то операционная система возвращает дескриптор файла (file handle). Дескриптор файла – это не сами данные, содержащиеся в файле, а его «описание», которое можно использовать для чтения данных. Вы получаете дескриптор, если запрошенный файл существует и вы обладаете правами, необходимыми для чтения этого файла.

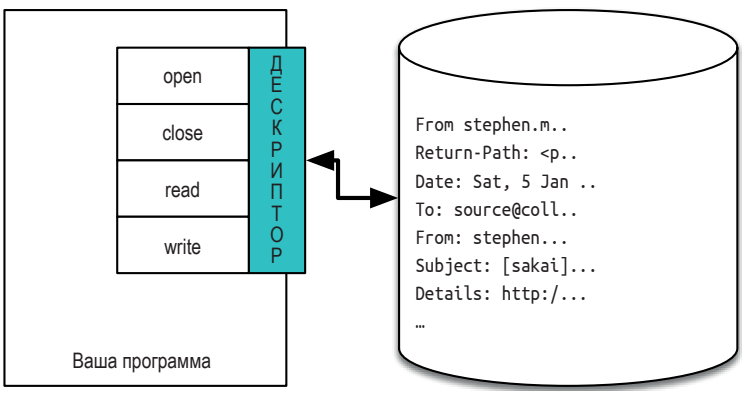


Рис. 7.2 ❖ Дескриптор файла

Если запрошенный файл не существует, то функция open завершится неудачно с обратной трассировкой, и вы не получите дескриптор для доступа к содержимому файла:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Позже мы воспользуемся инструкциями try и except для более аккуратной обработки в ситуации, когда мы пытаемся открыть (возможно) несуществующий файл.

7.3. Текстовые файлы и строки в них

Текстовый файл можно считать последовательностью строк текста практически так же, как строку в языке Python можно считать последовательностью символов. Например, ниже приведено содержимое текстового файла с записями об обработке сообщений электронной почты от различных отправителей в команде разработки проекта с открытым исходным кодом:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```


Полностью этот файл фиксации обмена электронной почтой доступен здесь:

www.py4e.com/code3/mbox.txt,

а его укороченная версия находится здесь:

www.py4e.com/code3/mbox-short.txt.

Эти файлы записаны в стандартном формате файла, содержащего многочисленные сообщения электронной почты. Строки, начинающиеся со слова «From», разделяют сообщения, а строки, начинающиеся с «From:», являются частью сообщений. Для получения более подробной информации о формате mbox см. «Википедию»: https://en.wikipedia.org/wiki/Mbox и https://ru.wikipedia.org/wiki/Mbox.

Для разделения файла на отдельные строки существует специальный символ, который представляет «конец строки» и называется символом перехода на новую строку (newline character).

В языке Python символ перехода на новую строку представлен как пара символов обратный слеш и буква n (\n) в строковых константах. Несмотря

на то что это выглядит как два символа, в действительности это один символ. Когда мы рассматриваем переменную, вводя «stuff» в интерпретаторе, то видим \n в строке, но при использовании функции print для ее вывода оказывается, что эта символьная строка разделена на две экранные строки символом перехода на новую строку.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

Также можно видеть, что длина строки X\nY равна трем символам, потому что символ перехода на новую строку тоже является отдельным полноправным символом.

Поэтому когда мы рассматриваем строки текста в файле, то необходимо мысленно представить, что здесь существует специальный невидимый символ перехода на новую строку, обозначающий конец каждой строки.

Итак, символ перехода на новую строку разделяет символы в файле на отдельные строки.

7.4. Чтение файлов

Несмотря на то что дескриптор файла не содержит сами данные этого файла, относительно просто создать цикл for для последовательного чтения и подсчета строк в файле:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

# Исходный код: http://www.py4e.com/code3/open.py
```

Дескриптор файла можно использовать как последовательность в показанном здесь цикле for. Этот цикл for просто подсчитывает количество строк в файле и выводит результат подсчета. Приблизительный перевод цикла for на обычный язык: «Для каждой строки в файле, представленном этим дескриптором, прибавлять единицу к переменной count».

Причина, по которой функция open не читает весь файл целиком, заключается в том, что файл может оказаться весьма большим и содержать несколько гигабайтов данных. Инструкция open выполняется за одно и то же время вне

зависимости от размера открываемого файла. В цикле `for` организовано действительное чтение данных из открытого файла.

При чтении файла таким способом с применением цикла `for` Python сам заботится о разделении данных из файла на отдельные строки, используя символ перехода на новую строку. Python читает каждую строку до символа перехода на новую строку и включает этот символ в переменную `line` на каждой итерации цикла `for`.

Поскольку цикл `for` читает данные поочередно по одной строке, он может эффективно считывать и подсчитывать строки в весьма больших файлах без исчерпания основной памяти для хранения этих данных. Приведенная выше программа может подсчитывать строки в файле любого размера, используя очень маленький объем памяти, так как каждая строка читается, подсчитывается, а затем отбрасывается.

Если вам известно, что файл относительно невелик по сравнению с объемом основной памяти вашего компьютера, то можно прочитать весь файл целиком в одну строку, воспользовавшись методом `read` для дескриптора файла.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

В этом примере все содержимое (все 94 626 символов) файла *mbox-short.txt* считывается прямо в переменную `inp`. Потом мы используем вырезку, чтобы вывести первые 20 символов строки данных, хранящихся в `inp`.

При чтении файла таким способом все символы, включая символы перехода на новую строку, представляют собой одну большую строку в переменной `inp`. Неплохая мысль – сохранить вывод метода `read` как переменную, потому что каждый вызов `read` расходует ресурс:

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```


Следует помнить, что такая форма функции `open` должна использоваться, только если файл данных может успешно разместиться в основной памяти вашего компьютера. Если файл слишком велик для размещения в основной памяти, то необходимо написать программу так, чтобы этот файл считывался фрагментами, применяя для этого цикл `for` или `while`.

7.5. Поиск в файле

При поиске по данным в файле весьма часто применяется шаблон чтения его содержимого, игнорирующий большинство строк и обрабатывающий только

строки, удовлетворяющие конкретному условию. Можно объединить этот шаблон чтения файла с методами строк для создания простых механизмов поиска.

Например, если необходимо читать файл и выводить только строки, начинающиеся с префикса «From:», то можно воспользоваться методом строки `startwith` для выбора строк с требуемым префиксом:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Исходный код: http://www.py4e.com/code3/search1.py
```

При выполнении этой программы получим следующий вывод:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```


Выведенные данные выглядят отлично, поскольку мы видим только те строки, которые начинаются с «From:», но почему между ними расположены дополнительные пустые строки? Причина в невидимом символе перехода на новую строку (`newline`). Каждая строка файла завершается символом перехода на новую строку, а инструкция `print` выводит строку из переменной `line`, включающую символ `newline`, затем `print` добавляет еще один символ `newline`, а в результате мы наблюдаем эффект удвоения расстояния между строками.

Можно было бы воспользоваться вырезкой строки для вывода ее содержимого без последнего символа, но есть более простой подход – использовать метод `rstrip`, удаляющий все пробельные символы справа от строки, как показано ниже:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)

# Исходный код: http://www.py4e.com/code3/search2.py
```

При выполнении этой программы получим следующий вывод:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
```

```
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

Когда программы обработки файлов становятся более сложными, может потребоваться структурирование циклов поиска с использованием инструкции `continue`. Основная идея цикла поиска заключается в поиске «интересующих нас» строк и эффективном пропуске строк, «для нас неинтересных». А когда мы обнаруживаем интересующую нас строку, мы что-то с ней делаем.

Структурировать цикл можно с помощью шаблона пропуска не интересующих нас строк, как показано ниже:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Пропуск 'не интересующих нас' строк.
    if not line.startswith('From:'):
        continue
    # Обработка 'интересующей нас' строки.
    print(line)

# Исходный код: http://www.py4e.com/code3/search3.py
```

Вывод этой программы тот же, что и в предыдущем примере. Объяснение на естественном языке: не интересующие нас строки не начинаются с префикса «From:» и пропускаются с помощью инструкции `continue`. Для «интересующих нас» строк (т. е. начинающихся с префикса «From:») выполняется определенная обработка.

Можно использовать метод строк `find` для имитации поиска в текстовом редакторе, который позволяет находить искомую подстроку в любом месте текстовой строки. Поскольку `find` ищет вхождение заданной строки в другую строку и возвращает позицию найденной подстроки или –1, если подстрока не найдена, можно написать следующий цикл для вывода строк, содержащих подстроку «@uct.ac.za» (т. е. сообщения из Кейптаунского университета в Южной Африке):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Исходный код: http://www.py4e.com/code3/search4.py
```

Вывод этой программы показан ниже:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
```

```
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

В рассматриваемом здесь примере также используется более компактная форма инструкции `if`, в которой инструкция `continue` расположена на той же строке, что и ключевое слово `if`. Эта компактная форма работает точно так же, как если бы инструкция `continue` располагалась на следующей строке и была сдвинута вправо.

7.6. Предоставление пользователю выбора имени файла

Редактировать исходный код Python каждый раз, когда требуется обработка другого файла, весьма нежелательно. Было бы более удобно предложить пользователю ввести строку с именем файла при каждом запуске программы, чтобы можно было использовать ее для обработки различных файлов без изменения исходного кода.

Сделать это достаточно просто, если считывать имя файла, вводимое пользователем, используя функцию `input`, как показано ниже:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Исходный код: http://www.py4e.com/code3/search6.py
```

Мы читаем имя файла, введенное пользователем, и помещаем его в переменную `fname`, затем открываем файл с этим именем. Теперь можно многократно выполнять эту программу для различных файлов.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Прежде чем перейти к чтению следующего раздела, посмотрите на код приведенной выше программы и спросите себя: «Что в ней может пойти не так?» или «Что наш друг-пользователь мог бы натворить, чтобы заставить эту прекрасную маленькую программу завершиться с ошибкой и обратной трассировкой, сделав нас не такими уж крутыми в глазах пользователей?».

7.7. Использование try, except и open

Я же сказал: не подглядывать. Даю последний шанс.

Что, если пользователь вводит нечто, не являющееся именем файла?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

Не надо смеяться. Пользователи, в конце концов, делают все возможное, чтобы «сломать» ваши программы намеренно или даже со злым умыслом. В сущности говоря, важной частью каждой команды разработки программного обеспечения является человек или группа, называемая Quality Assurance (QA; обеспечение контроля качества), задача которой состоит исключительно в том, чтобы проделывать всевозможные безумные действия в попытках «сломать» программное обеспечение, разработанное программистом.

Группа QA отвечает за поиск дефектов в программах до того, как они будут предоставлены конечным пользователям, которые, возможно, приобретают программное обеспечение или оплачивают его разработку. Поэтому группа QA – лучший друг программиста.

Теперь, когда мы обнаружили дефект в своей программе, можно аккуратно устранить его, используя структуру `try/except`. Следует предположить, что вызов функции `open` может завершиться неудачно, и добавить код восстановления для случая ошибки при вызове `open`, как показано ниже:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Исходный код: http://www.py4e.com/code3/search7.py
```

Функция `exit` завершает выполнение программы. Это функция, которая никогда не возвращается после вызова. Теперь если пользователь (или

группа QA) вводит какую-нибудь чушь или неверные имена файлов, то мы «перехватываем» их и аккуратно восстанавливаем выполнение программы с нормальным ее завершением:

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Защита вызова `open` – хороший пример правильного применения инструкций `try` и `except` в программах на языке Python. Термин «Pythonic» («питонический») используется, когда мы что-то делаем «способом, присущим языку Python» («Python way»). Можно сказать, что приведенный выше пример является способом, присущим языку Python, для открытия файла.

Когда вы станете более опытным программистом на Python, то сможете состязаться в остроумии с другими Python-программистами, решая, которое из двух равнозначных решений одной задачи является «более питоническим». Стремление стать «более питоническим» подтверждает точку зрения, утверждающую, что программирование частично является инженерной дисциплиной, а частично – искусством. Мы не всегда заинтересованы только лишь в простом выполнении некоторой работы, нам также хочется, чтобы решение было изящным и оценивалось как изящное нашими коллегами.

7.8. Запись в файлы

Для записи в файл необходимо открыть его в режиме «w» (write – запись), заданном как второй параметр:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

Если заданный файл уже существует, то его открытие в режиме записи стирает все старые данные и начинает «с чистого листа», так что будьте осторожны. Если заданный файл не существует, то создается новый.

Метод `write` объекта дескриптора файла записывает данные в файл и возвращает количество записанных символов. По умолчанию режим записи подразумевает текст для записи (и чтения) строк.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

Объект файла всегда следит за тем, где он находится в текущий момент, поэтому при повторном вызове `write` новые данные добавляются в конец файла.

Мы обязательно должны управлять концами строк при записи в файл, явно вставляя символ перехода на новую строку (newline), когда это необходимо. Инструкция `print` автоматически добавляет newline, но метод `write` не добавляет этот символ автоматически.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

Когда запись завершена, необходимо закрыть файл для полной уверенности в том, что самый последний фрагмент данных физически записан на диск и не будет потерян при отключении электропитания компьютера.

```
>>> fout.close()
```

Можно было бы закрыть и те файлы, которые открыты для чтения, но при этом возможны небольшие нестыковки, если открыто несколько файлов, поскольку Python всегда обеспечивает закрытие всех открытых файлов при завершении программы. При записи файлов необходимо явно закрывать эти файлы, чтобы не возникало никаких случайностей.

7.9. Отладка

При чтении и записи файлов вы, вероятно, столкнетесь с проблемами использования пробельных символов. Подобные ошибки трудно отлаживать, потому что пробелы, табуляции и символы перехода на новую строку обычно невидимы:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2 3
4
```

Здесь может помочь встроенная функция `repr`. Она принимает как аргумент любой объект и возвращает его представление в форме строки. Для строк эта функция представляет пробельные символы с помощью последовательностей с обратным слешем:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Это может оказаться полезным при отладке.

Может возникать и другая проблема из-за того, что различные системы используют разные символы для обозначения конца строки. Некоторые системы используют символ перехода на новую строку (newline), представленный последовательностью `\n`. В других применяется символ «возврат каретки» (return), представленный последовательностью `\r`. В некоторых системах используется комбинация этих двух последовательностей `\n\r`. Если вы пере-

даете файлы между различными системами, то эти несоответствия могут приводить к проблемам.

Для большинства систем существуют приложения, выполняющие преобразования из одного формата в другой. Такие приложения можно найти (и узнать подробнее об этой проблеме) здесь: https://www.wikipedia.org/wiki/Newline. Разумеется, вы можете самостоятельно написать подобное приложение.

7.10. Словарь терминов

- Контроль качества (Quality Assurance – QA) – человек или группа, сосредоточенная на обеспечении общего качества программного продукта. QA часто осуществляется на стадии тестирования продукта и выявления проблем до его выпуска.
- Перехват (catch) – предотвращение ошибки (исключения), вызывающей некорректное завершение программы, с использованием инструкций `try` и `except`.
- Переход на новую строку (символ) (newline) – специальный символ, используемый в файлах и строках для обозначения конца строки.
- Питонический (Pythonic) – технический прием, который естественно и изящно («элегантно») работает в Python. «Использование try и except – это питонический способ восстановления выполнения программы при попытке открыть несуществующие файлы».
- Текстовый файл (text file) – последовательность символов, сохраненная в постоянном хранилище, например на жестком диске.

7.11. Упражнения

УПРАЖНЕНИЕ 1 Написать программу чтения и вывода содержимого файла (поочередно по одной строке) с переводом всех букв в верхний регистр. При выполнении программы ее вывод должен выглядеть следующим образом:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN 5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

Файл можно загрузить здесь: www.py4e.com/code3/mbox-short.txt.

УПРАЖНЕНИЕ 2 Написать программу, которая предлагает ввести имя файла, а затем считывает его содержимое и ищет строки в форме:

```
X-DSPAM-Confidence: 0.8475
```

Когда встречается строка, начинающаяся с «X-DSPAM-Confidence:», взять эту строку для извлечения из нее числа с плавающей точкой. Подсчитать эти строки, затем вычислить общую сумму значений вероятности спама из этих строк.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Протестировать эту программу на файлах *mbox.txt* и *mbox-short.txt*.

УПРАЖНЕНИЕ 3 Иногда скучающие или желающие немного повеселиться программисты добавляют в свои программы безобидные «пасхальные яйца» (Easter Egg). Измените программу, запрашивающую имя файла, так, чтобы она выводила забавное сообщение, когда пользователь вместо имени файла вводит в точности фразу «na na boo boo». Программа должна вести себя нормально при вводе всех прочих имен файлов, как существующих, так и несуществующих. Ниже показан пример выполнения этой программы:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

Мы не поощряем введение подобных «пасхальных яиц» в ваши программы – это просто упражнение.

Глава 8

Списки

8.1. Список – это последовательность

Как и строка, список (list) – это последовательность значений. В строке значениями являются символы, в списке значения могут иметь любой тип. Значения в списке называются элементами (elements или items).

Существует несколько способов создания нового списка. Самый простой из них – заключение элементов в квадратные скобки («[« и «]»):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

Первый пример представляет собой список из четырех целых чисел. Второй – список из трех строк. Элементы списка не обязательно должны принадлежать к одному и тому же типу. Приведенный ниже список содержит строку, число с плавающей точкой, целое число и (внимание!) другой список:

```
['spam', 2.0, 5, [10, 20]]
```

Список, находящийся внутри другого списка, является вложенным (nested).

Список, который не содержит элементов, называется пустым (empty list), его можно создать, просто вводя пустые квадратные скобки: [].

Как можно предположить, значение типа список можно присваивать переменной:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2. Списки – изменяемые объекты

Синтаксис доступа к элементам списка аналогичен синтаксису доступа к символам в строке: оператор квадратные скобки. Выражение внутри квадратных скобок определяет индекс. Напомню, что индексы начинаются с 0:

```
>>> print(cheeses[0])
Cheddar
```

В отличие от строк, списки являются изменяемыми, поскольку вы можете изменять порядок элементов или переприсваивать элементы в списке. Если оператор квадратные скобки располагается в левой части инструкции присваивания, то он определяет элемент списка, которому будет присвоено значение.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

Первый элемент списка `numbers`, которым раньше было число 123, теперь представлен числом 5.

Можно считать список взаимоотношением между индексами и элементами. Такое взаимоотношение называется отображением (mapping), т. е. каждый индекс «отображается» в один из элементов.

Индексы списка работают точно так же, как индексы строки:

- любое целочисленное выражение можно использовать как индекс;
- при попытке чтения или записи элемента, который не существует, возникает ошибка `IndexError`;
- если индекс имеет отрицательное значение, то отсчет ведется от конца списка.

Оператор `in` также работает со списками.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

8.3. Проход по списку

Наиболее часто применяемым способом прохода по элементам списка является использование цикла `for`. Синтаксис точно такой же, как для строк:

```
for cheese in cheeses:
    print(cheese)
```

Этот способ хорошо работает, если нужно только лишь считывать элементы списка. Но если необходимо записывать или обновлять элементы, то требуются индексы. Для этого существует широко известный способ – сочетание функций `range` и `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Этот цикл проходит по списку и обновляет каждый элемент. Функция `len` возвращает число элементов в списке. Функция `range` возвращает список индексов от 0 до n – 1, где n – длина списка. На каждой итерации цикла переменная `i` получает индекс очередного элемента. Инструкция присваивания в теле цикла использует `i` для чтения старого значения элемента и присваивания нового значения.

Цикл `for` для пустого списка никогда не выполняет тело:

```
for x in empty:
    print('This never happens.')
```

Несмотря на то что список может содержать другой список, такой вложенный список продолжает считаться одним элементом. Например, длина приведенного ниже списка равна четырем:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4. Операции со списками

Оператор + выполняет конкатенацию (сцепление) списков:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Оператор * повторяет список заданное число раз:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

В первом примере список повторяется четыре раза. Во втором – три раза.

8.5. Вырезка из списка

Оператор вырезки также работает со списками:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Если пропущен первый индекс, то вырезка начинается с начала списка. Если пропущен второй индекс, то вырезка продолжается до конца списка. А если пропущены оба индекса, то вырезка является копией целого списка.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Поскольку списки изменяемы, зачастую полезно сделать копию списка перед выполнением операций, которые свертывают, вращают или искажают списки.

Оператор вырезки в левой части инструкции присваивания может обновлять сразу несколько элементов:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

8.6. Методы списков

Python предоставляет методы, которые работают со списками. Например, метод `append` добавляет новый элемент в конец списка:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

Метод `extend` принимает список как аргумент и добавляет все его элементы к исходному (вызывающему) списку:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

В этом примере список `t2` не изменяется.

Метод `sort` упорядочивает элементы в списке от самого малого значения до самого большого (в возрастающем порядке):

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Методы списков в большинстве своем пустые – они изменяют список и возвращают специальное значение `None`. Если вы случайно напишете `t = t.sort()`, то будете разочарованы результатом.

8.7. Удаление элементов

Существует несколько способов удаления элементов из списка. Если известен индекс требуемого элемента, то можно воспользоваться методом `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

Метод `pop` изменяет список и возвращает элемент, который был удален. Если индекс не задан, то метод удаляет и возвращает самый последний элемент.

Если удаляемое значение не нужно, то можно использовать оператор `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

Если известен сам элемент, который нужно удалить (но не его индекс), можно воспользоваться методом `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

Метод `remove` возвращает специальное значение `None`.

Для удаления более одного элемента можно использовать оператор `del` с вырезкой по индексу:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

Как обычно, вырезка выбирает все элементы от первого до второго указанного индекса, не включая сам второй индекс.

8.8. Списки и функции

Существует несколько встроенных функций, которые можно использовать для списков и которые позволяют быстро выполнить проход по списку без написания циклов:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
```

```
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

Функция `sum` работает, только если все элементы списка являются числами. Прочие функции (`max()`, `len()` и т. д.) работают со списками строк и другими типами, которые можно сравнивать.

Приведенный ранее пример программы, вычисляющей среднее арифметическое списка чисел, введенных пользователем, можно было бы переписать с использованием списка.

Сначала программа вычисляет среднее арифметическое без списка:

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)
```


```
# Исходный код: http://www.py4e.com/code3/avenum.py
```

В этой программе имеются переменные `count` и `total` для хранения числа и текущей суммы значений, введенных пользователем после повторяющегося запроса очередного числа.

Можно было бы просто запоминать каждое число, после того как пользователь ввел его, и воспользоваться встроенными функциями для вычисления суммы и количества чисел после завершения ввода.

```
numlist = list()
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)
```

```
# Исходный код: http://www.py4e.com/code3/avelist.py
```

Здесь перед началом цикла создается пустой список, затем при каждом получении очередного числа оно добавляется в этот список. В конце про-

граммы мы просто вычисляем сумму чисел в списке и делим ее на счетчик чисел в списке для получения среднего арифметического.

8.9. Списки и строки

Строка – это последовательность символов, а список – это последовательность значений, но список символов – это не то же самое, что строка. Для преобразования строки в список символов можно использовать функцию `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Поскольку `list` представляет собой имя встроенной функции, вы должны избегать его использования в качестве имени переменной. Я также избегаю использования буквы «l», так как она очень похожа на число «1». Именно поэтому я предпочитаю именовать список буквой «t».

Функция `list` разделяет строку на отдельные буквы. Если необходимо разделить строку на слова, то можно воспользоваться методом строки `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

После разделения с помощью метода `split` строки на список слов можно применить оператор индекса (квадратные скобки) для выбора конкретного слова в этом списке.

Метод `split` можно вызывать с дополнительным (необязательным) аргументом, называемым разделителем (delimiter), который определяет символы, используемые как границы слов. В приведенном ниже примере в качестве разделителя используется дефис:


```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

Действие метода `join` обратно действию метода `split`. Метод `join` принимает список строк и объединяет его элементы. Поскольку `join` – метод строки, вы должны вызывать его от имени разделителя, а список слов передавать как параметр:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

В этом примере разделителем является символ пробела, поэтому join вставляет между словами пробелы. Для объединения строк без пробелов можно использовать пустую строку "" как разделитель.

8.10. Синтаксический анализ (парсинг) строк

Обычно при чтении файла необходимо что-то делать с его строками, а не просто выводить на экран всю строку без изменений. Часто требуется находить «интересующие нас строки», затем выполнять синтаксический анализ (парсинг – parsing) каждой такой строки, чтобы найти ее часть, наиболее интересную для нас. Что, если необходимо выводить день недели из тех строк, которые начинаются с префикса «From»?

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Метод split весьма эффективен при решении задач такого типа. Можно написать небольшую программу, которая ищет строки, начинающиеся с префикса «From», разделяет их на слова (split), затем выводит третье слово в строке (т. е. второй элемент полученного списка):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Исходный код: http://www.py4e.com/code3/search5.py
```

Вывод этой программы показан ниже:

```
Sat
Fri
Fri
Fri
...
```

Немного позже мы изучим более изощренные методики отбора строк для обработки, а также узнаем, как отделить такие строки для извлечения в точности того фрагмента информации, который мы ищем.

8.11. Объекты и значения

Если выполняются следующие инструкции присваивания:

```
a = 'banana'
b = 'banana'
```

то нам известно, что обе переменные a и b ссылаются на строку, но мы не знаем, ссылаются ли они на одну и ту же строку. Возможны два состояния, показанных на рис. 8.1.

Рис. 8.1 ❖ Переменные и объекты

В первом случае a и b ссылаются на два различных объекта, имеющих одинаковое значение. Во втором случае обе переменные ссылаются на один и тот же объект.

Чтобы проверить, ссылаются ли две переменные на один и тот же объект, можно воспользоваться оператором is.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

В этом примере Python создал только один строковый объект, и обе переменные a и b ссылаются на него.

Но если вы создаете два списка, то получите два различных объекта:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

В этом случае мы должны сказать, что эти два списка равнозначны (equivalent), потому что содержат одинаковые элементы, но не идентичны (identical), потому что не являются одним и тем же объектом. Если два объекта идентичны, то они также равнозначны, но если они равнозначны, то не обязательно идентичны.

До настоящего момента мы использовали термины «объект» и «значение» как взаимозаменяемые, но более точно и правильно говорить, что объект имеет значение. Если выполнить присваивание a = [1, 2, 3], то a ссылается на объект списка, значением которого является конкретная последовательность элементов. Если другой список содержит те же самые элементы, то мы должны говорить, что он имеет то же самое значение.

8.12. Псевдонимы

Если переменная a ссылается на некоторый объект и выполняется присваивание b = a, то обе переменные ссылаются на один и тот же объект:

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

```
>>> b is a
True
```

Связывание переменной с объектом называется ссылкой (reference). В приведенном выше примере существуют две ссылки на один и тот же объект.

Объект с более чем одной ссылкой имеет более одного имени, поэтому мы говорим, что у такого объекта есть псевдоним (alias)[1].

Если объект с псевдонимами является изменяемым, то изменения, сделанные с помощью одного псевдонима, воздействуют и на все другие:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Такое поведение может оказаться полезным, но оно также является потенциальным источником ошибок. В общем случае безопаснее избегать использования псевдонимов, когда вы работаете с изменяемыми объектами.

Для неизменяемых объектов, таких как строки, создание псевдонимов не вызывает особых проблем. В первом примере из предыдущего раздела:

```
a = 'banana'
b = 'banana'
```

почти никогда не имеет значения, ссылаются переменные `a` и `b` на одну строку или нет.

8.13. Списки как аргументы

Если вы передаете список в функцию, то функция получает ссылку на список. Если функция изменяет параметр-список, то вызывающая сторона видит эти изменения. Например, функция `delete_head` удаляет первый элемент из списка:

```
def delete_head(t):
    del t[0]
```

Ниже показано, как используется эта функция:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

Здесь параметр `t` и переменная `letters` – псевдонимы для одного и того же объекта.

Важно понимать различие между операциями, изменяющими списки, и операциями, которые создают новые списки. Например, метод `append` изменяет список, но оператор `+` создает новый список:

---

[1] Программисты весьма часто используют термин-кальку «алиас». – *Прим. перев.*

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

Это различие важно, когда вы пишете функции, которые предположительно будут изменять списки. Например, приведенная ниже функция в действительности не удаляет голову (первый элемент) списка:

```
def bad_delete_head(t):
    t = t[1:]              # Это неправильно!
```

Оператор вырезки создает новый список, а присваивание заставляет переменную `t` ссылаться на этот новый список, но это не оказывает никакого воздействия на список, который был передан как аргумент.

Более правильно написать функцию, создающую и возвращающую новый список. Например, функция `tail` возвращает все, кроме первого элемента переданного ей списка:

```
def tail(t):
    return t[1:]
```

Эта функция оставляет исходный список неизмененным. Ниже показано, как она используется:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

УПРАЖНЕНИЕ 1 Написать функцию `chop`, которая принимает список, изменяет его, удаляя первый и последний элементы, и возвращает `None`. Затем написать функцию `middle`, которая принимает список и возвращает новый список, содержащий все элементы, кроме первого и последнего.

8.14. Отладка

Неаккуратное использование списков (и других изменяемых объектов) может привести ко многим часам отладки. Ниже описаны наиболее часто встречающиеся проблемы и способы, позволяющие их избежать.

1. Не забывайте, что большинство методов списков изменяют аргумент и возвращают `None`. Это полная противоположность методам строк,

которые возвращают новую строку, а исходную строку оставляют неизменной.
Если для работы со строкой используется следующий код:

```
word = word.strip()
```

то возникает соблазн написать такой же код и для работы со списком:

```
t = t.sort()          # Это неправильно!
```

Поскольку метод `sort` возвращает `None`, следующая операция, которую вы попытаетесь выполнить с переменной `t`, вероятнее всего, приведет к ошибке.
Перед использованием методов и операторов списков вы должны внимательно изучить документацию, а затем протестировать их в интерактивном режиме. Документация для методов и операторов, которые списки совместно используют вместе с другими последовательностями, находится здесь:

docs.python.org/library/stdtypes.html#common-sequence-operations.

Документация для методов и операторов, которые применяются только для изменяемых последовательностей, расположена здесь:

docs.python.org/library/stdtypes.html#mutable-sequence-types.

2. Выберите одно идиоматическое выражение языка для использования и всегда применяйте его.
Частично проблема при использовании списков заключается в том, что существует слишком много способов выполнения действий. Например, для удаления элемента из списка можно применять `pop`, `remove`, `del` или даже присваивание вырезки.

Для добавления элемента можно воспользоваться методом `append` или оператором +, но при этом не следует забывать, что следующие операции являются корректными:

```
t.append(x)
t = t + [x]
```

а эти операции некорректны:

```
t.append([x])         # Это неправильно!
t = t.append(x)       # Это неправильно!
t + [x]               # Это неправильно!
t = t + x             # Это неправильно!
```

Попробуйте выполнить все эти примеры в интерактивном режиме, чтобы убедиться в том, что вы понимаете их работу и полученные результаты. Следует отметить, что только самый последний пример приводит к ошибке времени выполнения, первые три допустимы, но делают неправильные вещи.

3. Создавайте копии, чтобы избегать псевдонимов.
   Если необходимо использовать метод, подобный `sort`, который изменяет аргумент, но при этом требуется также сохранить исходный список, то можно создать его копию.

   ```
   orig = t[:]
   t.sort()
   ```

   В этом примере также можно использовать встроенную функцию `sorted`, которая возвращает новый отсортированный список, а исходный оставляет неизмененным. Но в данном случае вы должны избегать использования `sorted` в качестве имени переменной.
4. Списки, метод `split` и файлы.
   При чтении и синтаксическом анализе файлов существует множество возможностей встретить ввод, который может аварийно завершить программу, поэтому полезно снова обратиться к шаблону-защитнику при написании программ, читающих все содержимое файла подряд в поисках «иголки в стоге сена».
   Вернемся к нашей программе, которая ищет день недели в строках заданного файла:

   ```
   From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
   ```

   Поскольку мы разделили эту строку на отдельные слова, можно было бы обойтись без метода `startwith` и просто выполнить поиск по первому слову, чтобы определить, интересует ли нас данная строка. Можно использовать инструкцию `continue` для пропуска строк без первого слова «From», как показано ниже:

   ```
   fhand = open('mbox-short.txt')
   for line in fhand:
       words = line.split()
       if words[0] != 'From' : continue
       print(words[2])
   ```

   Это выглядит намного проще, и нам даже не нужен метод `rstrip` для удаления символов перехода на новую строку в конце каждой текстовой строки. Но лучше ли эта версия на самом деле?

   ```
   python search8.py
   Sat
   Traceback (most recent call last):
     File "search8.py", line 5, in <module>
       if words[0] != 'From' : continue
   IndexError: list index out of range
   ```

   Программа вроде бы работает, и мы видим день недели (`Sat`), найденный в первой строке, но затем программа «падает» с обратной трассировкой ошибки. Что пошло не так? Какие некорректные данные

заставили нашу изящную, умную и в высшей степени питоническую программу завершиться с ошибкой?

Вы можете пристально разглядывать эту программу в течение длительного времени и ломать голову над данной проблемой или попросить кого-нибудь помочь разобраться, но быстрее и разумнее просто добавить инструкцию `print`. Самое подходящее место для вставки `print` – прямо перед строкой, где программа «падает», и здесь нужно вывести данные, которые предположительно привели к критической ошибке.

Теперь этот подход может сгенерировать огромное количество строк вывода, но, по крайней мере, вы сразу же получите представление о возникшей проблеме. Поэтому мы добавляем инструкцию вывода значения переменной `words` прямо перед пятой строкой. Мы даже добавили префикс «Debug: « («Отладка: «) в строку вывода, чтобы отделить обычный вывод от отладочной информации.

```
for line in fhand:
    words = line.split()
    print('Debug: ', words)
    if words[0] != 'From' : continue
    print(words[2])
```

После запуска этой программы по экрану прокручивается огромное количество строк, но в конце мы видим подготовленный нами отладочный вывод и обратную трассировку и можем узнать, что произошло прямо перед началом обратной трассировки.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Каждая отладочная строка выводит список слов, полученных в результате разделения (`split`) текстовой строки из файла на отдельные слова. Когда программа терпит крах, этот список слов пуст `[]`. Если открыть этот файл в текстовом редакторе и внимательно посмотреть на его содержимое, то можно заметить, что в этом месте оно выглядит следующим образом:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan 5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Ошибка возникла, когда программа встретила пустую строку. Разумеется, в пустой строке «ноль слов». Почему мы не подумали об этом, когда писали исходный код? Когда в коде выполняется поиск первого слова (`word[0]`) для проверки его совпадения с образцом «From», возникает ошибка «индекс вне диапазона».

Разумеется, это самое подходящее место для добавления некоторого защитного кода, чтобы избежать проверки первого слова, если его здесь нет. Существует много способов защиты подобного кода. Мы выбираем проверку количества слов, прежде чем начать сравнение первого слова:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

Сначала мы закомментировали отладочную инструкцию `print` вместо ее удаления на тот случай, если внесенное изменение окажется ошибочным и снова потребуется отладка. Затем добавили защитную инструкцию, проверяющую, не является ли список пустым, и если в списке ноль слов, то используется `continue` для пропуска текущей строки и перехода к следующей в просматриваемом файле.

Две инструкции `continue` можно считать способствующими очистке набора строк, которые «нас интересуют» и которые необходимо каким-то образом дополнительно обработать. Строка без слов для нас «неинтересна», поэтому мы немедленно переходим к следующей строке. Строка, не содержащая первого слова «From», также нас не интересует, поэтому мы ее пропускаем.

Программа, измененная, как показано выше, работает успешно, поэтому, возможно, она корректна. Защитная инструкция дает уверенность в том, что на `words[0]` никогда не возникает ошибка, но, возможно, этого недостаточно. Когда мы пишем программы, то всегда должны думать: «Что может пойти не так?»

УПРАЖНЕНИЕ 2 Выясните, какая строка в приведенной выше программе остается недостаточно защищенной. Подумайте, можно ли сформировать текстовый файл, который заставит программу завершиться с ошибкой, затем измените программу так, чтобы эта строка кода была надежно защищена, потом протестируйте программу, чтобы убедиться в том, что она корректно обрабатывает новый текстовый файл.

УПРАЖНЕНИЕ 3 Перепишите защитный код в показанном выше примере без двух инструкций `if`. Вместо них используйте составное логическое выражение с применением логического оператора `or` в одной инструкции `if`.

8.15. Словарь терминов

- Вложенный список (nested list) – список, который является элементом другого списка.
- Идентичный (identical) – представляющий один и тот же объект (при этом предполагается равнозначность).
- Индекс (index) – целочисленное значение, определяющее позицию элемента в списке.
- Объект (object) – сущность, на которую может ссылаться переменная. Объект имеет тип и значение.
- Проход по списку (list traversal) – последовательный доступ к каждому элементу списка.
- Псевдоним (alias) – имена двух и более переменных, ссылающихся на один и тот же объект.
- Равнозначный (equivalent) – имеющий одинаковое значение.
- Разделитель (delimiter) – символ или строка, определяющая места разделения строки на отдельные слова.
- Список (list) – последовательность значений.
- Ссылка (reference) – связь между переменной и ее значением.
- Элемент (element) – одно из значений в списке (или в другой последовательности). Также используется термин item.

8.16. Упражнения

УПРАЖНЕНИЕ 4 Найти все неповторяющиеся слова в файле.

Шекспир использовал в своих произведениях более 20 000 слов. Но как вы могли бы определить это? Как составить список всех слов, использованных Шекспиром? Нужно ли скачивать все его произведения, читать их и помечать все неповторяющиеся слова вручную?

Вместо этой рутины воспользуемся средствами Python. Составим список всех неповторяющихся слов, отсортируем их в алфавитном порядке. Исходные данные хранятся в файле *romeo.txt*, содержащем некоторое подмножество произведений Шекспира.

Чтобы начать работу, скачайте копию исходного файла здесь:

www.py4e.com/code3/romeo.txt.

Создайте список неповторяющихся слов, который будет содержать итоговый результат. Напишите программу, которая открывает файл *romeo.txt* и читает его содержимое строка за строкой. Каждая строка файла разделяется на список слов с помощью метода `split`. Для каждого слова в этом списке выполняется проверка: не содержится ли оно в текущем списке неповторяющихся слов. Если проверяемого слова нет в этом списке, то оно добавляется в него. Перед завершением программы выполняется сортировка и выводится список неповторяющихся слов в алфавитном порядке.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

УПРАЖНЕНИЕ 5 Минималистичный клиент электронной почты.

MBOX (mail box) – это широко распространенный формат файла для хранения и совместного использования набора сообщений (писем) электронной почты. Этот формат использовался на самых первых серверах электронной почты и в приложениях для настольных компьютеров (десктопов). Если не слишком углубляться в подробности, то MBOX – это текстовый файл, в котором в последовательном порядке хранятся сообщения электронной почты. Сообщения разделяются особой строкой, начинающейся с префикса `From` (обратите внимание на пробел после префикса). Важно, что строки, начинающиеся с префикса `From:` (обратите внимание на двоеточие после префикса), описывают само сообщение и не действуют как разделитель. Предположим, что вы написали минималистичное приложение электронной почты, которое составляет список сообщений от конкретных отправителей в почтовом ящике Inbox пользователя и подсчитывает количество этих сообщений.

Написать программу для чтения данных в файле электронной почты (mail box), и когда обнаруживается строка, начинающаяся с префикса «From», ее необходимо разделить на отдельные слова с помощью метода `split`. Нас интересует, кто отправил сообщение, – это второе слово в строке с префиксом «From».

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Вы будете выполнять синтаксический анализ (парсинг) строки с префиксом «From» и выводить второе слово из каждой From-строки, затем также подсчитаете число строк From (но не From:) и в конце выведете значение счетчика. Ниже показан корректный пример вывода, в котором некоторые строки удалены (для экономии места):

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...некоторые выводимые строки удалены...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

УПРАЖНЕНИЕ 6 Переписать программу, которая предлагает пользователю ввести список чисел и выводит максимальное и минимальное числа после

того, как пользователь введет слово «done», завершающее ввод. Написать программу для сохранения в списке введенных пользователем чисел и использовать функции `max()` и `min()` для определения максимального и минимального чисел после завершения цикла (ввода).

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

Глава 9

Словари

Словарь (dictionary) похож на список, но обладает более обобщенными свойствами. В списке индексы позиций должны быть целыми числами, в словаре индексы могут иметь (почти) любой тип.

Можно считать словарь отображением между множеством индексов (которые называются ключами (keys)) и множеством значений. Каждый ключ отображается в значение. Связь ключа и значения называется парой ключ-значение (key-value pair) или иногда записью (item).

В качестве примера создадим словарь, отображающий слова английского языка в слова испанского, поэтому ключи и значения будут представлены строками.

Функция `dict` создает новый словарь без записей. Поскольку `dict` является именем встроенной функции, следует избегать его использования в качестве имени переменной.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

Фигурные скобки {} представляют пустой словарь. Для добавления записей в словарь можно использовать квадратные скобки:

```
>>> eng2sp['one'] = 'uno'
```

Эта строка создает элемент, отображающий ключ `'one'` в значение `'uno'`. Если снова вывести содержимое словаря, то мы увидим пару ключ-значение с символом двоеточия, разделяющим ключ и значение:

```
>>> print(eng2sp)
{'one': 'uno'}
```

Этот формат вывода также является форматом ввода. Например, можно создать новый словарь с тремя элементами. Но если вывести содержимое `eng2sp`, то, возможно, вы будете удивлены:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Порядок пар ключ-значение не совпадает с порядком их ввода. В действительности, если вы введете этот пример на своем компьютере, тогда, возможно, получите другой результат. В общем случае порядок элементов в словаре непредсказуем.

Но это не является проблемой, так как элементы словаря никогда не индексируются целыми числами. Вместо этого используются ключи для поиска соответствующих значений:

```
>>> print(eng2sp['two'])
'dos'
```

Ключ `'two'` всегда отображается в значение `'dos'`, поэтому порядок элементов значения не имеет.

Если заданного ключа нет в словаре, то возникает исключение и выводится соответствующее сообщение:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

Функция `len` работает со словарями. Она возвращает число пар ключ-значение:

```
>>> len(eng2sp)
3
```

Оператор `in` также работает со словарями. Он сообщает, присутствует ли что-либо в словаре как ключ (присутствие как значения не рассматривается как корректное).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Чтобы увидеть наличие чего-либо в словаре как значения, можно воспользоваться методом `values`, который возвращает значения как тип, который можно преобразовать в список, а затем применить оператор `in`:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

Оператор `in` использует различные алгоритмы для списков и словарей. Для списков применяется алгоритм линейного поиска. По мере увеличения длины списка увеличивается и время поиска в прямой пропорции относительно длины. Для словарей Python использует алгоритм, который называется хеш-таблицей (hash table), обладающей замечательным свойством: для оператора `in` требуется почти одинаковое время независимо от того, сколько элементов содержится в словаре. Здесь я не буду объяснять, почему хеш-таблицы обладают таким волшебным свойством, но вы можете более подробно узнать об этом по адресу: wikipedia.org/wiki/Hash_table; https://ru.wikipedia.org/wiki/Хеш-таблица.

УПРАЖНЕНИЕ 1 Загрузить копию файла: www.py4e.com/code3/words.txt.

Написать программу, которая читает слова из файла *words.txt* и сохраняет их как ключи в словаре. Значения в данном случае не важны. Затем можно воспользоваться оператором `in` как быстрым способом, позволяющим проверить, содержится ли некоторая строка в этом словаре.

9.1. Словарь как множество счетчиков

Предположим, что задана некоторая строка и необходимо подсчитать, сколько раз встречается в ней каждая буква. Существует несколько возможных способов сделать это:

1) можно создать 26 переменных, по одной для каждой буквы английского алфавита. Затем выполнить проход по строке и для каждого найденного символа инкрементировать соответствующий счетчик, возможно, с использованием цепочных условных выражений;
2) можно создать список из 26 элементов. Затем преобразовать каждый символ в число (применив встроенную функцию `ord`), использовать полученные числа как индексы в списке и инкрементировать соответствующий счетчик;
3) можно создать словарь с символами как ключами и счетчиками как соответствующими значениями. Когда символ встречается в первый раз, он добавляется как элемент в словарь. После этого нужно просто инкрементировать значение существующего элемента.

Каждый из предложенных вариантов выполняет одно и то же вычисление, но в каждой реализации – различными способами.

Реализация (implementation) – это способ выполнения вычисления. Некоторые реализации лучше других. Например, преимуществом реализации с применением словаря является отсутствие необходимости знать заранее, какие буквы встречаются в строке, поэтому мы отводим место только для тех букв, которые действительно содержатся в строке.

Исходный код для решения этой задачи может выглядеть приблизительно так:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

Мы эффективно вычисляем гистограмму (histogram) – это статистический термин, обозначающий набор счетчиков (или частот).

Цикл `for` проходит по строке. На каждой итерации цикла если символ `c` не находится в словаре, то создается новый элемент с ключом `c` и начальным

значением 1 (поскольку обнаружено одно вхождение этой буквы). Если c уже находится в словаре, то выполняется инкрементирование d[c].

Ниже показан вывод этой программы:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

Эта гистограмма позволяет узнать, что буквы «a» и «b» встретились по одному разу, буква «o» обнаружена дважды и т. д.

Для словарей существует метод get, который принимает ключ и значение по умолчанию. Если ключ существует в словаре, то get возвращает соответствующее значение, иначе возвращается значение по умолчанию. Например:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

Можно использовать get для записи нашей гистограммы в более компактном виде. Поскольку метод get автоматически обрабатывает случай, в котором ключа нет в словаре, можно исключить четыре строки кода и убрать инструкцию if.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

Использование метода get для упрощения этого цикла подсчета в итоге становится весьма широко используемой «идиомой» языка Python, и мы будем использовать этот прием многократно в остальной части книги. Но вы должны уделить немного времени для сравнения цикла с использованием инструкции if и оператора in с циклом, где применяется метод get. Оба варианта делают одно и то же, но один из них более лаконичный.

9.2. Словари и файлы

Одним из широко распространенных способов использования словаря является подсчет вхождений слов в файле с некоторым записанным текстом. Начнем с очень простого файла с текстом, взятым из пьесы Шекспира «Ромео и Джульетта».

Для первой группы примеров будем использовать сокращенную и упрощенную версию этого текста без знаков препинания. В дальнейшем мы будем работать с полным текстом пьесы с включением знаков препинания.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
```

```
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Мы напишем программу на языке Python для построчного чтения из этого файла, разделим каждую строку на список слов, затем организуем цикл прохода по каждому слову в текущей строке и подсчитаем экземпляр каждого слова, используя для этого словарь.

Вы увидите, что для решения данной задачи потребуется два цикла `for`. Внешний цикл читает строки из файла, а внутренний цикл выполняет итерации по каждому слову в текущей конкретной строке. Это пример шаблона под названием «вложенные циклы», потому что один из циклов является внешним (outer), а другой – внутренним (inner).

Поскольку внутренний цикл выполняет все свои итерации каждый раз, когда внешний цикл выполняет только одну итерацию, мы считаем, что внутренний цикл выполняется «быстрее», а итерации внешнего цикла более медленные.

Такое сочетание двух вложенных циклов гарантирует, что мы подсчитаем каждое слово в каждой строке входного файла.

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Исходный код: http://www.py4e.com/code3/count1.py
```

В инструкции `else` используется более компактная альтернативная форма записи инкрементирования переменной. Выражение `counts[word] += 1` равнозначно выражению `counts[word] = counts[word] + 1`. Тот же способ можно использовать для изменения значения переменной на любое требуемое число. Аналогичные компактные формы записи существуют и для операций `-=`, `*=` и `/=`.

После запуска этой программы мы увидим неотформатированный вывод всех счетчиков слов в произвольном (неотсортированном) порядке хеш-таблицы. (Файл *romeo.txt* доступен здесь: www.py4e.com/code3/romeo.txt.)

```
python count1.py
Enter the file name: romeo.txt
```

```
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

Немного неудобно просматривать такой словарь в поисках наиболее часто встречающихся слов и их счетчиков, поэтому необходимо добавить некоторый код Python, чтобы получить более удобный формат вывода.

9.3. Циклы и словари

Если словарь используется как последовательность в инструкции цикла `for`, то выполняется проход по ключам словаря. Показанный ниже цикл выводит каждый ключ и соответствующее ему значение:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

Вот что выводит этот цикл:

```
jan 100
chuck 1
annie 42
```

И в этом случае ключи не располагаются в каком-то определенном порядке.

Можно воспользоваться этим шаблоном для реализации разнообразных идиом цикла, которые мы рассмотрели ранее. Например, если необходим поиск всех вхождений слов в словаре со значением, большим десяти, то можно было бы написать такой код:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```


Цикл `for` выполняет итерации по ключам словаря, поэтому мы обязательно должны использовать оператор индекса для извлечения соответствующего значения для каждого ключа. Вывод этой версии выглядит так:

```
jan 100
annie 42
```

Здесь мы видим только записи, значение которых больше 10.

Если требуется вывод ключей в алфавитном порядке, то сначала нужно создать список ключей в словаре, воспользовавшись методом `keys`, доступным для объектов словарей, затем отсортировать этот список и пройти в цик-

ле по отсортированному списку, рассматривая каждый ключ и выводя пары ключ-значение в отсортированном порядке, как показано ниже:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

Теперь вывод выглядит следующим образом:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

Сначала мы видим список ключей в неотсортированном порядке, полученный с помощью метода `keys`. Затем в цикле `for` выводятся пары ключ-значение в алфавитном порядке.

9.4. Расширенный синтаксический анализ текста

В приведенном выше примере с использованием файла *romeo.txt* мы сделали файл как можно более простым, удалив вручную все знаки пунктуации. В настоящем тексте много знаков пунктуации, как показано ниже.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Поскольку в Python функция `split` ищет пробелы и воспринимает слова как элементы, разделенные пробелами, мы должны считать «soft!» и «soft» различными словами и создать отдельные словарные записи для каждого слова.

Кроме того, так как в файле используются заглавные буквы, необходимо считать «who» и «Who» различными словами с отдельными счетчиками.

Обе эти проблемы можно решить, воспользовавшись методами строки `lower`, `punctuation` и `translate`. Метод `translate` наиболее изощренный из трех перечисленных. Ниже приведена краткая документация для этого метода:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

Заменяет символы в строке `fromstr` на символ в той же позиции в строке `tostr` и удаляет все символы, заданные в строке `deletestr`. Строки `fromstr` и `tostr` могут быть пустыми, а параметр `deletestr` можно пропустить.

Мы не будем определять строку tostr, но воспользуемся параметром deletestr для удаления всех знаков пунктуации. Мы даже позволим Python сообщить нам список символов, которые он считает «знаками пунктуации»:

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

Параметры, используемые методом translate, были другими в версии Python 2.0.

Вносим в нашу программу изменения, как показано ниже:

```
import string

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Исходный код: http://www.py4e.com/code3/count2.py
```

В части процесса обучения «Art of Python» или «Thinking Pythonically» утверждается, что Python часто имеет встроенные возможности для решения многих широко распространенных задач анализа данных. Со временем вы увидите достаточное количество примеров кода и прочитаете достаточный объем документации, чтобы узнать, где искать сведения о том, что кто-то уже написал исходный код, который сделает вашу работу намного проще.

Ниже приведена сокращенная версия вывода приведенной выше программы:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Просмотр этого вывода в полном объеме остается неудобным, и мы можем использовать Python, чтобы получить в точности то, что ищем, но для этого необходимо более подробно узнать о кортежах языка Python. После изучения кортежей мы вернемся к этому примеру.

9.5. Отладка

Когда вы работаете с более крупными наборами данных, может стать неудобной отладка при помощи вывода и проверки данных вручную. Ниже описаны некоторые предложения по отладке крупных наборов данных.

- **Уменьшение объема входных данных.** Если возможно, сократите размер набора данных. Например, если программа считывает текстовый файл, то начните только с первых 10 строк или даже с меньшего образца, который сможете найти. Можно отредактировать сами файлы или (лучше) изменить программу так, чтобы она считывала только первые `n` строк.
- **Проверка сводной информации и типов.** Вместо вывода и проверки всего набора данных в целом рассмотрите вариант вывода сводных характеристик данных: например, число элементов в словаре или общую сумму списка чисел.
  Наиболее частой причиной ошибок времени выполнения является значение неправильного типа. Для отладки этой разновидности ошибок зачастую достаточно вывести тип значения.
- **Использование кода самодиагностики.** Иногда можно написать код для автоматической проверки на ошибки. Например, если вычисляется среднее арифметическое списка чисел, то можно проверить, что результат не больше значения наибольшего и не меньше значения наименьшего элемента списка. Такой подход называется «проверкой корректности (с точки зрения здравого смысла)» (sanity check), потому что он позволяет обнаруживать результаты, которые «абсолютно нелогичны».
  Еще одна методика проверки сравнивает результаты двух различных вычислений, чтобы определить их согласованность. Этот подход называется «проверкой на согласованность» (consistency check).
- **Правильно отформатированный вывод.** Форматирование отладочного вывода может существенно облегчить поиск ошибок.

И напомню еще раз: время, затраченное на создание вспомогательных средств, может сократить время, потраченное на отладку.

9.6. Словарь терминов

- Вложенный цикл (nested loop) – когда один или несколько циклов находятся внутри другого цикла, это называется вложенными циклами.

Внутренний цикл выполняется полностью на каждой итерации внешнего цикла.

- Гистограмма (histogram) – набор (множество) счетчиков.
- Запись (item) – другое название пары ключ-значение.
- Значение (value) – объект в словаре, представляющий вторую часть пары ключ-значение. Это более специализированное определение по сравнению с ранее использованным термином «значение».
- Ключ (key) – объект в словаре, представляющий первую часть пары ключ-значение.
- Пара ключ-значение (key-value pair) – представление отображения ключа в значение.
- Поиск (lookup) – операция словаря, которая берет ключ и находит соответствующее значение.
- Реализация (implementation) – способ выполнения вычислений.
- Словарь (dictionary) – отображение набора ключей в соответствующие им значения.
- Хеш-таблица (hashtable) – алгоритм, используемый для реализации в Python словарей.
- Хеш-функция (hash function) – функция, используемая хеш-таблицей для вычисления позиции ключа.

9.7. Упражнения

УПРАЖНЕНИЕ 2 Написать программу, которая выполняет категоризацию сообщений электронной почты по дням недели, в которые были получены эти сообщения. Для этого необходимо рассмотреть строки, начинающиеся с «From», затем найти третье слово и обработать текущий счетчик каждого дня недели. В конце работы программа выводит содержимое словаря (порядок не важен).

Пример строки:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Пример выполнения:

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

УПРАЖНЕНИЕ 3 Написать программу для чтения журнала электронной почты и построения гистограммы с использованием словаря для подсчета сообщений, пришедших с каждого адреса email, и вывода итогового словаря.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
```

```
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

УПРАЖНЕНИЕ 4 Добавить в программу из упражнения 3 код, определяющий, от кого пришло наибольшее число сообщений, хранящихся в файле. После того как все данные прочитаны и создан словарь, пройти по словарю, используя цикл определения максимального значения (см. главу 5, раздел 5.6.2 «Циклы вычисления максимума и минимума») для поиска лица, отправившего наибольшее число сообщений, и вывести это число.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

УПРАЖНЕНИЕ 5 Эта программа фиксирует имя (почтового) домена (вместо полного адреса), из которого было отправлено сообщение, а не имя лица, от которого пришло сообщение (т. е. полный адрес email). В конце работы программа выводит содержимое созданного словаря.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

Глава 10

Кортежи

10.1. Кортежи неизменяемы

Кортеж (tuple)[1] – это последовательность значений, во многом похожая на список. Значения, хранящиеся в кортеже, могут иметь любой тип и индексируются целыми числами. Важное отличие состоит в том, что кортежи неизменяемы. Кортежи можно также сравнивать и хешировать, поэтому мы имеем возможность сортировать списки кортежей и использовать кортежи как ключи в словарях Python.

Синтаксически кортеж представляет собой список значений, разделенных запятыми:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Достаточно часто кортежи заключают в круглые скобки, хотя это не является обязательным требованием, скорее для того, чтобы проще было идентифицировать кортежи в исходном коде Python:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Для создания кортежа с одним элементом необходимо обязательно ввести завершающую запятую:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Без указания запятой Python воспринимает `('a')` как выражение со строкой в круглых скобках, результатом вычисления которого становится обычная строка:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

---

[1] Любопытный факт: слово «кортеж» (tuple) происходит от названий, которые присваивались последовательностям чисел различной длины: двойной (double), тройной (triple), четверной (quadruple), пятерной (quintuple), шестерной (sextuple), семерной (septuple) и т. д.

Другим способом создания кортежа является использование встроенной функции tuple. Если не задан аргумент, то создается пустой кортеж:

```
>>> t = tuple()
>>> print(t)
()
```

Если аргументом является последовательность (строка, список или кортеж), то результатом вызова tuple становится кортеж с элементами заданной последовательности:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Поскольку tuple – имя функции-конструктора, вы должны избегать его использования в качестве имени переменной.

Большинство операторов списка также работают и с кортежами. Оператор квадратные скобки индексирует элемент:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

Оператор вырезки выбирает заданный диапазон элементов.

```
>>> print(t[1:3])
('b', 'c')
```

Но если попытаться изменить какой-либо элемент кортежа, то возникает ошибка:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment (объект не поддерживает присваивание
элементов)
```

Нельзя изменять элементы кортежа, но можно заменить один кортеж на другой:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

10.2. Сравнение кортежей

Операторы сравнения работают с кортежами и другими последовательностями. Python начинает сравнение с первого элемента каждой последовательности. Если они равны, то выполняется сравнение следующих элементов и т. д. до тех пор, пока не будут найдены различающиеся элементы. Дальнейшие элементы не рассматриваются (даже если они действительно важны).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Функция `sort` работает аналогичным образом. Она выполняет сортировку прежде всего по первому элементу, но в случае равенства сортирует по второму элементу и т. д.

Это свойство естественным образом приводит к шаблону, называемому DSU (преобразование Шварца (Randal L. Schwartz)):

- decorate – формирование последовательности посредством создания списка кортежей с одним или несколькими ключами сортировки, записанными перед элементами из этой последовательности;
- sort – сортировка кортежей с использованием встроенной в Python функции `sort`;
- undecorate – извлечение отсортированных элементов из последовательности.

Например, предположим, что имеется список слов, и необходимо отсортировать их в порядке от самого длинного к самому короткому:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print(res)

# Исходный код: http://www.py4e.com/code3/soft.py
```

Первый цикл создает список кортежей, в котором каждый кортеж – это слово, перед которым записана его длина.

Функция `sort` сравнивает первый элемент – длину, а второй элемент рассматривает только при равенстве первого. Ключевое слово аргумент `reverse=True` сообщает функции `sort` о необходимости сортировки в убывающем порядке.

Второй цикл проходит по списку кортежей и создает список слов в убывающем порядке по длине. Слова из четырех букв сортируются в обратном алфавитном порядке, поэтому «what» располагается раньше, чем «soft» в создаваемом списке.

Вывод этой программы показан ниже:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Разумеется, строка утрачивает свое поэтическое воздействие после превращения в список Python и сортировки в обратном порядке по длине слова.

10.3. Присваивание кортежам

Одним из замечательных синтаксических свойств, присущих языку Python, является возможность записи кортежа в левой части инструкции присваивания. Это позволяет одновременно присваивать значения более чем одной переменной, когда левая часть инструкции присваивания является последовательностью.

В приведенном ниже примере имеется список с двумя элементами (который является последовательностью), и выполняется присваивание первого и второго элементов этой последовательности переменным `x` и `y` в одной инструкции.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

Это вовсе не магия, просто Python, условно говоря, выполняет преобразование синтаксиса присваивания кортежу в следующую форму[1]:

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

Стилистически при использовании кортежа в левой части инструкции присваивания мы не записываем круглые скобки, но ниже показан равнозначный допустимый синтаксис:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```


[1] Python не преобразовывает синтаксис буквально. Например, если вы попытаетесь сделать нечто подобное со словарем, то такой способ не будет работать так, как вы, вероятно, ожидаете.

Особенно хитроумное применение присваивания кортежу позволяет поменять местами значения двух переменных в одной инструкции:

```
>>> a, b = b, a
```

В обеих частях этой инструкции находятся кортежи, но в левой части – кортеж переменных, а в правой части – кортеж выражений. Каждое значение из правой части присваивается соответствующей переменной в левой части. Все выражения в правой части вычисляются до какого-либо присваивания.

Число переменных в левой и правой частях инструкции присваивания обязательно должно быть одинаковым:


```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

В более общем смысле правая часть может быть любым типом последовательности (строкой, списком или кортежем). Например, для разделения адреса электронной почты на имя пользователя и имя домена можно написать:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Возвращаемым значением из метода `split` является список с двумя элементами: первый присваивается переменной `uname`, второй – переменной `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

10.4. Словари и кортежи

Для словарей существует метод `items`, который возвращает список кортежей, где каждый кортеж является парой ключ-значение:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

Как и следовало ожидать при работе со словарем, его записи не располагаются в каком-либо определенном порядке.

Но поскольку последовательность кортежей является списком, а кортежи можно сравнивать, теперь появляется возможность сортировки списка кортежей. Преобразование словаря в список кортежей – это удобный для нас способ вывода содержимого словаря, отсортированного по ключу:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

Новый список отсортирован в возрастающем алфавитном порядке по ключу.

10.5. Множественное присваивание с помощью словарей

Объединив метод `items`, присваивание кортежей и цикл `for`, можно наблюдать замечательный шаблон кода для прохода по ключам и значениям словаря в одном цикле:

```
for key, val in list(d.items()):
    print(val, key)
```

Этот цикл содержит две итерационные переменные, потому что метод `items` возвращает список кортежей, а выражение `key, val` – это присваивание кортежа, позволяющее выполнять последовательные итерации по каждой паре ключ-значение в словаре.

При каждой итерации в этом цикле обе переменные `key` и `value` перемещаются к следующей паре ключ-значение в словаре (с поддержанием порядка хеширования).

Вывод этого цикла показан ниже:

```
10 a
22 c
1 b
```

Еще раз: это порядок хеширования ключа (т. е. какого-либо определенного порядка нет).

Если объединить эти две методики, то можно вывести содержимое словаря, отсортированное по значению, хранящемуся в паре ключ-значение.

Для этого сначала создадим список кортежей, где каждый кортеж имеет вид `(value, key)`. Метод `items` должен предоставить нам список кортежей `(key, value)`, но на этот раз требуется сортировка по значению, а не по ключу. Поскольку мы уже создали список с кортежами значение-ключ, не составляет никакого труда отсортировать этот список в обратном порядке и вывести новый отсортированный список.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
```

```
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

При аккуратном формировании списка кортежей с целью получить значение как первый элемент каждого кортежа мы можем отсортировать этот список кортежей и получить содержимое словаря, отсортированное по значению.

10.6. Наиболее часто встречающиеся слова

Вернемся к нашему работающему примеру обработки текста из пьесы Шекспира «Ромео и Джульетта», акт 2, сцена 2. Можно улучшить нашу программу, используя описанную в предыдущем разделе методику, чтобы вывести десять наиболее часто встречающихся слов в этом тексте, как показано ниже:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(str.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Сортировка словаря по значению
lst = list()
for key, val in list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]:
    print(key, val)

# Исходный код: http://www.py4e.com/code3/count3.py
```

Первая часть программы, в которой считывается текст из файла и вычисляется словарь, отображающий каждое слово в соответствующий счетчик его вхождения в текстовый документ, не изменилась. Но вместо простого вывода счетчиков `counts` и завершения программы здесь мы создаем список кортежей (`val`, `key`), а затем сортируем этот список в обратном порядке.

Поскольку значение является первым элементом кортежа, оно будет использоваться в операциях сравнения. Если существует более одного кортежа с одинаковым значением, то будет рассматриваться второй элемент (ключ), следовательно, кортежи с одинаковыми значениями дополнительно сортируются по ключу в алфавитном порядке.

В конце программы записан простой цикл `for`, который выполняет множественное присваивание на каждой итерации и выводит десять наиболее часто встречающихся в тексте слов, проходя по вырезке из итогового списка (`lst[:10]`).

Теперь окончательный вывод выглядит именно так, как требуется для анализа частоты слов в тексте.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

Тот факт, что этот сложный синтаксический анализ данных можно выполнить с помощью абсолютно понятной программы на языке Python, насчитывающей всего лишь 19 строк, – одна из причин, по которой Python является правильным вариантом выбора в качестве языка для обработки информации.

10.7. Использование кортежей как ключей в словарях

Поскольку кортежи можно хешировать, а списки нет, при необходимости создать составной ключ (composite key) в словаре мы непременно должны использовать кортеж как такой ключ.

Составной ключ может встретиться, если требуется создать телефонный справочник, отображающий пары (фамилия, имя) в номера телефонов. Предположим, что уже определены переменные `last`, `first` и `number`, тогда можно написать инструкцию присваивания для словаря следующим образом:

```
directory[last,first] = number
```

Выражение в квадратных скобках – это кортеж. Можно воспользоваться присваиванием кортежа в цикле `for` для прохода по этому словарю.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

Этот цикл выполняет проход по ключам в словаре `directory`, которые представлены кортежами. Здесь элементы каждого кортежа присваиваются переменным `last` и `first`, затем выводится имя, фамилия и соответствующий номер телефона.

10.8. Последовательности: строки, списки и кортежи – ну и ну!

Я сосредоточил все внимание на списках кортежей, но почти все примеры в этой главе также работают со списками списков, кортежами кортежей и кортежами списков. Чтобы избежать перечисления всех возможных сочетаний, иногда проще сказать о последовательностях последовательностей.

Во многих контекстах различные типы последовательностей (строки, списки и кортежи) могут использоваться как взаимозаменяемые. В таком случае как и почему мы выбираем один из этих типов?

Начнем с очевидного факта: строки более ограничены, чем другие последовательности, потому что их элементами непременно должны быть символы. Кроме того, строки неизменяемы. Если требуется возможность изменения символов в строке (но не создание новой строки), то, вероятно, вместо строки следует использовать список символов.

Списки используются более широко, чем кортежи, в основном потому, что являются изменяемыми. Но существует несколько случаев, в которых, возможно, более предпочтительны кортежи:

1) в некоторых контекстах, например в инструкции `return`, синтаксически проще создать кортеж, чем список. В других контекстах список может оказаться более предпочтительным;
2) если требуется использование последовательности как ключа в словаре, то рекомендуется воспользоваться неизменяемым типом, таким как кортеж или строка;
3) если последовательность передается как аргумент в функцию, то использование кортежа снижает вероятность непредсказуемого поведения из-за наличия псевдонимов.

Поскольку кортежи являются неизменяемыми, для них не предоставляются такие методы, как `sort` и `reverse`, которые изменяют существующие списки. Тем не менее Python предоставляет встроенные функции `sorted` и `reversed`, которые принимают любую последовательность как параметр и возвращают новую последовательность с теми же элементами, но в другом порядке.

10.9. Отладка

Списки, словари и кортежи в более обобщенном смысле известны как структуры данных (data structures). В этой главе мы начали рассматривать состав-

ные структуры данных, такие как списки кортежей и словари, содержащие кортежи как ключи и списки как значения. Составные структуры данных полезны, но они являются потенциальным источником ошибок, которые я называю ошибками формы (shape errors), т. е. ошибками, возникающими, когда структура данных имеет некорректный тип, размер или построение, или, возможно, вы написали некоторый код, забыли форму своих данных, и возникла ошибка. Например, если вы ожидаете список с одним целым числом, а я передаю вам просто обычное целое число (не в списке), то код работать не будет.

10.10. Словарь терминов

- DSU (decorate-sort-undecorate) – преобразование Шварца – шаблон, который предполагает создание списка кортежей, сортировку и извлечение части результата.
- Кортеж (tuple) – неизменяемая последовательность элементов.
- Присваивание кортежу (tuple assignment) – присваивание с последовательностью в правой части и кортежем переменных в левой части. Правая часть вычисляется, затем ее элементы присваиваются переменным в левой части.
- Разборка (scatter) – операция, интерпретирующая последовательность как список аргументов.
- Сборка (gather) – операция объединения аргумента в кортеж различной длины.
- Синглтон (singleton) – список (или другая последовательность) с одним элементом.
- Сравнимый (comparable) – тип, для которого можно проверить, что одно значение больше, меньше или равно другому значению того же типа. Значения сравнимых типов можно поместить в список и отсортировать.
- Структура данных (data structure) – набор связанных значений, часто организованный в виде списков, словарей, кортежей и т. п.
- Форма (структуры данных) (shape (of a data strucvture)) – общее описание типа, размера и построения структуры данных.
- Хешируемый (hashable) – тип, для которого существует хеш-функция. Неизменяемые типы, такие как целые числа, числа с плавающей точкой и строки, являются хешируемыми. Изменяемые типы, такие как списки и словари, хешируемыми не являются.

10.11. Упражнения

УПРАЖНЕНИЕ 1 Изменить ранее написанную программу следующим образом: прочитать и выполнить синтаксический анализ строк с префиксом «From» и извлечь адреса email из этих строк. Подсчитать число сообщений от каждого отправителя, используя словарь.

После того как все данные прочитаны, вывести отправителя с наибольшим числом коммитов, создав список кортежей (count, email) из словаря. Затем отсортируйте этот список в обратном порядке и выведите отправителя с наибольшим числом коммитов.

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

УПРАЖНЕНИЕ 2 Эта программа подсчитывает распределение по часам суток для всех сообщений. Значение часа можно извлечь из строки с префиксом «From», если найти в ней подстроку времени и разделить ее на части, используя символ двоеточия как разделитель. После того как накоплены счетчики для каждого часа, вывести эти счетчики по одному на строку с сортировкой по часам, как показано ниже.

```
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

УПРАЖНЕНИЕ 3 Написать программу, которая читает файл и выводит буквы в порядке убывания частоты. Программа должна выполнять преобразование всех вводимых букв в нижний регистр и считает только буквы a–z. Программа не должна считать пробелы, цифры, знаки пунктуации и какие-либо другие символы, кроме букв a–z. Найдите образцы текстов на нескольких разных языках и понаблюдайте, насколько частота букв различается в этих языках. Сравните полученные вами результаты с таблицами, расположенными здесь: https://wikipedia.org/wiki/Letter_frequencies; https://ru.wikipedia.org/wiki/Частотность.

Глава 11

Регулярные выражения

До настоящего момента мы считывали все содержимое файлов полностью, искали образцы и извлекали различные фрагменты строк, которые нас интересовали. При этом применялись такие методы строки, как `split` и `find`, а также списки и операция вырезки из строки для извлечения частей строк.

Задача поиска и извлечения информации встречается часто, поэтому в Python имеется весьма мощная библиотека обработки регулярных выражений (regular expressions), позволяющая решать многие из подобных задач изящно и эффективно. Причина, по которой регулярные выражения не были представлены раньше в этой книге, заключается в том, что, несмотря на их чрезвычайную мощь, регулярные выражения достаточно сложны, а к их синтаксису нужно немного привыкнуть.

Регулярные выражения – это почти самодостаточный специализированный язык программирования для поиска и синтаксического анализа строк текста. По правде говоря, на тему регулярных выражений написаны целые книги. В этой главе мы рассмотрим только основы регулярных выражений. Более подробно о регулярных выражениях можно узнать здесь:

https://en.wikipedia.org/wiki/Regular_expression;
https://ru.wikipedia.org/wiki/Регулярные_выражения;
https://docs.python.org/library/re.html.

Библиотеку обработки регулярных выражений `re` обязательно нужно импортировать в программу перед ее использованием. Самый простой способ ее практического применения – использование функции `search()`. В приведенной ниже программе демонстрируется простейший вариант применения функции `search`.

```
# Поиск строк, содержащих 'From'.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print(line)

# Исходный код: http://www.py4e.com/code3/re01.py
```

Мы открываем файл, проходим в цикле по каждой строке и используем функцию поиска по регулярному выражению `search()`, чтобы просто вывести

строки, содержащие подстроку «From:». В этой программе не используется истинная мощь регулярных выражений, так как можно было бы просто воспользоваться методом `line.find()` для получения того же результата.

Мощь регулярных выражений начинает проявляться, когда мы добавляем в строку поиска специальные символы, позволяющие более точно управлять образцами, определяющими искомые фрагменты строк. Добавление этих специальных символов в регулярные выражения обеспечивает более интеллектуальный поиск совпадений и извлечение при написании чрезвычайно малого объема кода.

Например, символ «крышка, карет (caret)» (^) используется в регулярных выражениях для определения «начала» строки. Приведенную выше программу можно изменить, чтобы она искала только те строки, в которых фрагмент «From:» находится в самом начале строки, как показано ниже:

```
# Поиск строк, которые начинаются с 'From'.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
        if re.search('^From:', line):
            print(line)

# Исходный код: http://www.py4e.com/code3/re02.py
```

Теперь мы ищем совпадение только в строках, начинающихся с подстроки «From:». Этот пример остается чрезвычайно простым, потому что можно было бы сделать то же самое с помощью метода `startswith()` из библиотеки обработки строк. Но такой пример служит для получения представления о форме записи регулярных выражений, содержащих специальные символы, которые обеспечивают более точное управление образцами для поиска совпадений.

11.1. Символы определения совпадений в регулярных выражениях

Существует несколько других специальных символов, позволяющих формировать еще более мощные регулярные выражения. Чаще всего используется специальный символ «точка» (.), который соответствует любому символу.

В приведенном ниже примере регулярное выражение `F..m:` будет совпадать с любой из строк «From:», «Fxxm:», «F12m:» или «F!@m:», поскольку символы точки в этом регулярном выражении соответствуют любому символу.

```
# Поиск строк, начинающихся с 'F', за которым следуют
# 2 любых символа, а после них располагается 'm:'.
import re
```

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)

# Исходный код: http://www.py4e.com/code3/re03.py
```

Это дает особенную мощь в сочетании с возможностью указать, что символ может повторяться любое число раз, если воспользоваться специальными символами * или + в регулярном выражении. Эти специальные символы означают, что вместо соответствия одному символу в строке поиска они соответствуют нулю и более символам (в случае использования звездочки) или одному и более символам (в случае использования знака плюс).

Можно сделать еще более узким диапазон искомых строк, применив повторяющийся символ универсального шаблона (wild card) в следующем примере:

```
# Поиск строк, начинающихся с From и содержащих символ "коммерческое at ".
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)

# Исходный код: http://www.py4e.com/code3/re04.py
```

Строка поиска `^From:.+@` успешно находит совпадение в строках, начинающихся с подстроки «From:», за которой следует один или несколько любых символов (.+), после чего встречается символ «коммерческое at». Следовательно, такая строка поиска соответствует следующей строке текста:

```
From: stephen.marquard@uct.ac.za
```

Можно считать шаблон (wild card) .+ расширением для определения соответствия всем символам между двоеточием и символом «коммерческое at».

```
From:.+@
```

Необходимо правильно воспринимать специальные символы «плюс» и «звездочка» как «агрессивные, пробивные». Например, в следующей строке должно быть определено совпадение с последним символом «коммерческое at», так как шаблон .+ агрессивно проталкивает поиск к концу строки, как показано ниже:

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen@iupui.edu
```

Есть возможность сообщить специальным символам «звездочка» и «плюс», что не следует быть такими «жадными», если добавить другой символ. См. в подробной документации информацию о настройке жадного поведения.

11.2. Извлечение данных с использованием регулярных выражений

Если необходимо извлечь данные из строки в Python, то можно воспользоваться методом `findall()` для извлечения всех подстрок, соответствующих заданному регулярному выражению. Рассмотрим пример попытки извлечения всего, что выглядит как адрес электронной почты из любой строки независимо от ее формата. Например, необходимо извлечь адреса электронной почты из каждой приведенной ниже строки:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

Не требуется писать код для каждого типа строки с выполнением разделения и вырезки в каждой строке отдельно. В приведенной ниже программе используется метод `findall()` для поиска строк с адресами электронной почты и извлечения одного или нескольких адресов из каждой строки.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)

# Исходный код: http://www.py4e.com/code3/re05.py
```

Метод `findall()` выполняет поиск в строке, заданной во втором аргументе, и возвращает список всех (под)строк, которые выглядят как адреса электронной почты. Здесь мы используем двухсимвольную последовательность, которая соответствует непробельному символу (`\S`).

Программа должна вывести следующий результат:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Интерпретация этого регулярного выражения: мы ищем подстроки, содержащие хотя бы один непробельный символ, за которым следует символ «коммерческое at», после которого расположен хотя бы один непробельный символ. Шаблон `\S+` соответствует любому возможному числу (не меньшему единицы) непробельных символов.

Это регулярное выражение должно обнаружить соответствие дважды (csev@umich.edu и cwen@iupui.edu), но не фиксировать соответствие для подстроки «@2PM», потому что перед символом «коммерческое at» отсутствует непробельный символ. Это регулярное выражение можно использовать в программе чтения всех строк из файла и вывода всего, что похоже на адрес электронной почты, как показано ниже:

```
# Поиск строк, содержащих символ "коммерческое at" между символами.
import re
```

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Исходный код: http://www.py4e.com/code3/re06.py
```

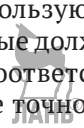
Здесь считывается каждая строка, затем извлекаются все подстроки, соответствующие заданному регулярному выражению. Поскольку метод `findall()` возвращает список, мы просто проверяем число элементов в этом возвращаемом списке, и если оно больше нуля, то выводятся только те строки, в которых найдена, как минимум, одна подстрока, похожая на адрес электронной почты.

Если запустить эту программу для обработки файла *mbox-short.txt*, то получим следующий вывод:


```
...
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
['cwen@iupui.edu']
['source@collab.sakaiproject.org']
['cwen@iupui.edu']
['cwen@iupui.edu']
['wagnermr@iupui.edu']
```

Некоторые из адресов электронной почты содержат некорректные символы, такие как «<» или «;» в начале или в конце. Примем дополнительное условие, определяющее, что нас интересует только та часть строки, которая начинается или заканчивается буквой или цифрой.

Для этого воспользуемся еще одним функциональным свойством регулярных выражений. Квадратные скобки используются для обозначения набора нескольких допустимых символов, которые должны учитываться при поиске соответствия. В этом смысле `\S` требует соответствия с набором «непробельных символов». Но теперь мы чуть более точно определим набор символов для поиска соответствия.



Ниже показано новое регулярное выражение:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

Оно стало немного сложнее, и вы, вероятно, начинаете понимать, почему регулярные выражения сами по себе являются небольшим языком программирования. Расшифровка этого регулярного выражения: мы ищем подстроки, начинающиеся с одной буквы в нижнем или верхнем регистре или цифры (`[a-zA-Z0-9]`), за которой следует ноль или более непробельных символов (`\S*`), потом коммерческое at, далее ноль или более непробельных символов (`\S*`), затем одна буква в нижнем или верхнем регистре (`[a-zA-Z]`). Обратите внимание: мы заменили + на * для указания нуля или более непробельных

символов, поскольку [a-zA-Z0-9] – это уже один непробельный символ. Напомню, что * или + применяются к одному символу, расположенному непосредственно слева от плюса или звездочки.

Если использовать это регулярное выражение в нашей программе, то извлекаемые данные будут более точными («чистыми»):

```
# Поиск строк, содержащих коммерческое at между символами.
# Символы обязательно должны быть буквой или цифрой.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0:
        print(x)

# Исходный код: http://www.py4e.com/code3/re07.py
```

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Следует отметить, что в строках source@collab.sakaiproject.org наше регулярное выражение удаляет два символа в конце строки (>;). Причина этого в том, что добавление [a-zA-Z] в конец регулярного выражения требует, чтобы любая обрабатываемая парсером регулярных выражений строка обязательно завершалась буквой. Поэтому когда в конце подстроки «sakaiproject.org>;» обнаруживается символ «>», парсер просто останавливается на последней найденной «совпадающей» букве (т. е. «g» – последнее правильное совпадение).

Также следует отметить, что выводом этой программы является список Python, содержащий строку как единственный элемент этого списка.

11.3. Объединение поиска и извлечения

Если необходимо найти числа в строках, начинающихся с подстроки «X-», подобных показанным ниже:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

то нам нужны не просто числа с плавающей точкой из любых строк. Требуется извлечение чисел из конкретных строк с заданным выше синтаксисом.

Для отбора таких строк можно сформировать следующее регулярное выражение:

```
^X-.*: [0-9.]+
```

Здесь мы сообщаем, что требуются строки, начинающиеся с подстроки `X-`, за которой следует ноль или более символов (.*), затем двоеточие (:), потом пробел. После пробела мы ищем один или более символов, являющихся либо цифрой (0–9), либо точкой `[0-9.]+`. Обратите внимание: внутри квадратных скобок точка соответствует настоящей точке (т. е. это не универсальный шаблон соответствия любому символу, если находится внутри квадратных скобок).

Это весьма строгое регулярное выражение, которое в точности соответствует только тем строкам, которые нас интересуют, как показано ниже:

```
# Поиск строк, начинающихся с 'X', за которым следуют любые непробельные символы
# и ':', потом пробел и любое число.
# Число может содержать десятичную точку.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)

# Исходный код: http://www.py4e.com/code3/re10.py
```

После запуска этой программы мы видим аккуратно отфильтрованные данные, показывающие только те строки, которые мы искали.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
...
```

Но теперь нужно решить задачу извлечения чисел. Ее можно было бы решить достаточно просто, используя метод `split`, но мы воспользуемся еще одним функциональным свойством регулярных выражений, обеспечивающим поиск и синтаксический анализ строки одновременно.

Круглые скобки – еще один специальный символ в регулярных выражениях. Когда вы добавляете круглые скобки в регулярное выражение, они не учитываются при определении соответствия с искомой строкой. Но при использовании метода `findall()` круглые скобки означают, что хотя требуется соответствие со всем выражением в целом, вас интересует только извлечение той части подстроки, которая соответствует заданному регулярному выражению.

Поэтому мы вносим в нашу программу следующее изменение:

```
# Поиск строк, начинающихся с 'X', за которым следуют любые непробельные символы
# и ':', потом пробел и любое число.
```

```
# Число может содержать десятичную точку.
# Затем выводится найденное число, если его длина больше нуля.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Исходный код: http://www.py4e.com/code3/re11.py
```


Вместо вызова `search()` мы добавляем круглые скобки, охватывающие ту часть регулярного выражения, которая представляет число с плавающей точкой, чтобы сообщить методу `findall()` о необходимости возврата только числа с плавающей точкой из совпадающей строки.

Вывод этой версии программы показан ниже:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
...
```

Числа остаются в списке как строки, и требуется преобразование из этих строк в числа с плавающей точкой, но здесь мы использовали мощь регулярных выражений лишь для поиска и извлечения интересующей нас информации.

Еще один пример применения этой методики при просмотре файла, в котором имеется некоторое количество строк следующего формата:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Если необходимо извлечь все номера корректировок (целое число в конце таких строк) с применением описанной выше методики, то можно написать следующую программу:

```
# Поиск строк, начинающихся с 'Details:', затем после любого числа символов следует
# подстрока 'rev=', за которой следуют цифровые символы.
# Выводится число, если оно было найдено.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9]+)', line)
    if len(x) > 0:
        print(x)

# Исходный код: http://www.py4e.com/code3/re12.py
```

Расшифровка регулярного выражения: мы ищем строки, начинающиеся с подстроки `Details:`, за которой следует любое число символов (.*), далее подстрока `rev=`, после нее одна или более цифр. Необходимо найти строки, соответствующие всему регулярному выражению в целом, но требуется извлечь только целое число в конце найденной строки, поэтому часть `[0-9]+` заключается в круглые скобки.

При запуске этой программы получим следующий вывод:

```
['39772']
['39771']
['39770']
['39769']
...
```

Напомню, что шаблон `[0-9]+` «жадный», и он пытается сформировать наибольшую возможную строку из цифр перед их извлечением. Такое «жадное» поведение является причиной того, что мы получаем все пять цифр каждого числа. Библиотека регулярных выражений распространяет подобное поведение в обоих направлениях до тех пор, пока не встретится нецифровой символ, или начало, или конец строки.

Теперь мы можем использовать регулярные выражения для повторного выполнения упражнения из предыдущих глав книги, в котором нас интересовало время суток, указанное в каждом сообщении электронной почты. Мы искали строки формата:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

и должны были извлечь значение часа суток из каждой строки. Ранее мы делали это с помощью двух вызовов метода `split`. Сначала строка разделялась на слова, затем извлекалось пятое слово и снова разделялось по символу двоеточия, чтобы извлечь два символа, которые нас интересовали.

Этот подход работал, хотя в результате получился весьма ненадежный код, который предполагал, что все строки корректно отформатированы. Если бы мы добавили код, достаточный для проверки ошибок (или крупный блок `try/except`), для уверенности в том, что программа никогда не завершится крахом при обработке некорректно отформатированных строк, то объем кода вырос бы до 10–15 строк, достаточно трудных для чтения.

Но то же самое можно сделать гораздо проще, если воспользоваться приведенным ниже регулярным выражением:

```
^From .* [0-9][0-9]:
```

Расшифровка этого регулярного выражения: мы ищем строки, начинающиеся с префикса «`From` » (обратите внимание на пробел), за которым следует любое число символов (.*), потом пробел и две цифры `[0-9][0-9]`, после которых расположено двоеточие. Это определение того типа строк, которые мы ищем.

Чтобы извлечь только значение часа с использованием метода `findall()`, мы добавляем круглые скобки, охватывающие две цифры, как показано ниже:

```
^From .* ([0-9][0-9]):
```

В итоге получаем следующую программу:

```
# Поиск строк, начинающихся с From и содержащих любые символы,
# за которыми следуют две цифры от 00 до 99, после них ':'.
# Затем выводится только строка найденных цифр, если ее длина больше нуля.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0: print(x)

# Исходный код: http://www.py4e.com/code3/re13.py
```

После запуска этой программы получаем следующий вывод:

```
['09']
['18']
['16']
['15']
...
```

11.4. Специальный символ экранирования (escape)

Поскольку мы используем специальные символы в регулярных выражениях для определения соответствия началу или концу строки или конкретных шаблонов, необходим способ, позволяющий указать, что эти символы являются «обычными», и требуется искать совпадение с реальным символом, таким как знак доллара или «крышка» (caret – карет).

Мы можем обозначить свое намерение просто найти совпадение с каким-либо (в том числе и специальным) символом, записав перед ним обратный слеш (бэкслеш – backslash). Например, можно найти денежную сумму с помощью следующего регулярного выражения.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Поскольку перед символом доллара находится обратный слеш, в действительности определяется совпадение именно с этим символом во входной строке, а не поиск «конца строки», а остальная часть регулярного выражения ищет одну или несколько цифр или точку. Примечание: внутри квадратных скобок символы перестают быть специальными. Поэтому, когда мы пишем `[0-9.]`, в действительности это означает цифры или точку. За пределами квадратных скобок точка представляет собой специальный шаблон (wild card), который соответствует любому символу. В квадратных скобках точка – это всегда точка.

11.5. Итоговый обзор специальных символов

Мы рассмотрели только лишь самые основные свойства регулярных выражений, но все же познакомились в общих чертах с этим специализированным языком. Регулярные выражения позволяют выполнять поиск в строках с использованием специальных символов, с помощью которых мы передаем свои намерения и требования в систему (механизм) регулярных выражений в форме, которая определяет «соответствие» и те фрагменты, которые должны извлекаться из «соответствующих» строк. Ниже кратко описаны некоторые из таких специальных символов и символьных последовательностей:

- ^ – соответствует началу строки;
- $ – соответствует концу строки;
- . – соответствует любому символу (универсальный шаблон);
- \s – соответствует пробельному символу;
- \S – соответствует непробельному символу (смысл противоположен \s);
- * – применяется к непосредственно предшествующему символу (символам) и обозначает соответствие ноль или более раз;
- *? – применяется к непосредственно предшествующему символу (символам) и обозначает соответствие ноль или более раз в «нежадном режиме»;
- + – применяется к непосредственно предшествующему символу (символам) и обозначает соответствие один или более раз;
- +? – применяется к непосредственно предшествующему символу (символам) и обозначает соответствие один или более раз в «нежадном режиме»;
- ? – применяется к непосредственно предшествующему символу (символам) и обозначает соответствие ноль или один раз;
- ?? – применяется к непосредственно предшествующему символу (символам) и обозначает соответствие ноль или один раз в «нежадном режиме»;
- [aeiou] – соответствует одному символу, если это символ из заданного в квадратных скобках набора. В данном примере символ должен соответствовать «a», «e», «i», «o» или «u», не никакому другому символу;
- [a-z0-9] – можно определять диапазоны символов, используя знак минус. В этом примере один символ должен соответствовать любой букве в нижнем регистре или цифре;
- [^A-Za-z] – если самым первым символом в наборе внутри квадратных скобок является карет («крышка»), то он обозначает логическую инверсию (отрицание). В этом примере один символ должен соответствовать только тому символу, который не является буквой в верхнем или нижнем регистре;
- () – если круглые скобки добавляются в регулярное выражение, то они не учитываются при поиске соответствия, но позволяют извлечь обозначенное ими конкретное подмножество совпадающей строки, а не всю строку в целом, когда применяется метод findall();

- \b – соответствует пустой строке, но только в начале или в конце слова;
- \B – соответствует пустой строке, но не в начале или в конце слова;
- \d – соответствует любой десятичной цифре – равнозначен набору [0-9];
- \D – соответствует любому нецифровому символу – равнозначен набору [^0-9].

11.6. Дополнительный раздел для пользователей систем Unix/Linux

Поддержка поиска файлов с использованием регулярных выражений была встроена в операционную систему Unix еще с 1960-х гг.[1], а сейчас доступна почти во всех языках программирования в той или иной форме.

Существует программа (утилита) командной строки, входящая в комплект Unix-подобных систем, под названием grep (Generalized Regular Expression Parser[2]), которая делает почти то же самое, что метод search() в примерах этой главы. Если вы работаете в системе Macintosh или Linux, то можете попробовать выполнить приведенные ниже команды в окне командной строки (терминале).

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

Эта команда сообщает grep о необходимости вывода строк, начинающихся с подстроки «From:», из файла *mbox-short.txt*. Если вы немного поэкспериментируете с командой grep и внимательно изучите ее документацию, то обнаружите некоторые тонкие различия между поддержкой регулярных выражений в Python и в grep. Утилита grep не поддерживает специальный символ \S, соответствующий непробельным символам, поэтому потребуется использование немного более сложной формы записи [^], означающей соответствие символу, отличающемуся от пробела[3].

---

[1] Здесь автор допускает неточность: разработка Unix началась в 1969 г., первая пробная работоспособная версия выпущена в 1970 г. (поэтому отсчет системного времени производится с 1 января 1970 г.), а утилита grep, выполняющая поиск с использованием регулярных выражений, появилась в ноябре 1973 г. – *Прим. перев.*

[2] Это не совсем точно – сами авторы утилиты называли различные версии происхождения названия: **g**lobally search for a **r**egular **e**xpression and **p**rint matching lines или search **g**lobally for lines matching the **r**egular **e**xpression, and **p**rint them (глобальный поиск строк, соответствующих регулярному выражению, и их вывод) и т. п. Указанная автором расшифровка grep появилась гораздо позже. – *Прим. перев.*

[3] Также можно воспользоваться утилитой egrep (равнозначна grep -E), которая поддерживает регулярные выражения в полном объеме. – *Прим. перев.*

11.7. Отладка

В Python имеется упрощенная встроенная документация, которая может оказаться достаточно полезной, если требуется быстро освежить в памяти информацию по точному имени конкретного метода (или функции). Эту документацию можно просматривать в интерпретаторе Python в интерактивном режиме.

Интерактивную систему помощи можно вызвать с помощью встроенной функции `help()`.

```
>>> help()

help> modules
```

Если вы знаете, какой модуль предполагаете использовать, то можно воспользоваться командой `dir()` для поиска методов в этом модуле, как показано ниже:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

Кроме того, можно получить небольшой фрагмент документации по конкретному методу, используя команду `dir`.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

Эта встроенная документация не очень информативна, но может оказаться полезной, когда вам необходимо срочно получить хотя бы краткое описание или если нет доступа к веб-браузеру либо механизму поиска.

11.8. Словарь терминов

- `grep` – команда (утилита), доступная в большинстве Unix-подобных систем, которая выполняет поиск в текстовых файлах строк, соответствующих заданным регулярным выражениям. Имя команды является аббревиатурой фразы «Generalized Regular Expression Parser»[1].
- Жадный поиск соответствия (greedy matching) – формат, в котором специальные символы + и * в регулярном выражении расширяют свое действие для поиска совпадения с наибольшей возможной строкой.

[1] См. прим. перев. выше.

- Ненадежный код (brittle code) – код, который работает, если входные данные представлены в определенном корректном формате, но с большой вероятностью перестает работать при любом отклонении от корректного формата данных. Такой код называется ненадежным, потому что он с легкостью приводит к критическому отказу.
- Регулярное выражение (regular expression) – специализированный язык для записи более сложного представления строки поиска. Регулярное выражение может содержать специальные символы, которые позволяют искать соответствия только в начале или в конце строки, а также определять многие другие аналогичные характеристики.
- Шаблон (универсальный) (wild card) – специальный символ, который соответствует любому символу. В регулярных выражениях символом универсального шаблона является точка[1].

11.9. Упражнения

УПРАЖНЕНИЕ 1 Написать простую программу, имитирующую работу команды grep в Unix-подобных системах. Предложить пользователю ввести регулярное выражение и подсчитать число строк, соответствующих этому регулярному выражению.

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4175 lines that matched java$
```

УПРАЖНЕНИЕ 2 Написать программу поиска строк следующей формы:

```
New Revision: 39772
```

Извлечь число из каждой такой строки, используя регулярное выражение и метод `findall()`. Вычислить среднее арифметическое извлеченных чисел и вывести полученное значение как целое число.

```
Enter file:mbox.txt
38549

Enter file:mbox-short.txt
39756
```

[1] И здесь автор не совсем точен: вообще говоря, к шаблонам (wild card) относятся также специальные символы +, * и ?. – *Прим. перев.*

Глава 12

Сетевые программы

Многие примеры в этой книге сосредоточены на чтении файлов и поиске информации в них, но существует и множество разнообразных источников информации, если принять во внимание доступность интернета.

В этой главе мы будем имитировать работу веб-браузера и извлекать веб-страницы с использованием протокола HTTP (Hypertext Transfer Protocol). Затем мы будем считывать данные из полученной веб-страницы и выполнять их синтаксический анализ.

12.1. Протокол HTTP – Hypertext Transfer Protocol

Сетевой протокол, обеспечивающий мощь веб-среды, в действительности относительно прост, и существует встроенная поддержка в Python, называемая `socket`, которая существенно упрощает установление сетевых соединений и извлечение данных через эти сокеты в программе на языке Python.

Сокет (socket) во многом похож на файл, за исключением того, что один сокет обеспечивает двунаправленное соединение между двумя программами. Вы можете читать из сокета и записывать в него. Если вы что-то записываете в сокет, то он отправляет эти данные приложению на другом конце соединения. Если вы читаете из сокета, то получаете данные, отправленные другим приложением.

Но если вы пытаетесь читать из сокета, когда программа на другом конце соединения не отправила никакие данные, то будете просто сидеть и ждать. Если программы на другом конце соединения просто ждут некоторые данные без отправки каких-либо своих данных, то ожидание продлится очень долго, поэтому важной частью программ, обменивающихся информацией через интернет, является наличие некоторого протокола.

Протокол – это набор точных правил, определяющих, кто начинает «диалог» первым, что делают участники «диалога», как следует отвечать на первое сообщение, кто передает следующее сообщение и т. д. В некотором смысле два приложения на разных концах соединения (сокета) танцуют, но при этом не должны наступать на ноги партнера.

Существует множество документов, описывающих эти сетевые протоколы. Протокол HTTP – Hypertext Transfer Protocol – описан в следующем документе:

https://www.w3.org/Protocols/rfc2616/rfc2616.txt.

Это длинный и сложный 176-страничный документ с описанием огромного количества подробностей. Если вы интересуетесь, то можете прочитать его полностью. Но если открыть страницу 36 документа RFC2616, то вы обнаружите там описание синтаксиса запроса GET. Для запроса документа с веб-сервера мы устанавливаем соединение с сервером `www.pr4e.org`, порт 80, а затем отправляем строку следующего формата:

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

Здесь второй параметр – запрашиваемая веб-страница, а затем мы также передаем пустую строку. Веб-сервер ответит передачей некоторого заголовка с информацией об этом документе и пустой строкой, за которой следует содержимое документа.

12.2. Самый простой в мире веб-браузер

Вероятно, самым простым способом демонстрации работы протокола HTTP является написание весьма простой программы на языке Python, которая устанавливает соединение с веб-сервером и соблюдает правила протокола HTTP для запроса документа и вывода того, что возвращает сервер.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if len(data) < 1:
        break
    print(data.decode(),end='')

mysock.close()

# Исходный код: http://www.py4e.com/code3/socket1.py
```

Сначала программа устанавливает соединение с портом 80 на сервере www.py4e.com. Поскольку наша программа исполняет роль веб-браузера, протокол HTTP утверждает, что мы обязательно должны передать команду GET, за которой следует пустая строка. Символы `\r\n` обозначают EOL (end of line – конец строки), а комбинация символов `\r\n\r\n` означает, что между двумя последовательностями EOL ничего нет. Это равнозначно пустой строке.

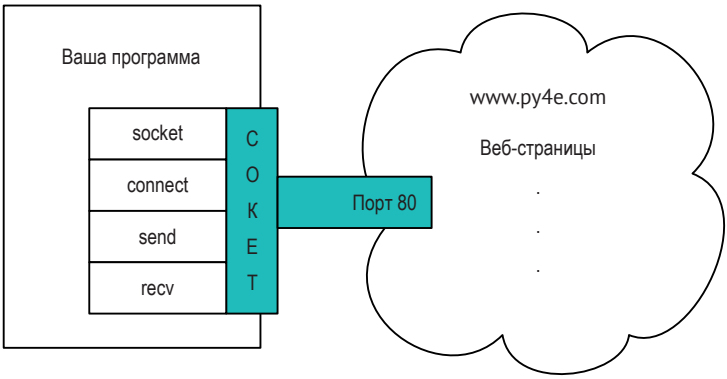


Рис. 12.1 ❖ Соединение через сокет

После передачи этой пустой строки мы пишем цикл, который принимает данные в виде 512-символьных фрагментов из сокета и выводит их до тех пор, пока читаемых данных не останется (т. е. когда метод сокета `recv()` возвращает пустую строку).

Вывод этой программы показан ниже:

```
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Вывод начинается с заголовков, которые веб-сервер передает для описания документа. Например, заголовок `Content-Type` сообщает, что передан документ с простым (неформатированным) текстом (`text/plain`).

После того как сервер передает нам заголовки, он добавляет пустую строку, чтобы обозначить конец заголовков, а затем отправляет реальные данные из файла *romeo.txt*.

Этот пример показывает, как устанавливается низкоуровневое сетевое соединение с помощью сокетов. Сокеты можно использовать для обмена информацией с веб-сервером, с почтовым сервером или с другими типами серверов. Все, что необходимо для этого, – найти документ, описывающий требуемый протокол, и написать код для передачи и приема данных в соответствии с этим протоколом.

Но поскольку мы чаще всего используем веб-протокол HTTP, в Python есть библиотека, специально предназначенная для поддержки протокола HTTP для извлечения документов и данных из веб-среды.

Одно из обязательных требований при использовании протокола HTTP – необходимость передачи и приема данных как байтовых объектов, а не строк. В показанном выше примере методы сокета `encode()` и `decode()` выполняют преобразование строк в байтовые объекты и обратно.

В следующем примере используется форма записи `b''`, чтобы определить, что переменная должна быть сохранена как байтовый объект. Метод `encode()` и форма записи `b''` равнозначны.

```
>>> b'Hello world'
b'Hello world'
>>> 'Hello world'.encode()
b'Hello world'
```

12.3. Извлечение изображения с использованием протокола HTTP

В показанном выше примере мы извлекли обычный текстовый файл, содержащий символы конца строк, и просто скопировали эти данные на экран при запуске программы. Можно использовать аналогичную программу для извлечения изображения с применением протокола HTTP. Вместо копирования данных на экран при запуске программы мы накапливаем данные в строке, отсекаем заголовки, затем сохраняем данные изображения в файле, как показано ниже:

```
import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
count = 0
picture = b""

while True:
    data = mysock.recv(5120)
    if len(data) < 1: break
    #time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()
```

```
# Поиск конца заголовка (2 символа CRLF).
pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
print(picture[:pos].decode())

# Пропустить весь заголовок и пустую строку и сохранить данные изображения.
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()

# Исходный код: http://www.py4e.com/code3/urljpeg.py
```

При запуске программа выводит следующую информацию:

```
$ python urljpeg.py
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:54:09 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Здесь можно видеть, что для этого указателя сетевого ресурса URL (uniform resource locator) заголовок `Content-Type` сообщает, что телом этого документа является изображение (`image/jpeg`). После завершения работы программы можно посмотреть данные изображения (точнее, само изображение), открыв файл *stuff.jpg* в любой программе просмотра изображений.

При запуске программы можно заметить, что при вызове метода `recv()` мы не всегда получаем заявленные 5120 символов. Мы принимаем столько символов, сколько было передано по сети веб-сервером в момент вызова метода `recv()`. В показанном выше примере в некоторый момент мы получили всего лишь 3200 символов данных вместо затребованных 5120.

Эти результаты могут быть различными, поскольку зависят от конкретной скорости вашего сетевого соединения. Кроме того, следует отметить, что при последнем вызове `recv()` мы получили 3167 байт, потому что это был конец потока, а следующий вызов `recv()` вернул строку нулевой длины, и это стало для нас признаком того, что сервер вызвал метод `close()` на своем конце соединения (т. е. закрыл сокет), и данные больше передаваться не будут.

Мы можем немного замедлить последовательность вызовов метода `recv()`, удалив символ комментария из строки вызова метода `time.sleep()`. В рассматриваемом здесь примере мы делаем паузу в четверть секунды после каждого вызова, чтобы сервер мог «обогнать» нас и передать больше данных, прежде чем метод `recv()` будет вызван в очередной раз. С такой задержкой программа выполняется так, как показано ниже:

```
$ python urljpeg.py
5120 5120
5120 10240
5120 15360
...
5120 225280
5120 230400
207 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 21:42:08 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Теперь мы получаем 5120 символов при каждом запросе новых данных, за исключением самого последнего вызова `recv()`.

Существует буфер между сервером, выполняющим запросы `send()`, и нашим приложением, выполняющим запросы `recv()`. При запуске программы с такой задержкой в некоторый момент сервер может заполнить буфер в сокете и вынужден сделать паузу до тех пор, пока наша программа не начнет освобождать буфер. Создание паузы передающим или принимающим приложением называется управлением потоком (flow control).

12.4. Извлечение веб-страниц с помощью библиотеки urllib

Хотя мы можем вручную отправлять и принимать данные по протоколу HTTP, используя библиотеку socket, существует гораздо более простой способ выполнения этой часто встречающейся задачи на языке Python – применение библиотеки urllib.

Используя библиотеку urllib, мы можем интерпретировать веб-страницу почти как обычный файл. Вы просто определяете веб-страницу, которую необходимо извлечь, а urllib обеспечивает соблюдение протокола HTTP и всех подробностей заголовка.

Равнозначный код для чтения файла *romeo.txt* с веб-сервера с применением библиотеки urllib показан ниже:

```
import urllib.request
fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())

# Исходный код: http://www.py4e.com/code3/urllib1.py
```


После того как веб-страница открыта с помощью метода urllib.request.urlopen, ее можно интерпретировать как файл и читать ее содержимое, используя цикл for.

После запуска этой программы мы видим только содержимое запрошенного файла. Заголовки продолжают передаваться, но внутренний код urllib скрывает их и возвращает пользователю лишь данные.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

В качестве следующего примера можно написать программу, которая извлекает данные из файла *romeo.txt* и вычисляет частоту каждого слова в этом файле, как показано ниже:

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

# Исходный код: http://www.py4e.com/code3/urlwords.py
```

И в этом примере, после того как веб-страница открыта, мы можем читать ее как локальный файл.

12.5. Чтение двоичных файлов с использованием библиотеки `urllib`

Иногда необходимо извлечь не текстовый, а двоичный (или бинарный) файл, например изображение или видеофайл. Данные в таких файлах обычно бесполезно выводить на экран в «сыром» виде, но можно с легкостью создать копию требуемого URL в локальном файле на жестком диске, воспользовавшись библиотекой `urllib`.

Шаблон решения этой задачи: открыть URL, использовать метод `read` для загрузки всего содержимого требуемого документа в строковую переменную (`img`), затем записать полученную информацию в локальный файл, как показано ниже:

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()

# Исходный код: http://www.py4e.com/code3/curl1.py
```

Программа считывает все данные за один прием по сети и сохраняет их в переменной `img` в основной памяти вашего компьютера, затем открывает файл *cover3.jpg* и записывает в него данные, сохраняя их на локальном диске. Аргумент `wb` для функции `open` позволяет открыть двоичный (**b**inary) файл только для записи (**w**rite). Программа будет работать корректно, если размер запрашиваемого файла меньше, чем объем основной памяти вашего компьютера.

Но если это большой аудио- или видеофайл, то в программе может возникнуть критический сбой, или, в лучшем случае, она будет работать чрезвычайно медленно, когда будет исчерпана основная память. Чтобы избежать переполнения памяти, мы извлекаем данные блоками (или буферами), а затем записываем каждый блок на диск, прежде чем извлечь следующий блок. Такой способ позволяет программе читать файл любого размера без переполнения основной памяти компьютера.

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0
while True:
    info = img.read(100000)
```

```
    if len(info) < 1: break
    size = size + len(info)
    fhand.write(info)

print(size, 'characters copied.')
fhand.close()

# Исходный код: http://www.py4e.com/code3/curl2.py
```

В этом примере мы считываем только 100 000 символов за один прием, а затем записываем их в файл *cover3.jpg*, перед тем как извлечь следующие 100 000 символов данных с веб-сервера.

Программа при выполнении выводит следующую информацию:

```
python curl2.py
230210 characters copied.
```

12.6. Синтаксический анализ формата HTML и веб-скрейпинг

Одним из наиболее частых вариантов использования функциональных возможностей библиотеки `urllib` в Python-программах является веб-скрейпинг (web-scraping). Веб-скрейпинг производится, когда мы пишем программу, которая имитирует работу веб-браузера и извлекает страницы, а затем исследует данные на этих страницах, выполняя поиск по образцам.

В качестве примера можно привести механизм поиска, например применяемый Google, который рассматривает источник одной веб-страницы и извлекает все ссылки (links) на другие страницы, затем получает эти страницы, извлекает из них ссылки и т. д. Используя эту методику, поисковый механизм Google, подобно пауку, распространяет пути поиска почти по всем страницам в веб-среде.

Google также использует характеристику частотности ссылок со страниц, которые этот механизм находит, на конкретную страницу, как количественную оценку «важности» этой страницы и использует эту оценку при определении места страницы в списке результатов поиска.

12.7. Синтаксический анализ формата HTML с использованием регулярных выражений

Один из самых простых способов синтаксического анализа формата HTML – использование регулярных выражений для многократно повторяющегося поиска и извлечения подстрок, соответствующих конкретному заданному образцу (шаблону).

Ниже приведен пример (исходного кода) простой веб-страницы:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

Можно создать корректно сформированное регулярное выражение для поиска соответствия и извлечения значений ссылок из приведенного выше текста, как показано ниже:

```
href="http[s]?://.+?"
```

Это регулярное выражение ищет строки, начинающиеся с подстроки `href="http://` или `href="https://`, за которой следует один или более символов (`.+?`), потом еще одна двойная кавычка. Знак вопроса после `[s]?` означает поиск подстроки «http», за которой следует ноль или одна буква «s».

Знак вопроса, добавленный к группе символов `.+?`, означает, что поиск совпадения не должен быть «жадным». Нежадное совпадение пытается найти наименьшую возможную совпадающую строку, а жадное совпадение – наибольшую возможную совпадающую строку.

В это регулярное выражение мы добавляем круглые скобки, чтобы определить, какую часть совпадающей строки необходимо извлечь, и пишем следующую программу:

```
# Поиск значений ссылок во введенном URL.
import urllib.request, urllib.parse, urllib.error
import re
import ssl

# Игнорировать ошибки проверки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
links = re.findall(b'href="(http[s]?://.*?)"', html)
for link in links:
    print(link.decode())

# Исходный код: http://www.py4e.com/code3/urlregex.py
```

Библиотека `ssl` позволяет этой программе получить доступ к веб-сайтам, которые строго требуют применения протокола HTTPS. Метод `read` возвращает исходный код HTML как байтовый объект вместо возвращения объекта `HTTPResponse`. Метод `findall` с заданным регулярным выражением возвращает список всех строк, соответствующих этому регулярному выражению, при этом извлекая только текст ссылки между двойными кавычками.

При запуске программы и вводе URL получим следующий вывод:

```
Enter - https://docs.python.org
https://docs.python.org/3/index.html
https://www.python.org/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://www.python.org/
https://www.python.org/psf/donations/
http://sphinx.pocoo.org/
```

Регулярные выражения работают весьма успешно, если код HTML правильно отформатирован и предсказуем. Но поскольку в интернете существует множество «неправильных» HTML-страниц, решение с использованием только регулярных выражений может пропустить некоторые корректные ссылки или завершиться с неверными данными.

Эту проблему можно решить, если воспользоваться надежной библиотекой синтаксического анализа кода HTML.

12.8. Синтаксический анализ формата HTML с использованием BeautifulSoup

Несмотря на то что код HTML внешне похож на XML[1] и некоторые страницы скомпонованы настолько аккуратно, что неотличимы от XML, большинство кода HTML содержит некорректные особенности, которые заставляют парсер языка XML отвергать всю HTML-страницу в целом как неправильно сформированную.

Существует несколько библиотек Python, которые могут помочь при выполнении синтаксического анализа кода HTML и извлечении данных с таких страниц. Каждая библиотека имеет свои сильные и слабые стороны, поэтому вы можете выбрать одну из них, исходя из своих потребностей.

В качестве примера выполним простой синтаксический анализ введенного кода HTML с извлечением ссылок, используя библиотеку BeautifulSoup. Эта библиотека в полной мере поддерживает HTML, а также позволяет с легкостью извлекать требуемые данные. Код библиотеки BeautifulSoup, необходимый для установки, можно скачать здесь:

https://pypi.python.org/pypi/beautifulsoup4.

---

[1] Формат XML описан в следующей главе.

Информацию об установке BeautifulSoup с помощью средства управления пакетами Python Package Index `pip` можно найти здесь:

https://packaging.python.org/tutorials/installing-packages/.

Мы будем использовать библиотеку `urllib` для чтения страницы, затем библиотеку `BeautifulSoup` для извлечения атрибутов `href` из тегов анкера (`a`).

```
# Для запуска этой программы необходимо скачать zip-файл BeautifulSoup здесь:
# http://www.py4e.com/code3/bs4.zip
# и разархивировать его в том каталоге, где находится файл этой программы.

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Игнорировать ошибки проверки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Извлечение всего содержимого из тегов анкера.
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Исходный код: http://www.py4e.com/code3/urllinks.py
```

Программа предлагает ввести веб-адрес, затем открывает заданную веб-страницу, считывает ее данные и передает их в парсер BeautifulSoup, после чего извлекает все теги анкера и выводит значение атрибута `href` для каждого тега.

При запуске программы получаем следующий вывод:

```
Enter - https://docs.python.org
genindex.html
py-modindex.html
https://www.python.org/
#
whatsnew/3.6.html
whatsnew/index.html
tutorial/index.html
library/index.html
reference/index.html
using/index.html
howto/index.html
installing/index.html
distributing/index.html
extending/index.html
c-api/index.html
```

```
faq/index.html
py-modindex.html
genindex.html
glossary.html
search.html
contents.html
bugs.html
about.html
license.html
copyright.html
download.html
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
genindex.html
py-modindex.html
https://www.python.org/
#
copyright.html
https://www.python.org/psf/donations/
bugs.html
http://sphinx.pocoo.org/
```

Этот список гораздо длиннее, потому что некоторые теги анкеров HTML являются относительными путями (например, `tutorial/index.html`) или ссылками внутри страницы (например, `#`), которые не содержат подстрок `http://` или `https://`, требуемых в обязательном порядке в нашем регулярном выражении.

Также можно использовать библиотеку BeautifulSoup для извлечения различных частей каждого тега:

```
# Для запуска этой программы необходимо скачать zip-файл BeautifulSoup здесь:
# http://www.py4e.com/code3/bs4.zip
# и разархивировать его в том каталоге, где находится файл этой программы.

from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl

# Игнорировать ошибки проверки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, "html.parser")
```

```
# Извлечение всего содержимого из тегов анкера.
tags = soup('a')
for tag in tags:
    # Выделение частей каждого тега.
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)

# Исходный код: http://www.py4e.com/code3/urllink2.py
```

```
python urllink2.py
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: ['\nSecond Page']
Attrs: [('href', 'http://www.dr-chuck.com/page2.htm')]
```

Метод `html.parser` – это парсер языка HTML, включенный в стандартную библиотеку Python 3. Информация о других парсерах HTML доступна здесь:

http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser.

Эти примеры демонстрируют лишь малую степень мощи, которой обладает BeautifulSoup в плане синтаксического анализа языка HTML.

12.9. Дополнительный раздел для пользователей систем Unix/Linux

Если вы пользуетесь компьютером с системой Linux, Unix или Macintosh, то, вероятно, применяете команды, встроенные в операционную систему[1], которые извлекают обычный текст и двоичные файлы, используя протоколы HTTP и FTP (File Transfer Protocol). Одной из таких команд является `curl`:

```
$ curl -O http://www.py4e.com/cover.jpg
```

Имя команды `curl` образовано сокращением фразы «copy URL», поэтому два примера из предыдущих разделов, в которых извлекались двоичные файлы с помощью библиотеки `urllib`, были обоснованно названы *curl1.py* и *curl2.py* на странице www.py4e.com/code3, так как в них реализована функциональность, почти такая же, как у команды `curl`. Существует еще и пример программы *curl3.py*, которая выполняет ту же задачу чуть более эффективно, и вы можете, если пожелаете, использовать этот пример в своей программе.

[1] Еще одна неточность автора: описанные ниже команды не являются встроенными в операционную систему, это отдельные программы-утилиты, которые входят в дистрибутивный комплект. Кроме того, существуют версии cURL и wget для Windows, поэтому данный раздел полезен и для пользователей этой ОС. – *Прим. перев.*

Вторая команда с аналогичной функциональностью – `wget`:

```
$ wget http://www.py4e.com/cover.jpg
```

Обе эти команды превращают скачивание веб-страниц и файлов, расположенных в удаленных локациях, в весьма простую задачу.

12.10. Словарь терминов

- BeautifulSoup – библиотека Python для синтаксического анализа документов в формате HTML и извлечения данных из HTML-документов, сглаживающая большинство недостатков в коде HTML, которые браузеры обычно игнорируют. Код библиотеки BeautifulSoup можно загрузить здесь: www.crummy.com.
- Веб-скрейпинг (scraping) – работа программы, имитирующая веб-браузер с извлечением веб-страниц и просмотром их содержимого. Часто такие программы следуют по ссылкам на одной странице, чтобы найти следующую страницу; действуя так, они могут обойти целую подсеть страниц или социальную сеть.
- Паук (spider) – деятельность механизма поиска в веб-среде по извлечению страницы с последующими переходами по ссылкам с этой страницы и т. д. до тех пор, пока не будут посещены почти все страницы в интернете, которые используются для формирования индекса поиска.
- Порт (port) – число, в общем смысле обозначающее, с каким конкретным приложением вы связываетесь при создании сетевого соединения через сокет с сервером. Пример: веб-трафик обычно использует порт 80, трафик электронной почты – порт 25.
- Сокет (socket) – сетевое соединение между двумя приложениями, после установления которого каждое приложение может передавать и принимать данные в обоих направлениях.

12.11. Упражнения

УПРАЖНЕНИЕ 1 Изменить программу с использованием сокета *socket1.py* так, чтобы она запрашивала у пользователя URL и могла прочитать любую веб-страницу. Можно воспользоваться методом `split('/')` для разделения URL на отдельные части, чтобы извлечь имя хоста для передачи в вызов метода сокета `connect`. Добавить код проверки на ошибки с использованием инструкций `try` и `except` для обработки ситуации, в которой пользователь вводит некорректно сформированный или несуществующий URL.

УПРАЖНЕНИЕ 2 Изменить программу с использованием сокета так, чтобы она подсчитывала число принятых символов и останавливала отображение любого текста после вывода 3000 символов. Программа должна извлечь весь

документ и подсчитать общее число символов, затем вывести значение счетчика символов в конце документа.

УПРАЖНЕНИЕ 3 Использовать библиотеку `urllib` для повторного выполнения предыдущего упражнения: (1) извлечение документа по URL, (2) вывод не более 3000 символов, (3) подсчет общего числа символов в документе. В этом упражнении можно не беспокоиться об обработке заголовков, нужно просто вывести первые 3000 символов содержимого документа.

УПРАЖНЕНИЕ 4 Изменить программу `urllinks.py` так, чтобы она извлекала и подсчитывала число тегов абзаца (`p`) из полученного HTML-документа и выводила значение счетчика абзацев как результат ее работы. Текст абзаца выводить не нужно, только считать абзацы. Проверить программу сначала на небольших, затем на более крупных веб-страницах.

УПРАЖНЕНИЕ 5 (более сложное) Изменить программу с использованием сокета так, чтобы она выводила только данные после заголовков и пустую строку, которая была получена. Напомню, что метод `recv` принимает символы (перехода на новую строку и все прочие), но не строки.

Глава 13

Использование веб-сервисов

После ознакомления с простыми способами извлечения и синтаксического анализа документов по протоколу HTTP с использованием программ не требуется длительное время для перехода к разработке методики, с помощью которой мы могли бы начать создание документов, специально предназначенных для потребления другими программами (т. е. не кода HTML, выводимого в браузере).

Существуют два широко распространенных формата, используемых при обмене данными через веб-среду. XML (eXtensible Markup Language) применяется уже в течение весьма длительного времени и наилучшим образом подходит для обмена данными, отформатированными в стиле документа. Если программа должна обмениваться словарями, списками или прочей внутренней информацией с другой программой, то применяется формат JSON (JavaScript Object Notation) (см.: www.json.org). Мы рассмотрим оба формата.

13.1. XML – eXtensible Markup Language

XML выглядит очень похожим на HTML, но XML является более структурированным языком. Ниже приведен пример документа на языке XML:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

Каждая пара открывающих (например, `<person>`) и закрывающих тегов (например, `</person>`) представляет элемент (element) или узел (node) с тем же именем, что и тег (например, `person`). Каждый элемент может содержать некоторый текст, некоторые атрибуты (например, `hide`) и другие вложенные

элементы. Если элемент XML пуст (т. е. не имеет содержимого), то его можно обозначить самозакрывающимся тегом (например, `<email />`).

Часто удобно воспринимать XML-документ как древовидную структуру, в которой существует верхний элемент (в данном примере `person`) и прочие теги (например, `phone`), изображенные как дети своих родительских элементов.

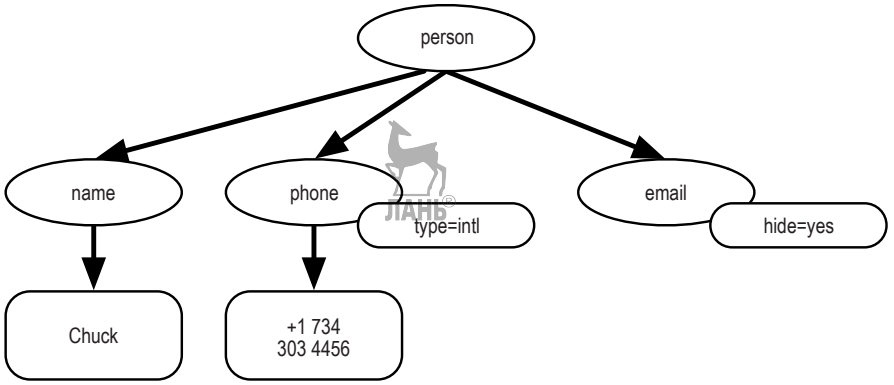


Рис. 13.1 ❖ Представление XML-документа в виде дерева

13.2. Синтаксический анализ XML

Ниже приведено простое приложение, которое выполняет синтаксический анализ некоторого кода XML и извлекает некоторые элементы данных из этого кода:

```
import xml.etree.ElementTree as ET

data = '''
<person>
<name>Chuck</name>
<phone type="intl">
+1 734 303 4456
</phone>
<email hide="yes" />
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Исходный код: http://www.py4e.com/code3/xml1.py
```

Тройные одиночные кавычки (`'''`), а также тройные двойные кавычки (`"""`) позволяют создавать строки текста, которые занимают несколько экранных строк.

При вызове метода fromstring выполняется преобразование представления в виде строк кода XML в «дерево» элементов XML. Когда код XML представлен в виде дерева, в наше распоряжение поступает группа методов, которые можно вызывать для извлечения фрагментов данных из XML-строки. Функция find выполняет поиск в XML-дереве и извлекает элемент, соответствующий заданному тегу.

```
Name: Chuck
Attr: yes
```

Использование парсера языка XML, такого как ElementTree, дает определенное преимущество, заключающееся в том, что хотя код XML в этом примере достаточно прост, тем не менее появляется возможность определить многие правила, касающиеся корректности кода XML. Кроме того, использование ElementTree позволяет извлекать данные из кода XML без беспокойства о соблюдении правил синтаксиса этого языка.

13.3. Проход в цикле по узлам

Часто в коде XML содержится множество узлов и необходимо написать цикл для обработки всех этих узлов. В показанной ниже программе мы в цикле проходим по всем узлам user:

```
import xml.etree.ElementTree as ET
input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get('x'))

# Исходный код: http://www.py4e.com/code3/xml2.py
```

Метод findall извлекает список Python, состоящий из деревьев, представляющих структуры user в дереве XML. Затем можно написать цикл for,

который просматривает каждый узел пользователя и выводит текстовые элементы name и id, а также атрибут x из узла user.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

Важно включить все элементы родительского уровня в инструкцию findall, за исключением элемента самого верхнего уровня (например, users/user). Иначе Python не найдет ни одного требуемого узла.

```
import xml.etree.ElementTree as ET
input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')
print('User count:', len(lst))

lst2 = stuff.findall('user')
print('User count:', len(lst2))
```

Список lst сохраняет все элементы user, вложенные в родительскую структуру users. В списке lst2 сохраняются найденные элементы user, которые не являются вложенными в элемент верхнего уровня stuff. Таких элементов здесь нет.

```
User count: 2
User count: 0
```

13.4. JSON – JavaScript Object Notation

Для формата JSON основным источником стал формат объекта и массива, используемый в языке JavaScript. Но поскольку Python был разработан раньше, чем JavaScript, синтаксис словарей и списков в Python повлиял на синтаксис

JSON. Поэтому формат языка JSON почти идентичен комбинации синтаксиса списков и словарей языка Python.

Ниже приведен пример применения JSON, кодирующий приблизительно ту же структуру, которая была показана в простом формате XML в предыдущем разделе:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
  },
  "email" : {
    "hide" : "yes"
  }
}
```


Вероятно, вы замечаете некоторые различия. Во-первых, в XML мы можем добавить атрибуты, например `intl` в тег `phone`. В JSON мы просто пишем пару ключ-значение. Также отсутствует тег XML `person`, его заменяет пара внешних фигурных скобок.

В общем случае структуры JSON проще, чем структуры XML, потому что в JSON меньше функциональных возможностей, чем в XML. Но JSON обладает важным преимуществом: его структуры напрямую отображаются в некоторую комбинацию словарей и списков. А поскольку почти все языки программирования поддерживают в той или иной степени структуры, равнозначные словарям и спискам Python, JSON представляет собой весьма естественный формат обмена данными между двумя сотрудничающими программами.

JSON быстро становится форматом, который выбирают почти для всех приложений, обменивающихся данными, из-за его относительной простоты по сравнению с форматом XML.

13.5. Синтаксический анализ формата JSON

Мы формируем документ в формате JSON, применяя по необходимости вложенные словари и списки. В приведенном ниже примере представлен список пользователей, где каждый пользователь – это множество (набор) пар ключ-значение (т. е. словарь). Поэтому получаем список словарей.

В приведенной ниже программе используется встроенная библиотека `json` для синтаксического анализа формата JSON и чтения данных. Тщательно сравним эту структуру данных с равнозначными данными XML и с исходным кодом, приведенным в предыдущем разделе. Код JSON содержит меньше подробностей, поэтому мы обязательно должны знать заранее, что получаем список, который сформирован из данных о пользователях, а каждый пользователь представлен набором пар ключ-значение. Формат JSON более лаконичен (это преимущество), но в то же время в меньшей степени описывает сам себя (это недостаток).

```
import json

data = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Исходный код: http://www.py4e.com/code3/json2.py
```

Если сравнить код для извлечения данных из форматов JSON и XML после синтаксического анализа, то вы заметите, что результат, полученный из `json.loads()`, – это список Python, по которому мы проходим с помощью цикла `for`, а каждым элементом этого списка является словарь Python. После завершения синтаксического анализа кода JSON можно использовать оператор индекса языка Python для извлечения различных фрагментов данных о каждом пользователе. Совсем не обязательно применять библиотеку поддержки JSON для глубокого исследования проанализированного формата JSON, потому что возвращаемые данные – это просто собственные структуры языка Python.

Вывод этой программы точно такой же, как и вывод версии с обработкой XML, показанной выше.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

Вообще говоря, в производственной сфере существует тенденция к отказу от формата XML в пользу формата JSON в применении к веб-сервисам. Поскольку JSON проще и более точно отображается в собственные структуры данных, уже имеющихся в языках программирования, синтаксический анализ и код извлечения данных обычно проще и понятнее при использовании JSON. Но XML в большей степени описывает сам себя, чем JSON, поэтому существует ряд приложений, в которых XML остается более предпочтитель-

ным форматом. Например, большинство текстовых процессоров сохраняют документы во внутреннем формате, использующем XML, а не JSON.

13.6. Программные интерфейсы приложений

Теперь у нас есть возможность обмениваться данными между приложениями, используя протокол HTTP (Hypertext Transfer Protocol), и способ представления данных сложной структуры, передаваемых в обоих направлениях между приложениями, используя формат XML (eXtensible Markup Language) или JSON (JavaScript Object Notation).

Следующим шагом является начало определения и документирования «контрактов» между приложениями с использованием этих технологий. Общее название для таких контрактов типа «приложение-приложение» – программные интерфейсы приложений (Application Program Interfaces – API). При использовании API в общем случае одна программа создает набор сервисов (services), доступных для других приложений, и публикует соответствующие API (т. е. «правила пользования»), которые непременно должны соблюдаться для доступа к сервисам, предоставляемым этой программой.

Когда мы начинаем создавать программы, функциональность которых включает доступ к сервисам, предоставляемым другими программами, то называем такой подход сервис-ориентированной архитектурой (Service-oriented architecture – SOA). Технология SOA позволяет всему приложению в целом использовать сервисы других приложений. При использовании методики без SOA приложение является автономным и самодостаточным и содержит весь код, необходимый для реализации его функциональности.

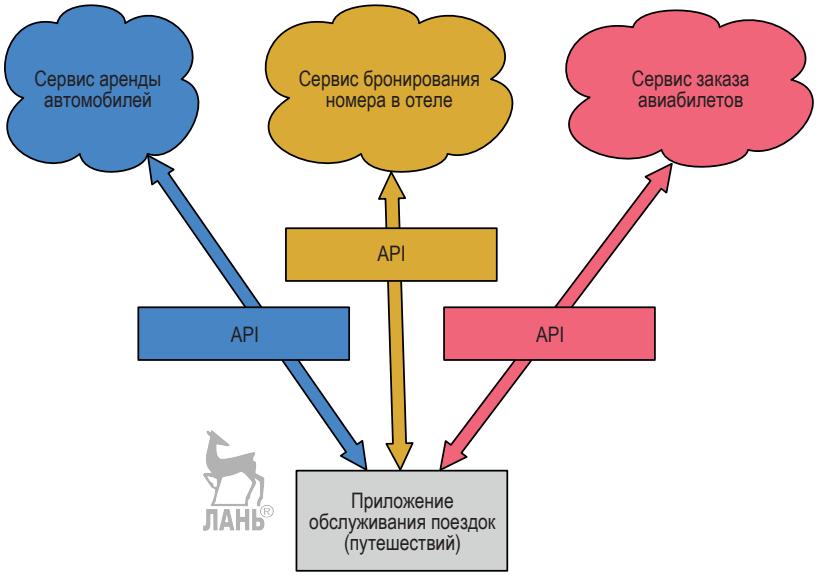


Рис. 13.2 ❖ Пример сервис-ориентированной архитектуры (SOA)

Пользуясь веб-средой, мы наблюдаем множество примеров применения SOA. Можно перейти на конкретный веб-сайт и заказать билеты на самолет, номер в отеле и арендовать автомобиль – и все это на единственном сайте. Данные о свободных номерах в отеле не хранятся в компьютерах авиакомпании. Просто компьютеры авиакомпании обращаются к сервисам компьютеров отелей и получают данные о свободных номерах и представляют их пользователю. Когда пользователь выбирает вариант бронирования номера в отеле, используя сайт авиакомпании, этот сайт обращается к другому сервису системы отеля, чтобы действительно выполнить бронирование номера. А когда приходит время расплатиться кредитной картой за всю транзакцию в целом, в этот процесс продолжают вовлекаться и другие компьютеры (сайты).

Сервис-ориентированная архитектура обладает многими преимуществами, в том числе:

1) мы всегда работаем только с одной копией данных (это особенно важно для случаев, подобных варианту бронирования номера в отеле, когда крайне нежелательны какие-либо повышенные обязательства);
2) владельцы данных могут устанавливать правила использования этих данных.

С учетом этих преимуществ любая SOA-система должна быть тщательно спроектирована с обеспечением высокой производительности и соответствия потребностям пользователя.

Когда в приложении создан набор сервисов и соответствующий API доступен в веб-среде, мы называем это веб-сервисами (web-services).

13.7. Безопасность и использование API

Часто возникает необходимость в получении API-ключа, чтобы воспользоваться программным интерфейсом, предлагаемым его владельцем. Общая идея заключается в том, что владельцы должны знать, кто использует их сервисы и в каком объеме. Возможно, имеются бесплатные и оплачиваемые звенья их сервисов, или существует стратегия, ограничивающая число запросов от одного лица за определенный интервал времени.

Иногда после получения API-ключа вы просто используете этот ключ как часть данных запроса POST или, вероятно, как параметр в URL при обращении к API.

В других случаях владелец API требует обеспечения более высокой степени безопасности запросов, следовательно, ожидает, что вы передаете сообщения с криптографической подписью с использованием пары ключей – открытого и секретного. Весьма часто применяется технология с применением запроса подписи через интернет, называемая OAuth. Более подробно о протоколе OAuth можно узнать здесь: www.oauth.net.

К счастью, существует несколько удобных и бесплатных библиотек поддержки OAuth, так что можно избежать реализации этого протокола с нуля с изучением его спецификации. Библиотеки обладают различными степеня-

ми сложности и полнофункциональности, а более подробную информацию о них вы найдете на сайте OAuth.

13.8. Словарь терминов

- API – Application Program Interface – программный интерфейс приложения, контракт между приложениями, который определяет шаблоны взаимодействия между компонентами двух приложений.
- ElementTree – встроенная библиотека Python, используемая для синтаксического анализа данных в формате XML.
- JSON – JavaScript Object Notation – формат, позволяющий размечать структурированные данные на основе синтаксиса JavaScript Objects (объектов JavaScript).
- SOA – Service-Oriented Architecture – сервис-ориентированная архитектура, позволяет приложению использовать компоненты, соединяемые по сети.
- XML – eXtensible Markup Language – формат, позволяющий размечать структурированные данные.

13.9. Приложение 1: веб-сервис геокодирования Google

Компания Google предоставляет превосходный веб-сервис, позволяющий воспользоваться ее крупной базой данных географической информации. Можно передать строку для географического поиска, например «Ann Arbor, MI» в API геокодирования и получить от Google возврат наилучшего предположения о том, в каком месте карты можно найти место, соответствующее заданной строке поиска. Кроме того, сообщается о расположенных поблизости общеизвестных ориентирах.

Сервис геокодирования бесплатный, но скорость ограничена, поэтому вы не сможете использовать этот API в коммерческом приложении. Но если имеются некоторые данные опроса, где конечный пользователь ввел некоторую локацию в окне ввода в свободном формате, то можно воспользоваться API геолокации для максимального уточнения введенных данных.

При использовании бесплатного API, такого как API геолокации Google, необходимо бережно и аккуратно относиться к предоставляемым ресурсам. Если слишком много пользователей бездумно эксплуатируют сервис, то Google может отключить его или существенно сократить возможность его бесплатного использования.

Вы можете прочитать онлайновую документацию по этому сервису, но его практическое использование осуществляется достаточно просто, и вы можете даже протестировать его с помощью обычного браузера, если введете следующий URL в строке адреса:

http://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI.

Убедитесь, что запись URL не имеет разрывов и переносов, и удалите все пробелы из этой строки URL перед копированием ее в панель адреса вашего браузера.

Ниже приведено простое приложение, которое предлагает пользователю ввести строку поиска, вызывает API геокодирования Google и извлекает информацию из возвращаемого кода JSON.

```
import urllib.request, urllib.parse, urllib.error
import json
import ssl

api_key = False
# Если у вас есть API-ключ Google Places, то введите его здесь:
# api_key = 'AIzaSy___IDByT70'
# https://developers.google.com/maps/documentation/geocoding/intro

if api_key is False:
    api_key = 42
    serviceurl = 'http://py4e-data.dr-chuck.net/json?'
else :
    serviceurl = 'https://maps.googleapis.com/maps/api/geocode/json?'

# Игнорировать ошибки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    address = input('Enter location: ')
    if len(address) < 1: break

    parms = dict()
    parms['address'] = address
    if api_key is not False: parms['key'] = api_key
    url = serviceurl + urllib.parse.urlencode(parms)

    print('Retrieving', url)
    uh = urllib.request.urlopen(url, context=ctx)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')

    try:
        js = json.loads(data)
    except:
        js = None

    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue

    print(json.dumps(js, indent=4))
```

```
    lat = js['results'][0]['geometry']['location']['lat']
    lng = js['results'][0]['geometry']['location']['lng']
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)

# Исходный код: http://www.py4e.com/code3/geojson.py
```

Программа принимает строку поиска и формирует URL с этой строкой поиска как с корректно закодированным параметром, затем использует библиотеку `urllib` для получения текста от API геокодирования Google. В отличие от фиксированной веб-страницы, получаемые данные зависят от переданных параметров и географических данных, хранящихся на серверах Google.

После получения данных в формате JSON выполняется их синтаксический анализ с помощью библиотеки `json` и несколько проверок для уверенности в том, что получены правильные данные, затем извлекается требуемая информация.

Вывод программы показан ниже (некоторые фрагменты возвращенного кода JSON удалены для экономии места):

```
$ python3 geojson.py
Enter location: Ann Arbor, MI
Retrieving http://py4e-data.dr-chuck.net/json?address=Ann+Arbor%2C+MI&key=42
Retrieved 1736 characters

{
    "results": [
        {
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "short_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ]
                },
                {
                    "long_name": "Washtenaw County",
                    "short_name": "Washtenaw County",
                    "types": [
                        "administrative_area_level_2",
                        "political"
                    ]
                },
                {
                    "long_name": "Michigan",
                    "short_name": "MI",
                    "types": [
                        "administrative_area_level_1",
                        "political"
```

```
                    ]
                },
                {
                    "long_name": "United States",
                    "short_name": "US",
                    "types": [
                        "country",
                        "political"
                    ]
                }
            ],
            "formatted_address": "Ann Arbor, MI, USA",
            "geometry": {
                "bounds": {
                    "northeast": {
                        "lat": 42.3239728,
                        "lng": -83.6758069
                    },
                    "southwest": {
                        "lat": 42.222668,
                        "lng": -83.799572
                    }
                },
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                },
                "location_type": "APPROXIMATE",
                "viewport": {
                    "northeast": {
                        "lat": 42.3239728,
                        "lng": -83.6758069
                    },
                    "southwest": {
                        "lat": 42.222668,
                        "lng": -83.799572
                    }
                }
            },
            "place_id": "ChIJMx9D1A2wPIgR4rXIhkb5Cds",
            "types": [
                "locality",
                "political"
            ]
        }
    ],
    "status": "OK"
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA

Enter location:
```

Версию программы для обработки варианта с кодом XML, полученного от API геолокации Google, можно загрузить здесь: www.py4e.com/code3/geoxml.py.

УПРАЖНЕНИЕ 1 Изменить программу *geojson.py* или *geoxml.py* для вывода двухсимвольного кода страны, извлеченного из полученных данных. Добавить проверку на ошибки, чтобы программа не выполняла обратную трассировку, если код страны отсутствует. Когда программа начнет работать правильно, выполнить поиск по строке «Atlantic Ocean», чтобы убедиться в том, что она корректно обрабатывает не только локации, являющиеся странами.

13.10. Приложение 2: Twitter

После того как значимость API Twitter стала постоянно повышаться, компания Twitter перешла от открытого и общедоступного к API, который требует использования подписей OAuth для каждого запроса API.

Для работы с этой программой необходимо загрузить файлы *twurl.py*, *hidden.py*, *oauth.py* и *twitter1.py* отсюда: www.py4e.com/code – и поместить их в отдельный каталог на вашем компьютере.

Чтобы воспользоваться этими программами, потребуется аккаунт в Twitter и авторизация кода Python как приложения, установка открытого ключа, секретного ключа, токена и секретного токена. Необходимо отредактировать файл *hidden.py* и поместить эти четыре строки в соответствующие переменные в данном файле:

```
# Сохраните этот файл отдельно.

# https://apps.twitter.com/
# Создать новое приложение и передать в него эти четыре строки.

def oauth():
    return {"consumer_key": "h7Lu...Ng",
            "consumer_secret" : "dNKenAC3New...mmn7Q",
            "token_key" : "10185562-eibxCp9n2...P4GEQQOSGI",
            "token_secret" : "H0ycCFemmC4wyf1...qoIpBo"}

# Исходный код: http://www.py4e.com/code3/hidden.py
```

Веб-сервис Twitter доступен при использовании URL, подобного показанному ниже:

https://api.twitter.com/1.1/statuses/user_timeline.json.

Но после добавления информации о защите (безопасности) этот URL будет выглядеть больше похожим на настоящий:

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

Вы можете изучить спецификацию OAuth, если хотите узнать больше о смысле различных параметров, добавляемых для соответствия требованиям безопасности по протоколу OAuth.

Для программ, работающих с Twitter, мы скрываем всю сложность в файлах *oauth.py* и *twurl.py*. Мы просто устанавливаем секретные параметры в файле *hidden.py*, затем отправляем требуемый URL в функцию `twurl.augment()`, а библиотечный код добавляет все необходимые параметры в этот URL автоматически.

Эта программа получает строку хронометража для конкретного пользователя Twitter и возвращает ее в строке формата JSON. Мы просто выводим первые 250 символов этой строки:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import ssl

# https://apps.twitter.com/
# Создать приложение и передать в него четыре строки, поместив их в файл hidden.py.

TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'

# Игнорировать ошибки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    print(data[:250])
    headers = dict(connection.getheaders())
    # Вывод заголовков.
    print('Remaining', headers['x-rate-limit-remaining'])

# Исходный код: http://www.py4e.com/code3/twitter1.py
```

После запуска эта программа выводит следующую информацию:

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178
```

```
Enter Twitter Account:fixpert

Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

Enter Twitter Account:
```

Вместе с данными хронометража Twitter также возвращает метаданные о запросе в заголовках ответа HTTP. В частности, один из заголовков `x-rate-limit-remaining` информирует нас, сколько еще запросов мы можем выполнить, прежде чем соединение закроется на короткий интервал времени. Можно видеть, что это число постоянно уменьшается на единицу при каждом нашем запросе к API.

В приведенном ниже примере мы получаем имена друзей пользователя Twitter, выполняем синтаксический анализ возвращаемого кода JSON и извлекаем некоторую информацию о друзьях. Также выводится все содержимое кода JSON после синтаксического анализа и «удобного» форматирования с применением отступа в четыре символа, позволяющего комфортно просматривать данные, если потребуется извлечь другие поля.

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import ssl

# https://apps.twitter.com/
# Создать приложение и передать в него четыре строки, поместив их в файл hidden.py.

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

# Игнорировать ошибки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()

    js = json.loads(data)
    print(json.dumps(js, indent=2))
```

```
    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])
    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
            print(' * No status found')
            continue
        s = u['status']['text']
        print(' ', s[:50])

# Исходный код: http://www.py4e.com/code3/twitter2.py
```

Поскольку код JSON становится набором списков и словарей языка Python, мы можем использовать комбинацию операций с индексами и циклы `for` для прохода по возвращенным структурам данных, при этом требуется очень малый объем кода на Python.

Вывод этой программы показан ниже (некоторые элементы данных сокращены для экономии места на странице):

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14

{
  "next_cursor": 1444171224491980205,
  "users": [
    {
      "id": 662433,
      "followers_count": 28725,
      "status": {
        "text": "@jazzychad I just bought one .__.",
        "created_at": "Fri Sep 20 08:36:34 +0000 2013",
        "retweeted": false,
      },
      "location": "San Francisco, California",
      "screen_name": "leahculver",
      "name": "Leah Culver",
    },
    {
      "id": 40426722,
      "followers_count": 2635,
      "status": {
        "text": "RT @WSJ: Big employers like Google ...",
        "created_at": "Sat Sep 28 19:36:37 +0000 2013",
      },
      "location": "Victoria Canada",
      "screen_name": "_valeriei",
      "name": "Valerie Irvine",
    }
  ],
  "next_cursor_str": "1444171224491980205"
}
```

```
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

В последнем фрагменте выводимой информации можно видеть, что цикл `for` считывает пять самых последних «друзей» пользователя Twitter с аккаунтом @drchuck и выводит самое последнее состояние каждого друга. В возвращенном коде JSON доступно гораздо больше данных. Если внимательно посмотреть на вывод этой программы, то также можно заметить, что «поиск друзей» конкретного аккаунта имеет ограничение, отличающееся от числа запросов хронометража, разрешенного в течение определенного интервала времени.

Такие защитные API-ключи позволяют Twitter иметь прочную уверенность в том, что его владельцы знают, кто использует их API и данные и на каком уровне. Методика ограничения числа запросов позволяет выполнять простые операции по извлечению личных данных, но не допускает создания программного продукта, извлекающего данные с помощью API Twitter миллионы раз в день.

Глава 14

Объектно-ориентированное программирование

14.1. Управление более крупными программами

В начале этой книги мы сформулировали четыре основных шаблона программирования, которые использовали для создания программ:

- последовательный код;
- условный код (инструкции if);
- повторяющийся код (циклы);
- сохранение и многократное использование (функции).

В последующих главах мы применяли простые переменные, а также наборы структур данных, такие как списки, кортежи и словари.

При создании программ мы проектируем структуры данных и пишем код для обработки этих структур. Существует много способов написания программ, но к настоящему моменту вы, вероятно, уже написали некоторые программы, которые «не слишком изящны», и другие программы, которые «более изящны». Даже если ваши программы имеют небольшой размер, вы начинаете понемногу видеть и понимать искусство и эстетику в написании исходного кода.

Когда программы разрастаются до нескольких миллионов строк, постепенно становится все более важным написание кода, который легко понять. Если вы работаете над программой с размером в миллион строк, то никогда не сможете одновременно удержать в памяти содержимое всей программы в целом. Поэтому необходим способ разделения больших программ на множество более мелких фрагментов, чтобы основное внимание сконцентрировать на решении задачи, исправлении ошибок или добавлении новых функциональных возможностей.

В определенной степени объектно-ориентированное программирование представляет собой способ такой организации исходного кода, которая позволяет сосредоточиться на 50 строках кода и понять их, при этом не обращая внимания на остальные 999 950 строк, в данный момент не имеющих значения.

14.2. Приступим

Как и для многих аспектов программирования, необходимо изучить концепции объектно-ориентированного программирования, прежде чем вы сможете эффективно его использовать. Вы должны подойти к чтению этой главы как к изучению некоторых терминов и концепций, а также поработать с несколькими простыми примерами, чтобы заложить основу для будущего обучения.

Главный результат изучения этой главы – базовое понимание того, как создаются объекты, как они функционируют и, самое важное, как мы используем функциональные возможности объектов, которые предоставляют нам язык и библиотеки Python.

14.3. Использование объектов

Оказывается, мы уже использовали объекты на протяжении всей книги. Python предоставляет множество встроенных объектов. Ниже показан простой код, в котором несколько первых строк должны выглядеть весьма простыми и естественными для вас.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Исходный код: http://www.py4e.com/code3/party1.py
```

Не будем отвлекаться на то, что именно делают эти строки, вместо этого рассмотрим, что происходит в действительности с точки зрения объектно-ориентированного программирования. Не беспокойтесь, если следующие абзацы покажутся вам не совсем понятными при первом прочтении, потому что мы пока еще не определили все необходимые термины.

Первая строка конструирует (создает) объект типа `list` (список), вторая и третья – вызывают метод `append()`. В четвертой строке вызывается метод sort(), пятая извлекает элемент в позиции 0.

В шестой строке вызывается метод `__getitem__()` в списке `stuff` с параметром ноль.

```
print (stuff.__getitem__(0))
```

В седьмой строке показан еще более детализированный способ извлечения нулевого элемента из списка `stuff`.

```
print (list.__getitem__(stuff,0))
```

В этом коде мы вызываем метод `__getitem__` из класса `list` и передаем в него конкретный список и элемент, который нужно извлечь из этого списка, как параметры.

Последние три строки в приведенном выше примере равнозначны, но более удобно просто использовать синтаксис квадратных скобок для поиска элемента в конкретной позиции в списке.

Можно узнать о функциональных возможностях объекта, просмотрев вывод функции `dir()`:


```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

В остальной части текущей главы будут определены все термины, использованные выше, поэтому после завершения ее чтения рекомендую вернуться и еще раз прочитать эти абзацы, чтобы окончательно убедиться в правильном понимании терминов.

14.4. Начнем с программ

Программа – это самая простая форма, принимающая некоторый ввод, выполняющая определенную обработку и генерирующая некоторый вывод. Показанная ниже программа преобразования счетчика этажей лифта демонстрирует весьма короткий, но полный пример выполнения всех трех вышеперечисленных шагов.

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)

# Исходный код: http://www.py4e.com/code3/elev.py
```

Если подумать немного глубже об этой программе, то можно понять, что существует «внешний мир» и сама программа. Ввод и вывод – это части, в которых программа взаимодействует с внешним миром. Внутри находятся код и данные для выполнения задачи, которую должна решить эта программа.

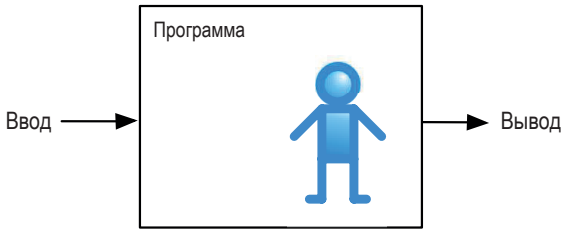


Рис. 14.1 ❖ Программа

Один из способов осмысления сущности объектно-ориентированного программирования – понимание того, что оно разделяет программу на несколько «зон». Каждая зона содержит некоторый код и данные (как и программа) и имеет четко определенные взаимодействия с внешним миром и другими зонами внутри программы.

Если вернуться к приложению, извлекающему ссылки, в котором использовалась библиотека BeautifulSoup, то можно увидеть программу, скомпонованную посредством соединения различных объектов для выполнения поставленной задачи:

```
# Для запуска этой программы необходимо скачать zip-файл BeautifulSoup здесь:
# http://www.py4e.com/code3/bs4.zip
# и разархивировать его в том каталоге, где находится файл этой программы.

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Игнорировать ошибки проверки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Извлечение всего содержимого из тегов анкера.
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Исходный код: http://www.py4e.com/code3/urllinks.py
```

Мы считываем URL в строку, затем передаем эту строку в библиотеку `urllib` для извлечения данных из веб-среды. Библиотека `urllib` использует библиотеку `socket` для создания реальных сетевых соединений с целью получения данных. Мы берем строку, возвращаемую `urllib`, и передаем ее в BeautifulSoup для синтаксического анализа. BeautifulSoup использует объект html.parser[1] и возвращает объект. Мы вызываем метод `tags()` полученного объекта, который возвращает словарь объектов-тегов. Мы в цикле проходим по тегам и для каждого тега вызываем метод `get()` для вывода атрибута `href`.

Можно нарисовать графическую схему этой программы, на которой показано, как все объекты работают вместе.

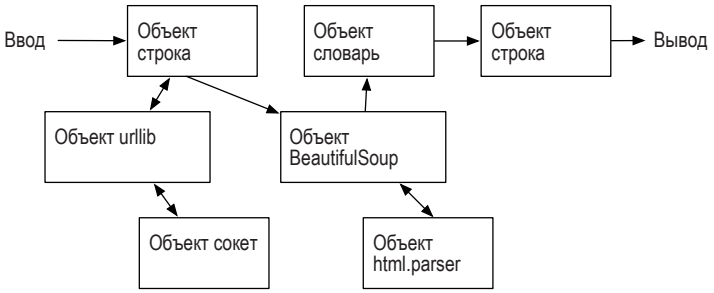


Рис. 14.2 ❖ Программа как сеть объектов

Полное понимание того, как работает эта программа, здесь не самое главное – гораздо важнее увидеть, как выстраивается сеть взаимодействующих объектов и организуется (в современной терминологии – «выполняется оркестровка») перемещение информации между этими объектами для создания программы. Кроме того, важно отметить, что при рассмотрении этой программы в главе 12 вы могли полностью понимать, что в ней происходит, даже не имея представления о том, что программа «выполняла оркестровку перемещения данных между объектами». Это были просто строки исходного кода, которые выполняли поставленную задачу.

14.5. Разделение задачи на подзадачи

Одним из преимуществ объектно-ориентированного подхода является возможность скрытия сложности. Например, хотя мы должны знать, как использовать код библиотек `urllib` и BeautifulSoup, нет никакой необходимости в понимании внутренней работы этих библиотек. Это позволяет сосредоточиться на той части программы, которая непосредственно решает поставленную задачу, и не обращать внимания на прочие части программы.

[1] https://docs.python.org/3/library/html.parser.html.

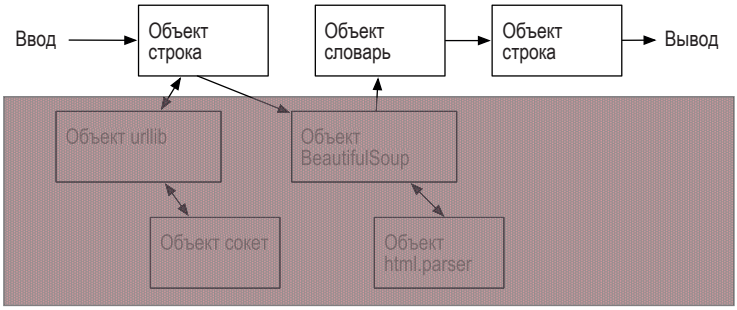


Рис. 14.3 ❖ Игнорирование подробностей при использовании объектов

Кроме того, возможность сосредоточиться исключительно на интересующей нас части программы и игнорирование прочих частей удобны и для разработчиков объектов, которые мы используем. Например, программисты, разрабатывающие библиотеку BeautifulSoup, не обязаны знать или как-либо заботиться о том, как мы получаем HTML-страницу, какую ее часть нужно прочитать или что мы планируем делать с данными, извлеченными из этой страницы.

14.6. Наш первый объект Python

На простейшем (базовом) уровне объект – это просто некоторый код плюс структуры данных, и все это меньше размера всей программы в целом. Определение функции позволяет сохранить небольшой фрагмент кода, дать ему имя, а затем вызывать этот код, используя имя функции.

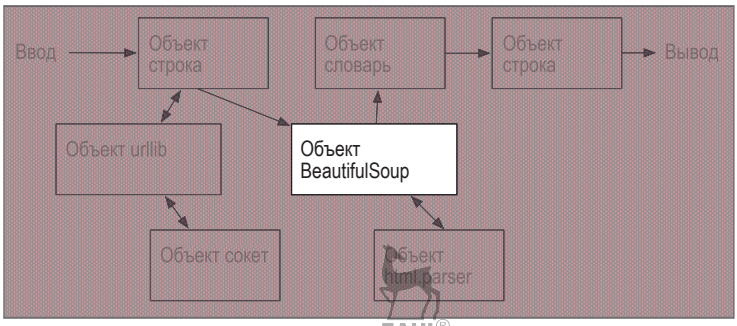


Рис. 14.4 ❖ Игнорирование подробностей при создании объекта

Объект может содержать несколько функций (которые мы называем методами), а также данные, используемые этими функциями. Элементы данных, являющиеся частями объекта, называются атрибутами (attributes).

Мы используем ключевое слово `class` для определения кода и данных, которые формируют каждый конкретный объект. За ключевым словом `class`

должно следовать имя класса, а далее начинается сдвинутый вправо блок кода, содержащий атрибуты (данные) и методы (код).

```
class PartyAnimal:
    x = 0

    def party(self) :
        self.x = self.x + 1
        print("So far",self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Исходный код: http://www.py4e.com/code3/party2.py
```

Каждый метод выглядит как функция, начинается с ключевого слова `def` и состоит из сдвинутого вправо блока кода. Этот объект содержит один атрибут (`x`) и один метод (`party`). Все методы имеют специальный первый параметр, которому по соглашению присвоено имя `self`.

Ключевое слово `def` не приводит к выполнению кода функции, точно так же и ключевое слово `class` не создает объект немедленно. Вместо этого ключевое слово `class` определяет шаблон, сообщающий, что записанные в нем данные и код будут содержаться в каждом объекте типа `PartyAnimal`. В этом смысле класс похож на форму для печенья, а создаваемые с его использованием объекты – это собственно печенье[1]. Вы не кладете глазурь в форму, а наносите ее на печенье и можете украшать разными типами глазури каждое отдельное печенье.

Рис. 14.5 ❖ Класс и два объекта

[1] Изображение печенья, защищенное лицензией CC-BY: https://www.flickr.com/photos/dinnerseries/23570475099.

Если продолжить разбор приведенного выше примера программы, то мы увидим первую выполняемую строку кода:

```
an = PartyAnimal()
```

Именно здесь мы сообщаем Python о необходимости сконструировать (т. е. создать) объект (object) или экземпляр (instance) класса `PartyAnimal`. Это выглядит как вызов функции с именем самого класса. Python конструирует (создает) затребованный объект с корректными данными и методами и возвращает созданный объект, который затем присваивается переменной `an`. В некотором смысле это почти похоже на следующую строку, которой мы пользовались постоянно:

```
counts = dict()
```

Здесь мы сообщаем Python о необходимости конструирования объекта с использованием шаблона `dict` (уже существующего в языке Python) с последующим возвратом созданного экземпляра словаря и присваиванием его переменной `counts`.

Если класс `PartyAnimal` используется для конструирования объекта, то переменная `an` применяется для указания (ссылки) на этот объект. Мы используем `an` для доступа к коду и данным этого конкретного экземпляра класса `PartyAnimal`.

Каждый объект/экземпляр класса `PartyAnimal` внутри содержит переменную `x` и метод/функцию с именем `party`. Мы вызываем метод `party` в следующей строке:

```
an.party()
```

При вызове метода `party` первый параметр (который по соглашению имеет имя `self`) указывает на конкретный экземпляр (объект) класса `PartyAnimal`, из которого вызван метод `party`. Внутри метода `party` мы видим строку кода:

```
self.x = self.x + 1
```

Этот синтаксис использует оператор точка, означающий «переменная `x` в объекте `self` (т. е. в самом себе)». При каждом вызове `party()` значение внутренней переменной `x` увеличивается на 1, и полученное в результате значение выводится.

В следующей строке продемонстрирован другой способ вызова метода `party` из объекта `an`:

```
PartyAnimal.party(an)
```

В этом варианте мы получаем доступ к коду из класса и явно передаем указатель на объект `an` как первый параметр (т. е. `self` внутри метода). Можно считать `an.party()` сокращенной записью приведенной выше строки.

При выполнении эта программа выводит следующую информацию:

```
So far 1
So far 2
```

```
So far 3
So far 4
```

Объект конструируется (создается), затем метод `party` вызывается четыре раза с инкрементированием и выводом полученного значения `x` внутри объекта `an`.

14.7. Классы как типы

Как нам уже известно, в Python все переменные имеют тип. Можно воспользоваться встроенной функцией `dir` для изучения функциональных свойств любой переменной. Также можно использовать функции `type` и `dir` для классов, которые мы сами создали.

```
class PartyAnimal:
    x = 0

    def party(self) :
        self.x = self.x + 1
        print("So far",self.x)

an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
print ("Type", type(an.party))

# Исходный код: http://www.py4e.com/code3/party3.py
```

Вывод этой программы показан ниже:

```
Type <class '__main__.PartyAnimal'>
Dir ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

Здесь можно видеть, что при использовании ключевого слова `class` мы создали новый тип. По информации, выводимой функцией `dir`, можно определить, что целочисленный атрибут `x` и метод `party` доступны в объекте этого типа.

14.8. Жизненный цикл объекта

В предыдущих примерах мы определили класс (шаблон), использовали его для создания экземпляра (объекта) этого класса, затем использовали созданный экземпляр. После завершения программы все переменные исчезают.

Обычно мы не задумываемся глубоко над созданием и уничтожением переменных, но когда наши объекты становятся более сложными, то чаще всего возникает необходимость в совершении некоторых действий внутри объекта для определенной настройки при его создании и, возможно, для очистки при его удалении.

Если необходимо, чтобы объект был оповещен о моментах создания (конструирования) и удаления (деструкции), мы добавляем методы со специальными именами в класс такого объекта:

```
class PartyAnimal:
    x = 0

    def __init__(self):
        print('I am constructed')

    def party(self) :
        self.x = self.x + 1
        print('So far',self.x)

    def __del__(self):
        print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)

# Исходный код: http://www.py4e.com/code3/party4.py
```

При выполнении программа выводит следующую информацию:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

Когда Python конструирует (создает) заданный объект, он вызывает указанный метод `__init__()`, чтобы предоставить нам возможность настроить некоторые значения по умолчанию или начальные значения для этого объекта. Когда Python обнаруживает строку:

```
an = 42
```

то в действительности «выбрасывает наш объект безвозвратно», чтобы можно было повторно использовать переменную `an` для хранения значения `42`. Именно в тот момент, когда наш объект `an` «уничтожается», вызывается код деструктора (`__del__`). Мы не можем защитить свою переменную от уничтожения, но получаем возможность корректно выполнить некоторые необходимые операции очистки, перед тем как наш объект окончательно прекратит свое существование.

При разработке объектов обычным, часто применяемым практическим приемом является добавление конструктора (constructor) в объект для настройки начальных значений его атрибутов. Деструктор для объекта требуется относительно редко.

14.9. Несколько экземпляров

К настоящему моменту мы определили класс, сконструировали (создали) один объект, использовали этот объект, а затем просто «выбросили» его. Но настоящая мощь объектно-ориентированного программирования проявляется, когда мы создаем несколько экземпляров своего класса.

Когда мы конструируем (создаем) несколько объектов нашего класса, возможно, потребуется установка различных начальных значений для каждого такого объекта. Данные можно передавать в конструкторы для присваивания каждому объекту начального значения, отличающегося от прочих:

```
class PartyAnimal:
    x = 0
    name = ''

    def __init__(self, nam):
        self.name = nam
        print(self.name,'constructed')

    def party(self) :
        self.x = self.x + 1
        print(self.name,'party count', self.x)

s = PartyAnimal('Sally')
j = PartyAnimal('Jim')

s.party()
j.party()
s.party()

# Исходный код: http://www.py4e.com/code3/party5.py
```

Конструктор принимает параметр `self`, указывающий на конкретный экземпляр объекта, и дополнительные параметры, которые передаются в него, когда объект создается (конструируется):

```
s = PartyAnimal('Sally')
```

Внутри конструктора вторая строка копирует параметр (`nam`), который передается в атрибут name в создаваемом экземпляре объекта.

```
self.name = nam
```

Вывод этой программы показывает, что каждый из созданных объектов (s и j) содержит собственные независимые копии `x` и `nam`:

```
Sally constructed
Jim constructed
Sally party count 1
Jim party count 1
Sally party count 2
```

14.10. Наследование

Еще одной мощной функциональной характеристикой объектно-ориентированного программирования является возможность создания нового класса с помощью расширения существующего класса. При расширении класса мы называем исходный класс родительским (parent class), а новый – классом-потомком (child class).

В приведенном ниже примере мы помещаем класс `PartyAnimal` в собственный отдельный файл. Затем можно «импортировать» класс `PartyAnimal` в новый файл и расширить его, как показано ниже:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):
    points = 0

    def six(self):
        self.points = self.points + 6
        self.party()
        print(self.name,"points", self.points)

s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))

# Исходный код: http://www.py4e.com/code3/party6.py
```

При определении класса `CricketFan` мы сообщаем, что выполняется расширение класса `PartyAnimal`. Это означает, что все переменные (`x`) и методы (`party`) из класса `PartyAnimal` наследуются (inherited) классом `CricketFan`. Например, внутри метода `six` в классе `CricketFan` вызывается метод `party` из класса `PartyAnimal`.

При выполнении программы создаются `s` и `j` как независимые экземпляры классов `PartyAnimal` и `CricketFan`. Объект `j` обладает дополнительными свойствами по сравнению с объектом `s`.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
```

```
Jim points 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

В выводе функции dir для объекта j (экземпляра класса CricketFan) можно видеть, что он содержит атрибуты и методы родительского класса, а также атрибуты и методы, добавленные при расширении для создания нового класса CricketFan.

14.11. Резюме

Это очень краткое введение в объектно-ориентированное программирование, сосредоточенное главным образом на объяснении терминологии и синтаксиса определения и использования объектов. Теперь еще раз рассмотрим код, приведенный в начале этой главы. Сейчас вы должны полностью понимать, что в нем происходит.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Исходный код: http://www.py4e.com/code3/party1.py
```

В первой строке создается (конструируется) объект типа список (list). Когда Python создает объект типа list, он вызывает метод-конструктор (constructor) (с именем __init__) для установки внутренних данных атрибутов, которые будут использоваться для хранения данных списка. В конструктор не передаются какие-либо параметры. После возврата из конструктора мы используем переменную stuff для указания на возвращаемый экземпляр класса list.

Во второй и третьей строках вызывается метод append с одним параметром для добавления нового элемента в конец списка с помощью обновления атрибутов в объекте stuff. Затем в четвертой строке вызывается метод sort без параметров для сортировки данных внутри объекта stuff.

Далее выводится первый элемент списка с использованием квадратных скобок, которые представляют собой сокращенную запись вызова метода __getitem__ из объекта stuff. Это равнозначно вызову метода __getitem__ из класса list с передачей объекта stuff как первого параметра и позиции требуемого элемента как второго параметра.

В конце программы объект stuff удаляется, но непосредственно перед этим вызывается деструктор (destructor) (с именем __del__), чтобы объект мог подчистить свои остающиеся следы, если это необходимо.

Таковы основы объектно-ориентированного программирования. Существует множество дополнительных подробностей, описывающих, как наилуч-

шим образом применять объектно-ориентированные методики при разработке крупных приложений и библиотек, но это уже выходит за рамки данной главы[1].

14.12. Словарь терминов

- Атрибут (attribute) – переменная, являющаяся частью класса.
- Деструктор (destructor) – необязательный метод со специальным именем (`__del__`), вызываемый непосредственно перед удалением объекта. Деструкторы используются редко.
- Класс (class) – шаблон, который можно использовать для создания (конструирования) объекта. Определяет атрибуты и методы, формирующие объект.
- Класс-потомок (child class) – новый класс, создаваемый при помощи расширения родительского класса. Класс-потомок наследует все атрибуты и методы родительского класса.
- Конструктор (constructor) – необязательный метод со специальным именем (`__init__`), вызываемый в тот момент, когда класс используется для создания (конструирования) объекта. Обычно конструктор применяется для установки начальных значений объекта.
- Метод (method) – функция, которая содержится внутри класса и объектов, конструируемых из этого класса. Для описания этой концепции некоторые шаблоны объектно-ориентированного программирования используют термин «сообщение» (message) вместо термина «метод».
- Наследование (inheritance) – выполняется при создании нового класса (потомка) при помощи расширения существующего класса (родителя). Класс-потомок содержит все атрибуты и методы родительского класса плюс дополнительные атрибуты и методы, определяемые в самом классе-потомке.
- Объект (object) – созданный экземпляр какого-либо класса. Объект содержит все атрибуты и методы, которые были определены в классе. В некоторой документации по объектно-ориентированному программированию используется термин «экземпляр» (instance) как равнозначный термину «объект».
- Родительский класс (parent class) – класс, который расширяется для создания нового класса-потомка. Родительский класс передает (позволяет наследовать) все свои методы и атрибуты в новый класс-потомок.

[1] Если вам интересно, где именно определяется класс `list`, то посмотрите здесь (надеюсь, что URL не изменился): https://github.com/python/cpython/blob/master/Objects/listobject.c – класс списка написан на языке программирования C. Если вы разобрались в этом исходном коде и нашли его интересным, то, вероятно, вам следует заняться обучением по нескольким курсам информационных технологий.

Глава 15

Использование баз данных и SQL

15.1. Что такое база данных

База данных (database) – это файл, специально организованный для хранения данных. Организация большинства баз данных похожа на словарь в том смысле, что они отображают ключи в значения. Самое большое различие заключается в том, что база данных размещена на диске (или на каком-либо другом устройстве постоянного хранения), поэтому продолжает существовать после завершения программы. Поскольку база данных расположена на устройстве постоянного хранения, она может содержать гораздо больше данных, чем словарь, размер которого ограничен объемом основной памяти компьютера.

Как и для словаря, программное обеспечение базы данных предназначено для поддержки чрезвычайно быстрых операций вставки и доступа, даже для весьма больших объемов данных. Программное обеспечение базы данных поддерживает высокую производительность, создавая индексы (indexes) как данные, которые добавляются в базу данных, чтобы обеспечить быстрый переход к конкретной записи.

Существует много различных систем управления базами данных, используемых для самых разнообразных целей: Oracle, MySQL, Microsoft SQL Server, PostgreSQL и SQLite. В этой книге мы сосредоточим внимание на SQLite, потому что это весьма часто применяемая на практике система управления базами данных, а кроме того, ее поддержка уже встроена в Python. SQLite предназначена для встраивания (embed) в другие приложения для обеспечения поддержки базы данных непосредственно внутри конкретного приложения. Например, в браузере Firefox также используется база данных SQLite, как и во многих других программных продуктах.

http://sqlite.org.

SQLite хорошо подходит для решения некоторых задач обработки данных, встречающихся в области информационных технологий, таких как приложение глобального поиска в Twitter, которое рассматривается в этой главе.

15.2. Концепции базы данных

При первом взгляде на базу данных она выглядит как электронная таблица с несколькими листами. Основными структурами в базе данных являются таблицы (tables), строки (rows) и столбцы (columns).

В технических описаниях реляционных баз данных концепции таблицы, строки и столбца более формально обозначаются как отношение (relation), кортеж (tuple) и атрибут (attribute) соответственно. В этой главе мы будем использовать менее формальные термины.

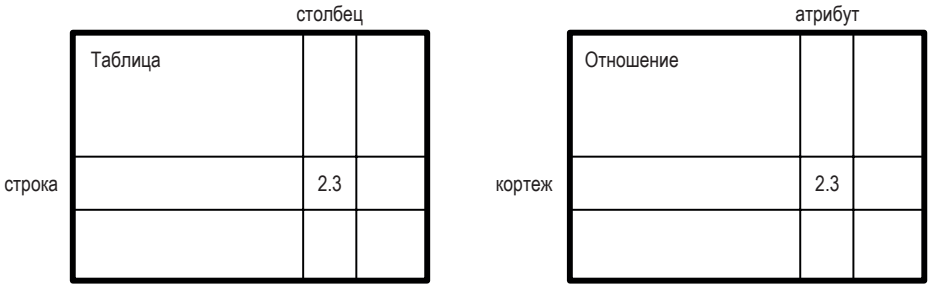


Рис. 15.1 ❖ Реляционные базы данных

15.3. Браузер базы данных для SQLite

Несмотря на то что в этой главе основное внимание уделено использованию Python для работы с данными в файлах базы данных SQLite, многие операции более удобно можно выполнять с помощью программы под названием Database Browser for SQLite (Браузер базы данных для SQLite), которая свободно (бесплатно) доступна здесь:

http://sqlitebrowser.org/.

С помощью этого браузера вы с легкостью можете создавать таблицы, вставлять и редактировать данные или выполнять простые SQL-запросы для извлечения данных из базы.

В некотором смысле этот браузер базы данных похож на текстовый редактор при работе с текстовыми файлами. Когда необходимо выполнить одну или несколько операций в текстовом файле, можно просто открыть файл в редакторе и внести требуемые изменения. Когда в текстовом файле необходимо сделать очень много изменений, то чаще всего вы пишете простую программу на Python. Мы обнаружим точно такой же подход и при работе с базами

данных. Простые операции будут выполняться в менеджере (браузере) базы данных, а более сложные намного удобнее выполнять с помощью Python.

15.4. Создание таблицы базы данных

Для баз данных требуется более строго определенная структура, чем для списков и словарей Python[1].

При создании таблицы (table) базы данных мы обязательно должны предварительно сообщить имя каждого столбца (column) в таблице и тип данных, которые мы планируем хранить в каждом столбце. Если программное обеспечение базы данных знает тип данных в каждом столбце, то может выбрать наиболее эффективный способ хранения и поиска данных на основе их типа.

О различных типах данных, поддерживаемых SQLite, можно узнать здесь:

http://www.sqlite.org/datatypes.html.

Предварительное определение структуры данных сначала может показаться неудобным, но это плата за быстрый доступ, даже если база содержит весьма большой объем данных.

Ниже приведен исходный код для создания файла базы данных и таблицы в ней с именем `Tracks`, содержащей два столбца:

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()

# Исходный код: http://www.py4e.com/code3/db1.py
```

Операция `connect` создает «соединение» с базой данных, хранящейся в файле *music.sqlite* в текущем каталоге. Если файл не существует, то он будет создан. Причина применения термина «соединение» заключается в том, что иногда база данных хранится на отдельном «сервере базы данных», отличающемся от сервера, на котором выполняется приложение. Во всех наших простых примерах база данных будет представлять собой только локальный файл в том же каталоге, в котором выполняется код Python.

Курсор (cursor) похож на дескриптор файла, и его можно использовать для выполнения операций с данными, хранящимися в базе. Вызов `cursor()` концептуально очень похож на вызов `open()` при работе с текстовыми файлами.

[1] В действительности SQLite допускает некоторую гибкость в определении типов данных, хранящихся в столбце, но в этой главе мы будем соблюдать строгость типов данных, чтобы концепции оставались применимыми и в других системах управления базами данных, таких как MySQL.

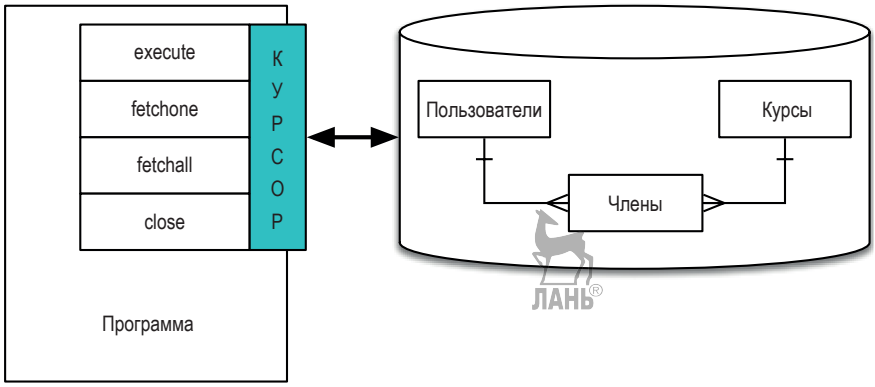


Рис. 15.2 ❖ Курсор базы данных

После создания курсора можно начать выполнение команд для работы с содержимым базы данных, используя метод `execute()`.

Команды базы данных подаются на специализированном языке, который был стандартизирован многими различными производителями систем управления базами данных, чтобы позволить пользователям изучить только один язык для работы с любыми базами данных. Этот специализированный стандартный язык называется Structured Query Language (язык структурированных запросов), или сокращенно SQL:

https://en.wikipedia.org/wiki/SQL;
https://ru.wikipedia.org/wiki/SQL.

В приведенном выше примере в созданной базе данных выполняются две команды языка SQL. По общему соглашению ключевые слова языка SQL записываются в верхнем регистре, а прочие части команды, которые мы добавляем (здесь: имена таблицы и столбцов), – в нижнем регистре.

Первая команда SQL удаляет таблицу `Tracks` из базы данных, если она существует. Этот шаблон просто позволяет запускать программу для создания таблицы `Tracks` снова и снова без возникновения ошибки. Обратите внимание: команда `DROP TABLE` удаляет таблицу и все ее содержимое из базы данных (т. е. невозможно «отменить» действие этой команды).

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

Вторая команда создает таблицу с именем `Tracks` с текстовым столбцом, названным `title`, и с целочисленным столбцом под названием `plays`.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

После создания таблицы `Tracks` можно записывать в нее данные, используя команду SQL `INSERT`. И в этом случае мы начинаем с установления соединения с базой данных и получения курсора. Затем выполняются команды SQL с использованием этого курсора.

Команда SQL `INSERT` определяет, какую таблицу мы используем, затем определяет новую строку, перечисляя поля, которые необходимо включить

(title, plays), а далее следуют значения VALUES, помещаемые в эту новую строку. Значения мы задаем как знаки вопроса (?, ?), чтобы сообщить, что действительные значения передаются как кортеж ('My Way', 15) во втором параметре вызова execute().

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
            ('Thunderstruck', 20))
cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
            ('My Way', 15))
conn.commit()

print('Tracks:')
cur.execute('SELECT title, plays FROM Tracks')
for row in cur:
    print(row)

cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()

cur.close()

# Исходный код: http://www.py4e.com/code3/db2.py
```

Tracks

title	plays
Thunderstruck	20
My Way	15

Рис. 15.3 ❖ Строки в таблице

Сначала мы вставляем (INSERT) две строки в таблицу и используем метод commit(), чтобы действительно записать эти данные в файл базы.

Затем выполняется команда SELECT для извлечения строк, которые только что были вставлены в таблицу. В команде SELECT мы определяем требуемые столбцы (title, plays) и сообщаем, из какой таблицы нужно извлечь данные. После выполнения команды SELECT курсор становится объектом, по которому можно пройти в цикле с помощью инструкции for. В целях эффективности курсор не читает все данные из базы, когда мы выполняем команду SELECT. Вместо этого данные считываются по запросу, когда мы проходим по строкам в цикле for.

Вывод этой программы показан ниже:

```
Tracks:
('Thunderstruck', 20)
('My Way', 15)
```

Цикл for нашел две строки, и каждая строка является кортежем Python с первым значением из столбца title и вторым – целым числом из столбца plays.

> **Примечание** В других книгах или в интернете вы можете увидеть строки, начинающиеся с префикса u'. Это старое обозначение в версии Python 2, сообщающее, что строки записаны в кодировке Unicode, т. е. могут содержать не только символы латинского алфавита. В версии Python 3 все строки по умолчанию записаны в кодировке Unicode.

В самом конце программы выполняется команда SQL DELETE для удаления только что созданных строк, чтобы программу можно было запускать многократно. В команде DELETE демонстрируется применение спецификатора WHERE, позволяющего определить условие выбора, чтобы можно было попросить базу данных применить эту команду только к тем строкам, которые соответствуют заданному условию. В приведенном выше примере условие определяет применение ко всем строкам, поэтому таблица очищается полностью, и можно выполнять программу многократно. После выполнения команды DELETE также вызывается метод commit() для действительного удаления данных из базы.

15.5. Обзор языка структурированных запросов SQL

До сих пор мы использовали язык структурированных запросов SQL в примерах кода на языке Python и уже рассмотрели многие из основных команд SQL. В этом разделе более подробно рассматривается язык SQL и приводится обзор синтаксиса этого языка.

Поскольку существует много различных производителей систем управления базами данных, язык структурированных запросов SQL (Structured Query Language) был стандартизирован, чтобы пользователи могли обмениваться информацией переносимым способом с системами баз данных от различных производителей.

Реляционная база данных состоит из таблиц, строк и столбцов. В общем случае столбцы имеют тип, такой как текст, число или календарная дата. При создании таблицы определяются имена и типы столбцов:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

Для вставки строки в таблицу применяется команда SQL INSERT:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

Команда INSERT определяет имя таблицы, затем список столбцов/полей, для которых будут установлены значения в новой строке, далее следует ключевое слово VALUES и список соответствующих значений для каждого заданного поля.

Команда SQL SELECT используется для извлечения строк и столбцов из базы данных. Команда SELECT позволяет определить, какие столбцы необходимо

извлечь, а также указать спецификатор WHERE для выбора по условию строк, которые нужно рассматривать. Также можно записать дополнительный (необязательный) спецификатор ORDER BY для управления сортировкой возвращаемых строк.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

Применение символа * означает, что требуется возврат из базы данных всех столбцов из каждой строки, которая соответствует условию, определенному в спецификаторе WHERE.

Следует отметить, что, в отличие от Python, в SQL-спецификаторе WHERE используется только один знак = для проверки на равенство, а не два (==). Другими логическими операторами, допустимыми в спецификаторе WHERE, являются <, >, <=, >=, !=, а также AND, OR и круглые скобки для создания логических выражений.

Можно потребовать, чтобы возвращаемые строки были отсортированы по одному из полей, как показано ниже:

```
SELECT title,plays FROM Tracks ORDER BY title
```

Для удаления строки необходимо наличие спецификатора WHERE в команде SQL DELETE. Спецификатор WHERE определяет, какие строки должны быть удалены:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

Существует возможность обновления (UPDATE) столбца или нескольких столбцов в одной или в нескольких строках таблицы с использованием команды SQL UPDATE, как показано ниже:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

Команда UPDATE определяет таблицу, затем список полей и изменяемых значений после ключевого слова SET, а далее следует необязательный спецификатор WHERE для выбора строк, которые необходимо обновить. Одна команда UPDATE может обновить все строки, соответствующие условию в спецификаторе WHERE. Если спецификатор WHERE не задан, то выполняется обновление всех строк в таблице.

Эти четыре основные команды SQL (INSERT, SELECT, UPDATE и DELETE) позволяют выполнять четыре основные операции, необходимые для создания, обработки и сопровождения данных.

15.6. Реализация глобального поиска в Twitter с использованием базы данных

В этом разделе мы создадим простую программу глобального поиска (spidering), которая проходит по аккаунтам Twitter и создает базу данных по ним.

Примечание Будьте чрезвычайно осторожны и внимательны при запуске этой программы. Не следует извлекать слишком много данных или выполнять программу в течение очень длительного времени, чтобы Twitter не закрыл доступ к вашему собственному аккаунту.

Одна из сложностей для этого типа программ глобального поиска заключается в том, что требуется возможность многократного останова и повторного запуска, но весьма нежелательна потеря данных, которые уже были извлечены. Не хотелось бы всегда перезапускать процедуру извлечения данных с самого начала, поэтому необходимо сохранять ранее извлеченные данные, чтобы программа могла начать с того места, где она остановилась в прошлый раз.

Начнем с извлечения данных о друзьях одного пользователя, имеющего аккаунт в Twitter, и о состояниях этих друзей, проходя в цикле по списку их имен и добавляя имя каждого друга в базу данных, чтобы можно было извлечь его в будущем. После обработки всех друзей этого пользователя Twitter мы проверяем созданную базу данных и извлекаем одного из друзей конкретного друга. Мы повторяем это действие многократно, выбирая каждый раз «пока еще не посещенного» пользователя, извлекая список его друзей и добавляя те имена, которые пока еще не появлялись в списке для следующих посещений.

Кроме того, мы отслеживаем, сколько раз имя конкретного друга появлялось в базе данных, чтобы получить некоторое представление о его «популярности».

Сохраняя список известных аккаунтов и сведения о том, извлекался ли данный аккаунт ранее или нет, и о популярности этого аккаунта на жестком диске компьютера, мы можем останавливать и перезапускать программу столько раз, сколько пожелаем.

Эта программа несколько более сложная. Она основана на исходном коде упражнения из главы 13 этой книги, в котором использовался API Twitter.

Ниже приведен исходный код приложения, выполняющего глобальный поиск (spidering) в Twitter.

```
from urllib.request import urlopen
import urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS Twitter
            (name TEXT, retrieved INTEGER, friends INTEGER)''')

# Игнорировать ошибки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
```

```
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print('No unretrieved Twitter accounts found')
            continue

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'})
    print('Retrieving', url)
    connection = urlopen(url, context=ctx)
    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Remaining', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    # Отладочная инструкция:
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1', (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?', (count+1, friend))
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                        VALUES (?, 0, 1)''', (friend, ))
            countnew = countnew + 1
    print('New accounts=', countnew, ' revisited=', countold)
    conn.commit()

cur.close()

# Исходный код: http://www.py4e.com/code3/twspider.py
```

Эта база данных хранится в файле *spider.sqlite* и содержит одну таблицу с именем `Twitter`. Каждая строка в таблице `Twitter` содержит столбец для имени аккаунта, признак выполненной операции извлечения друзей этого аккаунта и количество его «попаданий в друзья».

В основном цикле программы пользователю предлагается ввести имя аккаунта в Twitter или слово «quit» для выхода. Если пользователь вводит имя

аккаунта Twitter, то мы извлекаем список друзей и их состояний для этого пользователя и добавляем имя каждого друга в базу данных, если он пока еще туда не записан. Если друг уже находится в списке, то к значению поля `friends` прибавляется 1 в соответствующей строке базы данных.

Если пользователь нажимает клавишу **Enter** (Ввод), то мы ищем в базе данных следующий аккаунт Twitter, который пока еще не извлекался, получаем список друзей и состояний для этого аккаунта, добавляем его в базу данных или обновляем поля его строки и инкрементируем соответствующий счетчик `friends`.

После получения списка друзей и состояний выполняется проход в цикле по всем элементам `user` в возвращенном коде JSON и извлечение `screen_name` для каждого пользователя. Затем мы используем команду `SELECT`, чтобы проверить, не сохранялось ли это конкретное имя `screen_name` в базе данных, и если такая запись существует, то извлекается ее счетчик `friends`.

```
countnew = 0
countold = 0
for u in js['users']:
    friend = u['screen_name']
    print(friend)
    cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1', (friend, ))
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?', (count+1, friend))
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                       VALUES (?, 0, 1)''', (friend, ))
        countnew = countnew + 1
print('New accounts=', countnew, ' revisited=', countold)
conn.commit()
```

После того как курсор выполнит команду `SELECT`, необходимо извлечь строки. Можно было бы сделать это с помощью цикла `for`, но поскольку мы извлекаем только одну строку (`LIMIT 1`), то можно воспользоваться методом `fetchone()` для получения первой (и только первой) строки, являющейся результатом операции выбора `SELECT`. Так как `fetchone()` возвращает строку как кортеж (даже если он содержит только одно поле), мы берем первое значение из этого кортежа, используемое для получения текущего счетчика друзей в переменную count.

Если эта операция извлечения прошла успешно, то применяется команда SQL `UPDATE` со спецификатором `WHERE` для прибавления 1 к значению столбца `friends` для строки, которая совпадает с именем аккаунта найденного друга. Обратите внимание: в команде SQL указаны два шаблона (т. е. знаки вопроса), а вторым параметром метода `execute()` является кортеж из двух элементов, содержащий значения для подстановки в команду SQL вместо знаков вопроса.

Если код в блоке `try` завершается неудачно, то, вероятно, потому, что нет записей, соответствующих условию спецификатора `WHERE name = ?` в команде

SELECT. Тогда в блоке except используется команда SQL INSERT для добавления имени screen_name текущего друга в таблицу с указанием о том, что мы пока еще ни разу не извлекали имя screen_name, и его счетчик friends устанавливается в единицу.

При первом запуске этой программы после входа в аккаунт Twitter выводится следующая информация:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20 revisited= 0
Enter a Twitter account, or quit: quit
```

Поскольку это самый первый запуск программы, база данных пуста, и мы создаем ее в файле *spider.sqlite*, потом добавляем в нее таблицу Twitter. Затем извлекаем список имен нескольких друзей и добавляем их в таблицу, так как база данных до этого была пустой.

В этом месте, возможно, потребуется написать простой код для вывода всего содержимого базы данных (database dumper), чтобы просмотреть содержимое файла *spider.sqlite*:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
    count = count + 1
print(count, 'rows.')
cur.close()

# Исходный код: http://www.py4e.com/code3/twdump.py
```

Эта программа просто открывает файл базы данных и выбирает все столбцы всех строк в таблице Twitter, затем в цикле проходит по строкам и выводит содержимое каждой строки.

Если запустить эту программу после первого выполнения приложения глобального поиска в Twitter, то она выведет следующую информацию:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 rows.
```

Мы видим одну строку для каждого имени screen_name, признак того, что мы пока еще не извлекали данные для имен screen_name, и информацию о том, что каждый пользователь в этой базе данных имеет одного друга.

В текущий момент база данных отображает извлечение списка друзей первого заданного аккаунта Twitter (drchuck). Можно запустить эту программу еще раз и попросить ее извлечь друзей для следующего «необработанного» аккаунта, просто нажав клавишу **Ввод** (Enter) вместо ввода имени аккаунта Twitter, как показано ниже:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18 revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17 revisited= 3
Enter a Twitter account, or quit: quit
```

Так как была нажата клавиша **Ввод** (т. е. не задан конкретный аккаунт Twitter), выполняется показанный ниже фрагмент кода:

```
if (len(acct) < 1):
    cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print('No unretrieved Twitter accounts found')
        continue
```

Мы используем команду SQL `SELECT` для получения имени первого (`LIMIT 1`) пользователя, для которого значение со смыслом «извлекался ли этот пользователь ранее» остается равным нулю. Также используется шаблон `fetchone()[0]` в блоке `try/except` либо для извлечения имени `screen_name` из полученных данных, либо для вывода сообщения об ошибке (о неудачном завершении операции поиска) и немедленного возврата к началу цикла.

Если успешно получено необработанное имя `screen_name`, то мы извлекаем соответствующие ему данные, как показано ниже:

```
url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'})
print('Retrieving', url)
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)

cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ))
```

После успешного извлечения данных используется команда `UPDATE` для установки значения столбца `retrieved` в 1, обозначающую, что мы завершили извлечение всех имен друзей для этого аккаунта. Это позволяет избежать повторного извлечения одних и тех же данных и способствует продвижению вперед по сети друзей в Twitter.

Если еще раз запустить программу и нажать клавишу **Ввод** дважды, чтобы получить следующий список друзей непосещенного друга, а затем выполнить вспомогательную программу вывода содержимого базы данных, то получим следующий вывод:

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 rows.
```

Здесь можно видеть, что правильно зафиксировано посещение имен `lhawthorn` и `opencontent`. Кроме того, аккаунты `cnxorg` и `kthanos` уже имеют двух подписчиков. Поскольку теперь мы извлекли имена друзей трех пользователей (`drchuck`, `opencontent` и `lhawthorn`), в таблице содержится 55 строк с записями об извлеченных именах друзей.

При каждом следующем запуске программы и нажатии клавиши **Ввод** она будет выбирать следующий непосещенный аккаунт (например, сейчас следующим аккаунтом будет `steve_coppin`), извлекать имена его друзей, помечать их как извлеченные, и для всех друзей пользователя `steve_coppin` либо добавлять их в конец базы данных, либо обновлять их счетчик `friends`, если они уже записаны в базу данных.

Так как все данные, с которыми работает эта программа, сохраняются в файл базы данных на жестком диске, операция глобального поиска может приостанавливаться и возобновляться столько раз, сколько вы пожелаете, без потерь данных.

15.7. Основы моделирования данных

Истинная мощь реляционных баз данных проявляется, когда создается несколько таблиц и устанавливаются связи между этими таблицами. Процесс принятия решения о том, как разделить данные, с которыми работает приложение, на несколько таблиц, и установления отношений между этими таблицами называется моделированием данных (data modeling). Проектный документ, отображающий таблицы и отношения между ними, называется моделью данных (data model).

Моделирование данных – относительно сложное искусство, поэтому в этом разделе представлены только самые общие концепции моделирования реляционных данных. Более подробное изучение моделирования данных можно начать отсюда:

http://en.wikipedia.org/wiki/Relational_model;
http://ru.wikipedia.org/wiki/Реляционная_модель_данных.

Предположим, что в нашем приложении глобального поиска в Twitter вместо простого подсчета друзей каждого пользователя необходимо сохра-

нять список всех входящих отношений, чтобы можно было составить список всех пользователей, следящих за конкретным аккаунтом.

Поскольку у любого пользователя потенциально может обнаружиться множество аккаунтов, наблюдающих за ним, невозможно просто добавить один столбец в таблицу `Twitter`. Поэтому мы создаем новую таблицу, в которой постоянно отслеживаются пары друзей. Ниже показан простой способ создания такой таблицы:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

Каждый раз при обнаружении пользователя, следящего за аккаунтом `drchuck`, необходимо вставлять в новую таблицу строку следующей формы:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

Если мы обрабатываем 20 имен друзей, полученных по аккаунту `drchuck` в Twitter, то вставляем 20 записей с указанием `drchuck` как первого параметра и в конце концов придем к многократному дублированию одной и той же строки в нашей базе данных.

Такое дублирование данных типа строка нарушает одно из наиболее эффективных практических правил нормализации базы данных (database normalization), утверждающее, что мы никогда не должны помещать одну и ту же строку данных в таблицу базы более одного раза. Если требуется запись данных более одного раза, то для таких данных создается числовой ключ (key), и любая ссылка на реальные данные должна использовать этот ключ.

С практической точки зрения строка занимает намного больше места, чем целое число как на диске, так и в памяти компьютера, а кроме того, требует больше процессорного времени для сравнения и сортировки. Если имеется лишь несколько сотен записей, то объем хранилища и процессорное время вряд ли имеют значение. Но при наличии миллиона пользователей в нашей базе данных и потенциальной возможности обработки 100 млн ссылок на друзей становится весьма важной способность просмотра и поиска данных с максимально возможной скоростью.

Мы будем хранить аккаунты Twitter в таблице с именем `People` вместо таблицы `Twitter`, используемой в предыдущем примере. В таблице `People` имеется дополнительный столбец для хранения числового ключа, связанного с конкретной строкой для каждого пользователя Twitter. В SQLite существует функциональная возможность автоматического добавления значения ключа для любой строки, которая вставляется в таблицу при использовании специального типа столбца (`INTEGER_PRIMARY_KEY`).

Можно создать таблицу `People` с таким дополнительным столбцом `id`, как показано ниже:

```
CREATE TABLE People (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

Обратите внимание: мы больше не отслеживаем счетчик друзей в каждой строке таблицы `People`. После выбора `INTEGER_PRIMARY_KEY` как типа столбца `id`

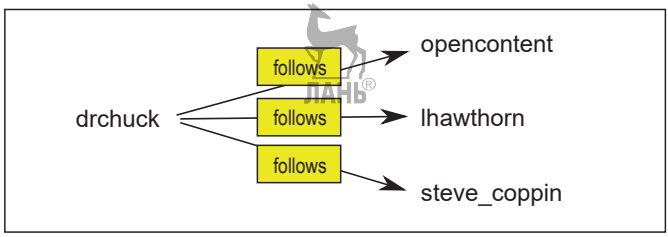
мы показываем, что хотели бы передать SQLite управление этим столбцом и автоматическое присваивание неповторяющегося числового ключа каждой вставляемой строке. Кроме того, мы добавляем ключевое слово `UNIQUE`, означающее, что не разрешаем SQLite вставлять в таблицу две строки с одинаковым значением поля `name`.

Теперь вместо создания таблицы `Pals`, показанного выше, мы создаем таблицу Follows с двумя целочисленными столбцами `from_id` и `to_id` и ограничивающим условием для этой таблицы, определяющим, что комбинация столбцов `from_id` и `to_id` обязательно должна быть не повторяющейся в этой таблице (т. е. запрещено вставлять одинаковые строки) нашей базы данных.

```
CREATE TABLE Follows (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))
```

При добавлении спецификаторов `UNIQUE` в создаваемые таблицы мы определяем набор правил, которые непременно обязана соблюдать база данных при наших попытках вставки записей. Мы создаем эти правила как удобный прием в своих программах, как мы в скором времени убедимся. Эти правила оберегают нас от совершения ошибок и упрощают написание некоторой части исходного кода.

По существу, при создании этой таблицы `Follows` мы моделируем «отношение», в котором один пользователь «следует» за каким-либо другим, и представляем это отношение с помощью пары чисел, означающей, что (a) эти пользователи соединены (отношением) и (b) определено направление этого отношения.



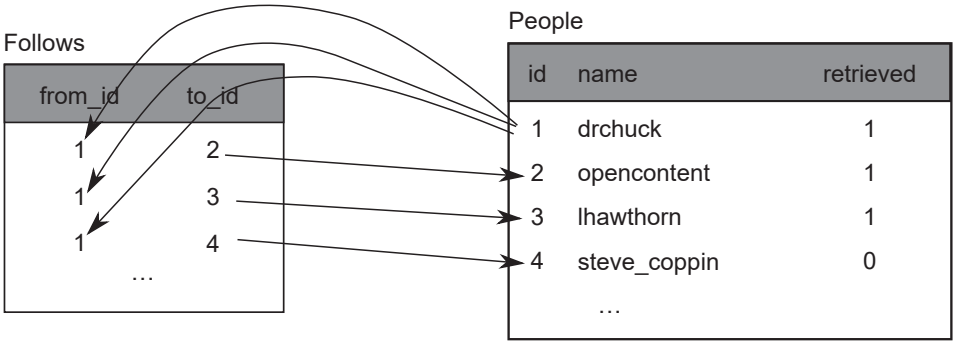


Рис. 15.4 ❖ Отношения между таблицами

15.8. Программирование с использованием нескольких таблиц

Теперь мы перепишем программу глобального поиска (spidering) в Twitter, используя две таблицы, главный (первичный) ключ (primary key) и ссылки на этот ключ, как описано в предыдущем разделе. Ниже приведен исходный код для новой версии этой программы.


```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS People
               (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
               (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')

# Игнорировать ошибки сертификатов SSL.
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT id, name FROM People WHERE retrieved=0 LIMIT 1')
        try:
            (id, acct) = cur.fetchone()
        except:
            print('No unretrieved Twitter accounts found')
            continue
    else:
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1', (acct, ))
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO People
                        (name, retrieved) VALUES (?, 0)''', (acct, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', acct)
                continue
            id = cur.lastrowid
```

```
    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '100'})
    print('Retrieving account', acct)
    try:
        connection = urllib.request.urlopen(url, context=ctx)
    except Exception as err:
        print('Failed to Retrieve', err)
        break

    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Remaining', headers['x-rate-limit-remaining'])

    try:
        js = json.loads(data)
    except:
        print('Unable to parse json')
        print(data)
        break

    # Отладочная инструкция:
    # print(json.dumps(js, indent=4))

    if 'users' not in js:
        print('Incorrect JSON received')
        print(json.dumps(js, indent=4))
        continue

    cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1', (friend, ))
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                        VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
                    VALUES (?, ?)''', (id, friend_id))
    print('New accounts=', countnew, ' revisited=', countold)
    print('Remaining', headers['x-rate-limit-remaining'])
    conn.commit()

cur.close()

# Исходный код: http://www.py4e.com/code3/twfriends.py
```

Эта программа немного сложнее для понимания, но в ней демонстрируются шаблоны, которые необходимо применять при использовании целочисленных ключей для связывания таблиц базы данных. Основные шаблоны описаны ниже:

1) создать таблицы с главными (первичными) ключами и ограничивающими условиями;
2) когда мы имеем логический ключ для пользователя (т. е. имя аккаунта) и требуется значение идентификатора `id` для этого пользователя в зависимости от того, находится или отсутствует данный пользователь в таблице `People`, необходимо: (1) найти этого пользователя в таблице `People` и извлечь значение `id` для него или (2) добавить этого пользователя в таблицу `People` и получить значение `id` для новой добавленной строки;
3) вставить строку, которая содержит отношение «следования».

Рассмотрим каждый из этих шаблонов по отдельности.

15.8.1. Ограничивающие условия в таблицах базы данных

При проектировании структуры таблиц можно сообщить системе управления базой данных, что мы намереваемся применить несколько собственных правил. Эти правила помогают избежать ошибок и ввода некорректных данных в таблицы и определяются при создании таблиц:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
               (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
               (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

Мы сообщаем, что столбец name в таблице People обязательно должен быть неповторяющимся (`UNIQUE`). Еще определяется, что комбинация двух чисел в каждой строке таблицы `Follows` также должна быть неповторяющейся. Эти ограничения предохраняют нас от совершения ошибок, таких как добавление некоторого отношения более одного раза.

Мы можем воспользоваться преимуществом от применения этих ограничивающих условий в следующем коде:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved) VALUES ( ?, 0)''', ( friend, ) )
```

Мы добавляем спецификатор `OR IGNORE` в команду `INSERT`, сообщая, что если эта конкретная операция вставки приведет к нарушению правила «имя `name` обязательно должно быть неповторяющимся», то системе управления базой данных разрешено игнорировать эту команду `INSERT`. Ограничивающие условия для базы данных используются как защитная сеть для уверенности в том, что мы не сделаем что-либо неправильно без какого бы то ни было умысла.

Аналогично приведенный ниже код гарантирует, что мы не добавим дважды одно и то же отношение следования в таблицу `Follows`.

15.8.2. Извлечение и/или вставка записи

Когда мы предлагаем пользователю ввести аккаунт Twitter, если такой аккаунт существует, то необходимо найти соответствующее ему значение идентификатора `id`. Если такого аккаунта пока еще нет в таблице `People`, то необходимо вставить соответствующую запись и взять значение `id` из этой новой вставленной строки.

Это весьма часто применяемый шаблон, который выполняется дважды в приведенной выше программе. Его код показывает, как мы определяем `id` для аккаунта друга после извлечения имени `screen_name` из узла `user` в полученном от Twitter коде JSON.

Поскольку со временем вероятность того, что аккаунт уже находится в базе данных, возрастет, мы сначала проверяем, существует ли такая запись в таблице `People`, используя для этого команду `SELECT`.

Если все идет как нужно[1] внутри секции `try`, то мы извлекаем запись с помощью метода `fetchone()`, затем извлекаем первый (только один) элемент из возвращенного кортежа и сохраняем его в переменной `friend_id`.

Если команда `SELECT` завершается неудачно, то при выполнении кода `fetchone()[0]` возникает ошибка, и управление передается в секцию `except`.

```
friend = u['screen_name']
cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1', (friend, ))
try:
    friend_id = cur.fetchone()[0]
    countold = countold + 1
except:
    cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES (?, 0)''', (friend, ))
    conn.commit()
    if cur.rowcount != 1:
        print('Error inserting account:', friend)
        continue
    friend_id = cur.lastrowid
    countnew = countnew + 1
```

Если эта операция завершается в коде секции `except`, это просто означает, что строка не найдена, и необходимо вставить новую строку. Мы используем команду `INSERT OR IGNORE`, чтобы избежать ошибок, затем вызываем метод `commit()` для реального обновления базы данных. После завершения операции записи на диск можно проверить значение `cur.rowcount`, т. е. количество действительно измененных строк. Поскольку мы пытаемся вставить одну строку, то если число измененных строк отличается от 1, значит, это сигнал об ошибке.

Если команда `INSERT` завершилась успешно, то можно проверить `cur.lastrowid` и посмотреть, какое значение присвоила база данных столбцу `id` в новой вставленной строке.

---

[1] В общем случае, если предложение начинается с фразы «если все идет как нужно», вы обнаружите, что в коде требуется применение `try/except`.

15.8.3. Сохранение отношения следования за другом

Как только мы узнаем значение ключа как для пользователя Twitter, так и для друга в коде JSON, не составляет никакого труда вставить два числа в таблицу `Follows` с помощью следующего кода:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)', (id, friend_id) )
```

Обратите внимание: мы позволяем базе данных заботиться о том, чтобы не допустить «двойной вставки» отношения, создавая таблицу с ограничивающим условием неповторяемости и затем добавив спецификатор `OR IGNORE` в команду `INSERT`.

Ниже приведен пример вывода при выполнении этой программы:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20 revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17 revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17 revisited= 3
Enter a Twitter account, or quit: quit
```

Мы начинаем с аккаунта `drchuck`, затем позволяем программе автоматически выбрать следующие два аккаунта для извлечения и добавления в базу данных.

Ниже показано несколько первых строк в таблицах `People` и `Follows` после завершения выполнения программы:

```
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

Здесь можно видеть поля идентификатора id, имени name и посещений visited в таблице People, а также числовые значения на обеих сторонах отношения в таблице Follows. В таблице People мы видим, что первые три пользователя были посещены, и их данные извлечены. Данные в таблице Follows показывают, что drchuck (пользователь 1) является другом всех людей, показанных в первых пяти строках. Это имеет смысл, потому что первыми данными, которые мы извлекли и сохранили, были данные друзей drchuck в Twitter. Если бы мы вывели больше строк из таблицы Follows, то также увидели бы друзей пользователей 2 и 3.

15.9. Три типа ключей

Теперь, когда мы начали создавать модель данных, помещая данные в несколько связанных таблиц и связывая строки в этих таблицах с помощью ключей, нам нужно уточнить некоторую терминологию, связанную с ключами. Обычно в модели базы данных используются ключи трех типов.

- Логический ключ (logical key) – это ключ, который в «реальном мире» может использоваться для поиска строки. В нашем примере модели данных поле имени name является логическим ключом. Это имя screen_name пользователя Twitter, и мы действительно несколько раз ищем строку пользователя в программе, используя поле name. Вы часто будете замечать, что имеет смысл добавить ограничивающее условие UNIQUE к логическому ключу. Поскольку логический ключ определяет то, как мы ищем строку из внешнего мира, не имеет смысла разрешать существование нескольких строк с одним и тем же значением в таблице.
- Главный (или первичный) ключ (primary key) – это обычно числовое значение, которое присваивается автоматически базой данных. Вообще говоря, он не имеет никакого смысла вне программы и используется только для связывания строк из разных таблиц. Когда мы хотим найти строку в таблице, как правило, поиск строки с использованием главного ключа – это самый быстрый способ найти требуемую строку. Поскольку главные ключи являются целыми числами, они занимают очень мало места при хранении, и их можно весьма быстро сравнивать или сортировать. В нашей модели данных поле id является примером первичного ключа.
- Внешний ключ (foreign key) – это обычно числовое значение, которое указывает на главный (первичный) ключ связанной строки в другой таблице. В нашей модели данных поле from_id является примером внешнего ключа.

Мы используем следующее соглашение об именах: всегда называем поле первичного ключа id и добавляем суффикс _id к каждому имени поля, которое является внешним ключом.

15.10. Использование JOIN для извлечения данных

Теперь, когда мы выполнили правила нормализации базы данных и разделили данные на две таблицы, связанные вместе с помощью главного (первичного) ключа и внешних ключей, нам нужна возможность формирования команды SELECT, которая повторно собирает данные из разных таблиц.

SQL использует спецификатор JOIN для соединения этих таблиц. В спецификаторе JOIN определяются поля, которые используются для соединения строк между таблицами.

Ниже приведен пример команды SELECT с использованием спецификатора JOIN:

```
SELECT * FROM Follows JOIN People ON Follows.from_id = People.id WHERE People.id = 1
```

Спецификатор JOIN указывает, что выбираемые поля существуют в обеих таблицах Follows и People. Спецификатор ON определяет, как эти две таблицы должны быть соединены: берутся строки из Follows и добавляется строка из People, в которой поле from_id в Follows содержит то же значение, что и поле id в таблице People.

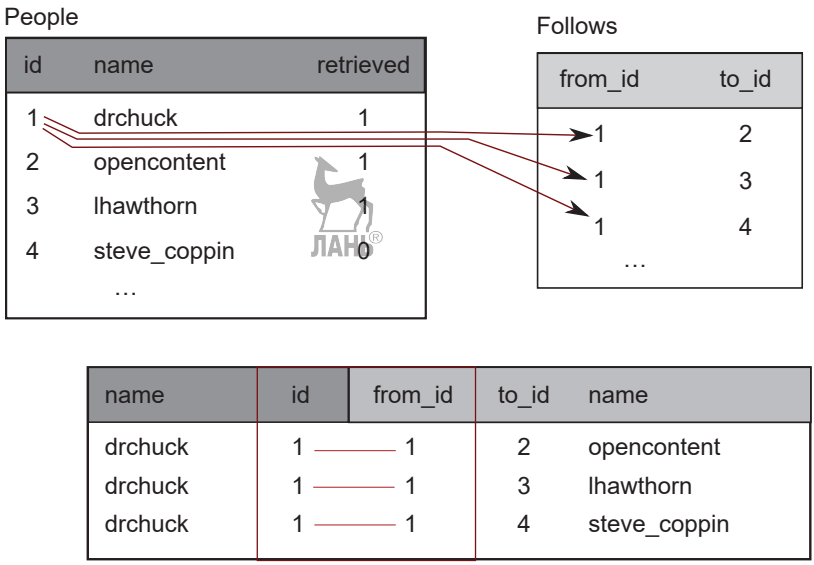


Рис. 15.5 ❖ Соединение таблиц с использованием спецификатора JOIN

Результатом соединения JOIN является создание сверхдлинных «метастрок», в которых есть поля из таблицы People и из соответствующих полей таблицы Follows. Если существует более одного соответствия между полем

id из People и from_id из People, то JOIN создает метастроку для каждой из совпадающих пар строк, дублируя данные по мере необходимости.

В приведенном ниже коде демонстрируются данные, содержащиеся в базе данных после того, как программа глобального поиска в Twitter с использованием нескольких таблиц (приведенная выше) была выполнена несколько раз.

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print('People:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.execute('SELECT * FROM Follows')
count = 0
print('Follows:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.execute('''SELECT * FROM Follows JOIN People ON Follows.to_id = People.id
               WHERE Follows.from_id = 2''')
count = 0
print('Connections for id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.close()

# Исходный код: http://www.py4e.com/code3/twjoin.py
```

В этой программе сначала выводится содержимое таблиц People и Follows, затем выводится подмножество данных в соединении этих таблиц.

Ниже показан вывод данной программы:

```
python twjoin.py
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
```

```
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, 'drchuck', 1)
(2, 28, 28, 'cnxorg', 0)
(2, 30, 30, 'kthanos', 0)
(2, 102, 102, 'SomethingGirl', 0)
(2, 103, 103, 'ja_Pac', 0)
20 rows.
```


Здесь можно видеть столбцы из таблиц `People` и `Follows`, а последний набор строк является результатом выполнения команды `SELECT` со спецификатором `JOIN`.

В последнем фрагменте результатов выполнения команды выбора мы ищем аккаунты, которые являются друзьями пользователя `opencontent` (т. е. `People.id=2`).

В каждой из «метастрок» в последнем фрагменте выбора первые два столбца взяты из таблицы `Follows`, а за ними следуют столбцы с третьего по пятый из таблицы `People`. Также можно видеть, что второй столбец (`Follows.to_id`) совпадает с третьим столбцом (`People.id`) в каждой из соединенных «метастрок».

15.11. Резюме

В этой главе было рассмотрено множество вопросов, что позволило представить обзор основ использования баз данных в Python. Код для использования базы данных сложнее написать, чем код для работы со словарями Python или простыми файлами, поэтому нет особых причин использовать базу данных, если вашему приложению действительно не нужны ее специальные возможности. Ситуации, в которых база данных может быть весьма полезной: (1) когда вашему приложению необходимо сделать много небольших произвольных обновлений в большом наборе данных, (2) когда ваши данные настолько велики, что не могут поместиться в словаре, и вам нужно многократно выполнять поиск информации или (3) если существует длительный процесс, который вы хотите останавливать, перезапускать и сохранять данные после очередного сеанса выполнения.

Можно создать простую базу данных с одной таблицей, чтобы удовлетворить потребности многих приложений, но для большинства задач потребуется несколько таблиц и связи/отношения между строками в разных таблицах. Когда вы начинаете устанавливать связи между таблицами, важно тщательно продумать проектное решение и выполнять правила нормализации баз данных, чтобы максимально использовать их функциональные возможности.

Поскольку основная причина использования базы данных – это необходимость обработки больших объемов данных, важно эффективно моделировать данные, чтобы программы работали как можно быстрее.

15.12. Отладка

Один из самых распространенных шаблонов при разработке программы на языке Python для установления соединения с базой данных SQLite – это запуск программы на Python и проверка результатов ее работы с помощью браузера базы данных для SQLite (Database Browser for SQLite). Этот браузер позволяет быстро проверить, правильно ли работает ваша программа.

Вы должны действовать осторожно и внимательно, потому что SQLite следит за тем, чтобы две программы не изменяли одни и те же данные одновременно. Например, если вы открываете базу данных в браузере и вносите изменения в базу данных, но еще не нажали кнопку **Save** (Сохранить) в браузере, то браузер «блокирует» файл базы данных и не позволяет любой другой программе получить доступ к этому файлу. В частности, ваша программа на Python не сможет получить доступ к файлу, если он заблокирован.

В этом случае решение состоит в том, чтобы либо закрыть браузер базы данных, либо воспользоваться пунктом меню **File** (Файл), чтобы закрыть базу данных в браузере, прежде чем пытаться получить доступ к базе данных из Python, чтобы избежать проблемы, связанной с отказом при выполнении вашего кода Python из-за блокировки базы данных.

15.13. Словарь терминов

- Атрибут (attribute) – одно из значений в кортеже. Гораздо чаще называется «столбцом» или «полем».
- Браузер базы данных (database browser) – часть программного обеспечения, позволяющая устанавливать соединение непосредственно с базой данных и напрямую работать с данными без написания программы.
- Внешний ключ (foreign key) – числовой ключ, который указывает на главный (первичный) ключ строки в другой таблице. Внешние ключи устанавливают отношения между строками, хранящимися в разных таблицах.
- Главный (первичный) ключ (primary key) – числовой ключ, присваиваемый каждой строке и используемый для ссылки на одну строку таблицы из другой таблицы. Чаще всего база данных сконфигурирована для автоматического присваивания главных ключей при вставке каждой новой строки.
- Индекс (index) – дополнительные данные, которые база данных обрабатывает как строки и вставляет в таблицы для существенного ускорения поиска.
- Кортеж (tuple) – одна запись в таблице базы данных, представляющая собой набор атрибутов. Гораздо чаще кортеж называют строкой (row).

- Курсор (cursor) – позволяет выполнять команды SQL в базе данных и извлекать данные из базы. Курсор в определенной степени похож на сокет или на дескриптор файла для сетевых соединений или файлов соответственно.
- Логический ключ (logical key) – ключ, который во «внешнем мире» используется для поиска конкретной строки. Например, в таблице пользовательских аккаунтов адрес электронной почты пользователя может оказаться хорошим кандидатом при выборе логического ключа для пользовательских данных.
- Нормализация (normalization) – проектирование модели данных с полным исключением повторения одинаковых данных. Мы храним каждый элемент данных в одном определенном месте в базе данных, а для ссылок на него из любого другого места используется внешний ключ.
- Ограничивающее условие (constraint) – определяется в базе данных для выполнения правила в некотором поле или строке таблицы. Наиболее часто применяемое ограничивающее условие: запрещение дублирующихся значений в конкретном поле (т. е. все значения в этом поле должны быть неповторяющимися).
- Отношение (relation) – область внутри базы данных, содержащая кортежи и атрибуты. Гораздо чаще ее называют таблицей.

Глава 16

Визуализация данных

До сих пор мы осваивали основы языка Python, а затем учились использовать этот язык, сеть и базы данных для обработки данных.

В этой главе мы рассмотрим три полных приложения, которые объединяют весь ранее изученный материал для обработки и визуализации данных. Вы можете использовать эти приложения в качестве примеров кода, которые помогут вам приступить к решению реальной задачи.

Каждое приложение представляет собой ZIP-файл, который вы можете загрузить, распаковать на своем компьютере и выполнить.

16.1. Создание карты OpenStreetMap по данным геокодирования

В этом проекте мы используем API геокодирования OpenStreetMap, чтобы привести в порядок некоторые введенные пользователем географические локации с названиями университетов, а затем поместить данные на фактическую карту OpenStreetMap.

Чтобы начать работу, скачайте приложение здесь:

www.py4e.com/code3/opengeo.zip.

Первая проблема, которую необходимо решить, заключается в том, что эти API геокодирования ограничены определенным количеством запросов в день. Если у вас много данных, то, возможно, потребуется несколько раз остановить и перезапустить процесс поиска. Поэтому мы разделяем решение данной задачи на два этапа.

На первом этапе мы берем входные «опросные» данные из файла *where.data* и читаем их последовательно по одной строке за один прием, а затем извлекаем геокодированную информацию из сервиса Google и сохраняем ее в базе данных *geodata.sqlite*. Прежде чем использовать API геокодирования для каждой введенной пользователем локации, мы просто проверяем, есть ли у нас уже данные для этой конкретной строки ввода. База данных функционирует как локальный «кеш» данных геокодирования, чтобы мы не запрашивали у Google одни и те же данные дважды.

Этот процесс в любое время можно перезапустить «с нуля», если удалить файл базы данных *geodata.sqlite*.

Запустите программу *geoload.py*. Она будет читать строки ввода из файла *where.data* и для каждой строки проверять, содержится ли она уже в базе данных. Если у нас нет данных о заданной локации, то программа вызывает API геокодирования, чтобы извлечь требуемые данные и сохранить их в базе данных.

Рис. 16.1 ❖ Карта OpenStreetMap

Ниже приведен пример выполнения этой программы после того, как в базе данных уже были сохранены некоторые данные:

```
Found in database AGH University of Science and Technology

Found in database Academy of Fine Arts Warsaw Poland

Found in database American University in Cairo

Found in database Arizona State University

Found in database Athens Information Technology

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=BITS+Pilani
Retrieved 794 characters {"type":"FeatureColl

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=Babcock+University
Retrieved 760 characters {"type":"FeatureColl
```

```
Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=Banaras+Hindu+University
Retrieved 866 characters {"type":"FeatureColl
...
```

Первые пять локаций уже содержатся в базе данных, поэтому они пропускаются. Программа выполняет последовательный просмотр до того места, где обнаруживаются новые локации, и начинается извлечение данных о них.

Программу *geoload.py* можно остановить в любой момент времени. В ней существует счетчик, который можно использовать для ограничения числа обращений к API геокодирования при каждом запуске. С учетом того, что в файле *where.data* содержится всего лишь несколько сотен элементов данных, ограничение на число вызовов API геокодирования вряд ли будет превышено, но если данных будет значительно больше, то, вероятнее всего, потребуется несколько сеансов выполнения программы в течение нескольких дней, чтобы база данных собрала все данные геокодирования для исходных локаций.

После того как некоторые данные загружены в файл *geodata.sqlite*, можно выполнить их визуализацию, воспользовавшись для этого программой *geodump.py*. Эта программа считывает содержимое базы данных и записывает в файл *where.js* информацию о локации, широту и долготу в форме выполняемого кода JavaScript.

Результат выполнения программы *geodump.py* показан ниже:

```
AGH University of Science and Technology, Czarnowiejska,
Czarna Wieś, Krowodrza, Kraków, Lesser Poland
Voivodeship, 31-126, Poland 50.0657 19.91895

Academy of Fine Arts, Krakowskie Przedmieście,
Northern Śródmieście, Śródmieście, Warsaw, Masovian
Voivodeship, 00-046, Poland 52.239 21.0155
...
260 lines were written to where.js
Open the where.html file in a web browser to view the data.
```

Файл *where.html* состоит из кода HTML и JavaScript для визуализации карты Google. Этот код считывает самые свежие записи из файла *where.js*, чтобы получить данные, которые необходимо визуализировать. Ниже показан формат файла *where.js*:

```
myData = [
[50.0657,19.91895,
'AGH University of Science and Technology, Czarnowiejska,
Czarna Wieś, Krowodrza, Kraków, Lesser Poland
Voivodeship, 31-126, Poland '],
[52.239,21.0155,
'Academy of Fine Arts, Krakowskie Przedmieścieee,
Śródmieście Północne, Śródmieście, Warsaw,
Masovian Voivodeship, 00-046, Poland'],
...
];
```

Это переменная JavaScript, содержащая список списков. Синтаксис списковых констант JavaScript очень похож на Python, поэтому он должен быть хорошо знакомым для вас.

Чтобы увидеть все найденные локации, просто откройте файл *where.html* в любом браузере. Вы можете навести указатель мыши на каждую метку карты, чтобы найти местоположение, которое API геокодирования вернул для введенных пользователем данных. Если вы не видите никаких данных при открытии файла *where.html*, то можете проверить код JavaScript или перейти в консоль разработчика вашего браузера.

16.2. Визуализация сетей и сетевых соединений

В этом приложении мы будем выполнять некоторые функции механизма сетевого поиска. Сначала выполним глобальный поиск (spidering) в небольшом подмножестве веб-среды и запустим упрощенную версию алгоритма ранжирования страниц Google, чтобы определить, с какими страницами чаще всего устанавливается соединение, затем выполним визуализацию этого ранжирования страниц и частоты установления соединений в нашем маленьком уголке веб-среды. Мы воспользуемся библиотекой визуализации D3 JavaScript (https://d3js.org/) для создания вывода визуализированных данных.

Это приложение можно скачать здесь:

www.py4e.com/code3/pagerank.zip.

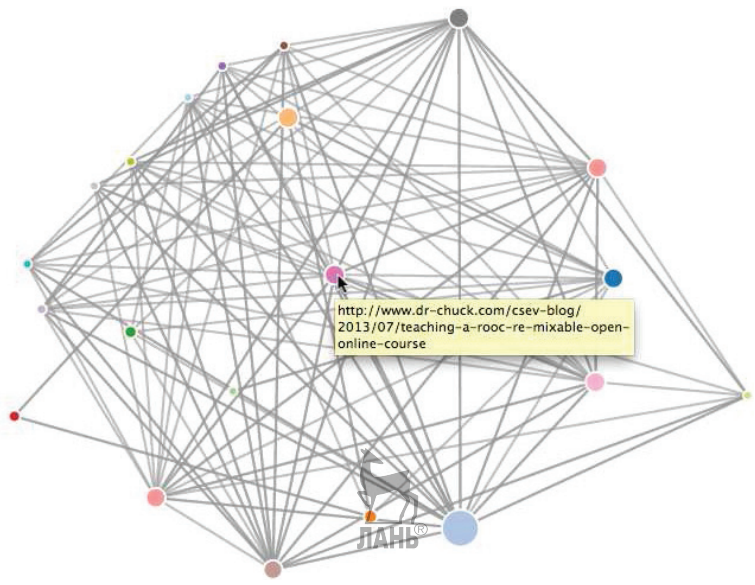


Рис. 16.2 ❖ Ранжирование веб-страниц

Первая программа *spider.py* медленно проходит по веб-сайту и затягивает последовательность страниц в базу данных *spider.sqlite*, записывая все ссылки (links) между страницами. В любой момент этот процесс можно перезапустить с самого начала, если удалить файл *spider.sqlite* и повторно выполнить программу *spider.py*.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

В этом примере выполнения мы приказали программе просканировать веб-сайт и получить две страницы. Если еще раз запустить программу и приказать просканировать большее число страниц, то она не будет повторно сканировать те страницы, которые уже содержатся в базе данных. После перезапуска программа переходит на случайную непросканированную страницу и начинает работу там. Таким образом, каждый последующий запуск программы *spider.py* является аддитивным (т. е. сохраняет и накапливает ранее найденные данные).

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

В одной базе данных может существовать несколько начальных точек – в программе они называются «паутинами» (webs). Программа-паук случайным образом выбирает среди всех непосещенных ссылок во всех паутинах следующую страницу для глобального поиска. Если вы хотите вывести содержимое файла *spider.sqlite*, то можете запустить программу *spdump.py*, результат выполнения которой показан ниже:

```
(5, None, 1.0, 3, 'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

Здесь показано число входящих ссылок, старый и новый ранги страницы, ее числовой идентификатор и URL. Программа *spdump.py* показывает только те страницы, которые имеют хотя бы одну входящую ссылку, указывающую на них.

После записи нескольких страниц в базу данных можно вычислить их ранг, используя программу *sprank.py*. Просто укажите, сколько итераций вычисления ранга страницы требуется выполнить.

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

Можно еще раз вывести содержимое базы данных, чтобы проверить, как изменились значения ранга страниц:

```
(5, 1.0, 0.985, 3, 'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

Вы можете выполнять программу *sprank.py* столько раз, сколько захотите, и она будет просто улучшать ранг страниц при каждом запуске. Можно даже запустить несколько раз *sprank.py*, а затем направить программу-паука *spider.py* на несколько новых страниц, потом снова выполнить sprank.py для обеспечения сходимости значений ранга страниц. Механизм сетевого поиска обычно выполняет программы глобального поиска и ранжирования страниц одновременно.

Если необходимо перезапустить с самого начала процесс вычисления ранга страниц без повторного сканирования всех веб-страниц, то можно воспользоваться программой *spreset.py*, а затем снова выполнить программу *sprank.py*.

```
How many iterations:50
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
42 0.000109076928206
43 9.91987599002e-05
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05
[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

На каждой итерации алгоритма ранжирования страницы выводится средняя величина изменения ранга каждой страницы. Изначально сеть не вполне сбалансирована, поэтому значения ранга отдельных страниц значительно изменяются на каждой итерации. Но за несколько коротких итераций ранг страницы сходится. Придется запускать программу *sprank.py* достаточно долго, чтобы добиться сходимости значений ранга страницы.

Если необходима визуализация веб-страниц с самым высоким рангом в текущий момент, то выполните программу *spjson.py* для чтения базы данных и записи данных для страниц с наибольшим числом ссылок на них в формате JSON, который можно просматривать в веб-браузере.

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

Эти данные можно увидеть, если открыть файл *force.html* в любом веб-браузере. Будет показана автоматически сформированная графическая схема узлов и ссылок. Можно щелкнуть и перетащить любой узел в другое место, а двойной щелчок по узлу выводит соответствующий ему URL.

Если вы повторно выполняли другие утилиты, то запустите еще раз программу *spjson.py* и нажмите кнопку **Refresh** (Обновить) в браузере для получения новых данных из файла *spider.json*.

16.3. Визуализация данных электронной почты

К этому моменту вы уже хорошо знакомы с файлами данных *mbox-short.txt* и *mbox.txt*. Пришло время перейти на новый уровень анализа данных электронной почты.

В реальной жизни иногда приходится загружать почтовые данные с серверов. Это может занимать некоторое время, а данные могут быть несогласованными, содержать множество ошибок и требовать значительной очистки или корректировки. В этом разделе мы будем работать с приложением, которое является самым сложным в данный момент. С помощью этого приложения мы извлекаем почти гигабайт данных и визуализируем их.

Приложение можно скачать здесь:

https://www.py4e.com/code3/gmane.zip.

Мы будем использовать данные из бесплатной службы архивирования списков рассылки под названием http://gmane.io. Этот сервис часто используется в проектах с открытым исходным кодом, потому что он предоставляет удобный архив электронной почты с возможностью поиска. Владельцы сервиса также проводят весьма либеральную политику в отношении доступа к своим данным через их API. Они не устанавливают ограничения по скорости, но просят не перегружать сервис и извлекать только те данные, которые вам действительно нужны.

Очень важно, чтобы вы разумно использовали данные gmane.io, добавляя задержки при доступе к их сервисам и распределяя долговременно выполняемые задания по более продолжительному интервалу времени. Не злоупотребляйте этой бесплатной услугой, не становитесь виновником ее недоступности для всех пользователей.

Когда данные электронной почты Sakai обрабатывались с помощью этого программного обеспечения, был получен почти гигабайт данных, и потре-

бовалось несколько запусков в течение нескольких дней. Файл *README.txt* в вышеупомянутом zip-архиве может содержать инструкции относительно того, как вы можете загрузить предварительно сформированную копию файла *content.sqlite* для большей части корпуса электронной почты Sakai, поэтому вам не придется выполнять глобальный поиск в течение пяти дней только лишь для того, чтобы запустить программы. Если вы загружаете предварительно сформированный контент, вам все равно придется запустить процесс глобального поиска, чтобы получить более свежие сообщения.

Рис. 16.3 ❖ Облако слов из списка Sakai Developer List

Первым шагом будет сканирование репозитория gmane. Базовый URL жестко закодирован в программе *gmane.py* и в списке разработчиков Sakai. Вы можете сканировать другой репозиторий, изменив в исходном коде этот базовый URL. Не забудьте удалить файл *content.sqlite*, если измените базовый URL.

Файл *gmane.py* работает как настоящая надежная программа-паук с кешированием в том плане, что работает она медленно и извлекает одно почтовое сообщение в секунду, чтобы избежать блокировки со стороны gmane. Программа хранит все свои данные в базе и может прерываться и перезапускаться настолько часто, насколько это необходимо. На скачивание всех данных может уйти много часов. Поэтому, возможно, потребуется многократный перезапуск этой программы.

Ниже показан результат выполнения программы *gmane.py* при извлечении последних пяти сообщений из списка разработчиков Sakai:

```
How many messages:10
http://download.gmane.io/gmane.comp.cms.sakai.devel/51410/51411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://download.gmane.io/gmane.comp.cms.sakai.devel/51411/51412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
http://download.gmane.io/gmane.comp.cms.sakai.devel/51412/51413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://download.gmane.io/gmane.comp.cms.sakai.devel/51413/51414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.io/gmane.comp.cms.sakai.devel/51414/51415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.io/gmane.comp.cms.sakai.devel/51415/51416 0

Does not start with From
```

Программа последовательно сканирует файл *content.sqlite* по одной записи до первого номера сообщения, в котором пока еще не выполнялся поиск, и начинает поиск в нем. Программа продолжает сканирование до тех пор, пока не получит желаемое количество сообщений или не достигнет страницы, которая не выглядит как правильно отформатированное сообщение.

Иногда на gmane.io некоторые сообщения отсутствуют. Возможно, администраторы могут удалять их, или сообщения теряются. Если ваша программа-паук останавливается и кажется, что она наткнулась на отсутствующее сообщение, то войдите в диспетчер SQLite и добавьте строку с пропущенным идентификатором, оставив все остальные поля пустыми, и перезапустите программу *gmane.py*. Это разблокирует процесс глобального поиска и позволит ему продолжиться. Такие пустые сообщения будут проигнорированы на следующем этапе процесса.

Хорошая новость: после того как вы проверили все сообщения и поместили их в базу данных *content.sqlite*, можно снова запускать программу *gmane.py*, чтобы получать новые сообщения по мере их отправки в список.

Данные в базе *content.sqlite* довольно сырые (необработанные), с неэффективной моделью данных и несжатые. Это сделано намеренно, поскольку такой подход позволяет просматривать содержимое *content.sqlite* в диспетчере SQLite для отладки и устранения проблем, связанных с процессом глобального поиска. Было бы плохой идеей направлять какие-либо запросы в эту базу данных, так как они будут выполняться довольно медленно.

Второй процесс – выполнение программы *gmodel.py*. Эта программа считывает необработанные данные из файла *content.sqlite* и создает очищенную и правильно смоделированную версию данных в файле *index.sqlite*. Этот файл будет намного меньше (часто в 10 раз меньше), чем файл *content.sqlite*, поскольку он сжимает и заголовок, и основной текст.

При каждом запуске *gmodel.py* эта программа удаляет и перестраивает файл *index.sqlite*, что позволяет настраивать его параметры и редактировать отображение таблиц в файле *content.sqlite*, чтобы настроить процесс очистки данных. Ниже показан пример результата выполнения программы *gmodel.py*. Она выводит строку при обработке каждого из 250 почтовых сообщений, чтобы можно было наблюдать за динамикой ее выполнения, так как эта

программа может работать достаточно долго, обрабатывая почти гигабайт почтовых данных.

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

Программа *gmodel.py* выполняет несколько задач по очистке данных.

Имена доменов усекаются до двух уровней для .com, .org, .edu и .net. Другие имена доменов усекаются до трех уровней. То есть `si.umich.edu` становится `umich.edu`, а `caret.cam.ac.uk` становится `cam.ac.uk`. Адреса электронной почты также принудительно переводятся в нижний регистр, а некоторые из адресов @gmane.io, подобные показанному ниже:

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.io
```

преобразовываются в реальный адрес всякий раз, когда есть соответствующий реальный адрес электронной почты в другом месте корпуса сообщений.

В базе данных *mapping.sqlite* есть две таблицы, которые позволяют отображать как доменные имена, так и отдельные адреса электронной почты, которые меняются за время существования списка рассылки. Например, Стив Гитенс (Steve Githens) использовал следующие адреса электронной почты, когда менял работу во время существования списка разработчиков Sakai:

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

Мы можем добавить две записи в таблицу `Mapping` в базе данных *mapping.sqlite*, чтобы программа *gmodel.py* отображала все три адреса в один:

```
s-githens@northwestern.edu -> swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

Также можно добавить аналогичные записи в таблицу `DNSMapping`, если существует несколько DNS-имен, которые необходимо отобразить в одно DNS-имя. Показанное ниже отображение было добавлено в данные Sakai:

```
iupui.edu -> indiana.edu
```

чтобы все аккаунты из различных кампусов университета Индианы отслеживались вместе.

Можно многократно запускать программу *gmodel.py* для просмотра данных и добавлять отображения, чтобы данные становились все более чистыми. После завершения работы вы получите хорошо проиндексированную версию списка адресов электронной почты в файле *index.sqlite*. Именно этот файл нужно использовать для анализа данных. С этим файлом анализ данных будет выполнен действительно быстро.

Во-первых, простейший анализ данных заключается в определении: «кто отправил наибольшее количество сообщений?» и «какая организация отправила наибольшее количество сообщений?». Это делается с помощью программы *gbasic.py*:

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584

Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184

Top 5 Email list organizations
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

Обратите внимание, насколько быстрее выполняется программа *gbasic.py* по сравнению с *gmane.py* или даже *gmodel.py*. Все они работают с одними и теми же данными, но *gbasic.py* использует сжатые и нормализованные данные из файла *index.sqlite*. Если вам нужно обрабатывать большой объем данных, то многоэтапный процесс, подобный тому, что представлен в этом приложении, может занять немного больше времени при разработке, но позволит сэкономить весьма много времени, когда вы действительно начнете исследовать и визуализировать свои реальные данные.

Можно выполнить простую визуализацию частотности слов в строках темы сообщений с помощью программы *gword.py*:

```
Range of counts: 33229 129
Output written to gword.js
```

В результате создается файл *gword.js*, который можно визуализировать, используя файл *gword.html* для создания облака слов, похожего на показанное в начале этого раздела на рис. 16.3.

Визуализация второго типа создается программой *gline.py*. Она вычисляет степень участия организаций в процессе обмена сообщениями электронной почты за определенный интервал времени.

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

Вывод этой программы записывается в файл *gline.js*, который визуализируется с использованием файла *gline.htm*.

Это достаточно сложное и интеллектуальное приложение, обладающее функциональными возможностями для выполнения некоторых операций по извлечению, очистке и визуализации реальных данных.

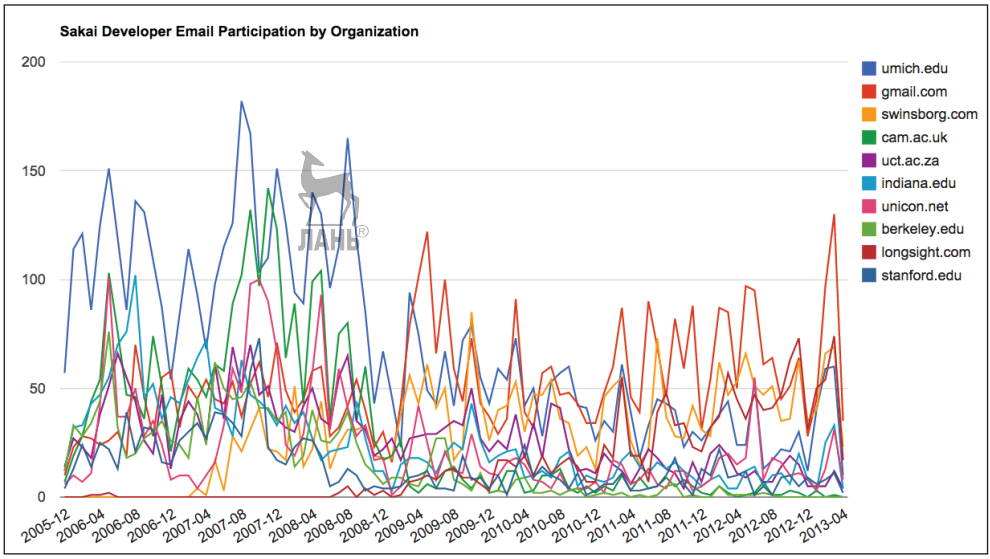


Рис. 16.4 ❖ Деятельность по обмену сообщениями электронной почты в списке разработчиков Sakai по отдельным организациям

Приложение A

Участники проекта

A.1. Список участников проекта «Python for Everybody» («Python для всех»)

Анджей Войтович (AndrzejWójtowicz), Элиотт Хаузер (Elliott Hauser), Стивен Катто (Stephen Catto), Сью Блуменберг (Sue Blumenberg), Тамара Браннок (Tamara Brunnock), Михаэла Мак (Mihaela Mack), Крис Колосивски (Chris Kolosiwsky), Дастин Фарли (Dustin Farley), Йенс Леерссен (Jens Leerssen), Навин КТ (Naveen КТ), Мирза Ибрахимович (Mirza Ibrahimovic), Навин (Naveen) (@togarnk), Чжоу Фаньги (Zhou Fangyi), Алистер Уолш (Alistair Walsh), Эрика Броуди (Erica Brody), Цзи-Шен Хуан (Jih-Sheng Huang), Луис Лонгкесорн (Louis Luangkesorn) и Майкл Фьюдж (Michael Fudge).

Более подробную информацию об участниках этого проекта можно найти здесь:

https://github.com/csev/py4e/graphs/contributors.

A.2. Список участников проекта «Python for Informatics»

Брюс Шилдс (Bruce Shields) – особая благодарность за редактирование ранних черновиков, Сара Хегге (Sarah Hegge), Стивен Черри (Steven Cherry), Сара Кэтлин Барбароу (Sarah Kathleen Barbarow), Андреа Паркер (Andrea Parker), Радафат Чонгтаммакун (Radaphat Chongthammakun), Меган Хиксон (Megan Hixon), Кирби Арнер (Kirby Urner), Кэти Куйала (Katie Kujala), Ноа Ботимер (Noah Botimer), Эмили Алиндер (Emily Alinder), Марк Томпсон-Кьюлар (Mark Thompson-Kular), Джеймс Перри (James Perry), Эрик Хофер (Eric Hofer), Эйтан Адар (Eytan Adar), Питер Робинсон (Peter Robinson), Дебора Дж. Нелсон (Deborah J. Nelson), Джонатан Энтони (Jonathan C. Anthony), Эден Рассетте (Eden Rassette), Джанетт Шрёдер (Jeannette Schroeder), Джастин Физелл (Justin Feezell), Чуань-Ци Ли (Chuanqi Li), Джералд Гординье (Gerald Gordinier),

Гэвин Томас Штрассель (Gavin Thomas Strassel), Райан Клемент (Ryan Clement), Алисса Толли (Alissa Talley), Кэтлин Холман (Caitlin Holman), Юн-Ми Ким (Yong-Mi Kim), Карен Стоувер (Karen Stover), Шери Эдмондс (Cherie Edmonds), Мария Зайферле (Maria Seiferle), Ромер Кристи Д. Аранас (Romer Kristi D. Aranas) (RK), Грант Бойер (Grant Boyer), Эдемари Дюссан (Hedemarrie Dussan).

A.3. Предисловие к книге «Think Python»

A.3.1. Странная история книги «Think Python»

(Аллен Б. Дауни (Allen B. Downey))

В январе 1999 г. я готовился читать вводный курс программирования на Java. Я преподавал этот курс трижды, и меня это расстраивало. Процент неуспевающих на курсе был слишком высоким, и даже для успевающих студентов общий уровень подготовки был слишком низким.

Одной из проблем, которые я наблюдал, были книги. Они были слишком большими, в них было слишком много ненужных подробностей о Java и не хватало высокоуровневых руководств непосредственно по программированию. И все книги страдали одним и тем же недостатком – так называемым «эффектом аварийного люка»: в начале было легко, постепенно нарастало напряжение, а потом приблизительно около главы 5 внезапно проваливалось дно. Студенты получали чрезмерно много нового материала слишком быстро, и я тратил остаток семестра на «сбор осколков».

За две недели до первого дня курса я решил написать собственную книгу и поставил перед собой следующие цели:

- сохранение краткости. Для студентов лучше прочесть 10 страниц, чем не прочесть 50;
- внимательное отношение к терминологии. Я старался свести к минимуму профессиональный жаргон и определять каждый термин при его первом использовании;
- постепенный рост сложности. Чтобы избежать «эффекта аварийного люка», самые сложные темы я разделил на последовательность небольших этапов (шагов);
- сосредоточение на программировании, а не на языке программирования. В книгу я включил минимально возможное, но самое полезное подмножество языка Java, а остальное исключил.

Книге необходимо название, поэтому я по какому-то мимолетному порыву выбрал «How to Think Like a Computer Scientist» («Как научиться мыслить подобно специалисту по информатике»).

Моя первая версия была не самой удачной, но она работала. Студенты читали и понимали достаточно, чтобы я мог проводить время в аудитории, обсуждая сложные и интересные темы и (что наиболее важно) позволяя студентам получать практический опыт.

Я выпустил книгу под лицензией GNU Free Documentation License, которая позволяет пользователям копировать, изменять и распространять книгу.

Потом произошло потрясающее событие. Джефф Элкнер (Jeff Elkner), учитель средней школы из Вирджинии, отредактировал мою книгу и перевел ее на Python. Он прислал мне копию своей редакции, и у меня был весьма необычный опыт изучения Python – чтение собственной книги.

Джефф и я отредактировали книгу, включив в нее учебный пример Криса Мейерса (Chris Meyers), и в 2001 г. мы выпустили новую версию книги с названием «How to Think Like a Computer Scientist: Learning with Python» («Как научиться думать подобно специалисту по информатике: обучение с помощью Python»), также под лицензией GNU Free Documentation License. Под маркой Green Tea Press я опубликовал книгу и начал продавать бумажные копии через Amazon.com и книжные магазины колледжей. Другие книги от Green Tea Press доступны на сайте https://greenteapress.com/wp/.

В 2003 г. я начал работу в колледже Олин (Olin Colledge) и впервые стал преподавать Python. Контраст с Java был разительным. Студенты меньше боролись с трудностями, больше учились, работали над более интересными проектами и в целом получали гораздо больше удовольствия.

В течение последних пяти лет я продолжал дорабатывать книгу, исправляя ошибки, улучшая некоторые примеры и добавляя материал, особенно практические упражнения. В 2008 г. я начал серьезную переработку книги – в то же время со мной связался редактор издательства Cambridge University Press, который был заинтересован в публикации следующего издания. Как раз вовремя!

Я надеюсь, что вам понравится работать с этой книгой и что она поможет вам научиться программировать и мыслить, по крайней мере, хотя бы немного как специалист по информатике.

A.3.2. Благодарности за работу над «Think Python»

(Аллен Б. Дауни (Allen B. Downey))

Прежде всего, что наиболее важно, я благодарю Джеффа Элкнера (Jeff Elkner), который перевел мою книгу с Java на Python, с чего и начался этот проект, который познакомил меня с языком программирования, почти сразу ставшим моим любимым языком.

Я также благодарю Криса Мейерса (Chris Meyers), написавшего несколько разделов для книги «How to Think Like a Computer Scientist» («Как научиться мыслить подобно специалисту по информатике»).

И я благодарен Фонду свободного программного обеспечения (Free Software Foundation) за разработку лицензии свободной документации GNU, которая помогла мне сделать возможным мое сотрудничество с Джеффом и Крисом.

Я также благодарю редакторов в Lulu, которые работали над книгой «How to Think Like a Computer Scientist» («Как научиться мыслить подобно специалисту по информатике»).

Я благодарю всех студентов, которые работали с более ранними версиями этой книги, и всех участников (перечисленных в приложении), которые прислали исправления и предложения.

И я благодарю мою жену Лайзу (Lisa) за ее работу над этой книгой, за Green Tea Press, а также за все остальное.

Аллен Б. Дауни (Allen B. Downey)
Needham MA

Аллен Дауни – адъюнкт-профессор информатики в инженерном колледже Франклина В. Олина (Franklin W. Olin College of Engineering).

A.4. Список участников проекта «Think Python»

Аллен Б. Дауни (Allen B. Downey)

Более 100 проницательных и вдумчивых читателей прислали предложения и исправления за последние несколько лет. Их вклад и энтузиазм в работе над этим проектом оказали огромную помощь.

Более подробную информацию о конкретном вкладе каждого из этих людей см. в тексте книги «Think Python».

Ллойд Хью Аллен (Lloyd Hugh Allen), Ивон Булианн (Yvon Boulianne), Фред Бреммер (Fred Bremmer), Йона Коэн (Jonah Cohen), Майкл Конлон (Michael Conlon), Бенуа Жирар (Benoit Girard), Куртни Глисон (Courtney Gleason) и Катрин Смит (Katherine Smith), Ли Харр (Lee Harr), Джеймс Кэйлин (James Kaylin), Дэвид Кершоу (David Kershaw), Эдди Лэм (Eddie Lam), Мань-Юн Ли (Man-Yong Lee), Давид Майо (David Mayo), Крис МакЭлун (Chris McAloon), Мэтью Дж. Мёльтер (Matthew J. Moelter), Саймон Дикон Монфорд (Simon Dicon Montford), Джон Оутцтс (John Ouzts), Кевин Паркс (Kevin Parks), Дэвид Пул (David Pool), Майкл Шмитт (Michael Schmitt), Робин Шоу (Robin Shaw), Пол Слей (Paul Sleigh), Крейг Т. Снайдал (Craig T. Snydal), Иэн Томас (Ian Thomas), Кейт Верхейден (Keith Verheyden), Питер Уинстэнли (Peter Winstanley), Крис Вробель (Chris Wrobel), Моше Задка (Moshe Zadka), Кристоф Цвершке (Christoph Zwerschke), Джеймс Майер (James Mayer), Хейден МакАфи (Hayden McAfee), Энджел Арнал (Angel Arnal), Таухидул Хоке (Tauhidul Hoque) и Лекс Бережны (Lex Berezhny), др. Мишель Альсетта (Dr. Michele Alzetta), Энди Митчелл (Andy Mitchell), Кэлин Харви (Kalin Harvey), Кристофер П. Смит (Christopher P. Smith), Дэвид Хатчинс (David Hutchins), Грегор Лингл (Gregor Lingl), Джули Питерс (Julie Peters), Флорин Оприна (Florin Oprina), Д. Дж. Вебре (D. J. Webre), Кен (Ken), Иво Уивер (Ivo Wever), Кёртис Янко (Curtis Yanko), Бен Логан (Ben Logan), Джейсон Армстронг (Jason Armstrong), Луи Кордье (Louis Cordier), Брайан Кейн (Brian Cain), Роб Блэк (Rob Black), Жан-Филипп Рей (Jean-Philippe Rey) из Ecole Centrale Paris, Джейсон Мадер (Jason Mader) из George Washington University, Иан Гунтдтофте-Бруун (Jan Gundtofte-Bruun), Абель

Дэвид (Abel David) и Алексис Динно (Alexis Dinno), Чарльз Тайер (Charles Thayer), Роджер Сперберг (Roger Sperberg), Сэм Булл (Sam Bull), Эндрю Чжуан (Andrew Cheung), Кори Капел (C. Corey Capel), Алессандра (Alessandra), Вим Шампань (Wim Champagne), Дуглас Райт (Douglas Wright), Джаред Спиндор (Jared Spindor), Линь Пэй-Хэн (Lin Peiheng), Рей Хагтведт (Ray Hagtvedt), Торстен Хюбш (Torsten Hübsch), Инга Петухов (Inga Petuhhov), Арне Бабенхаузерхайде (Arne Babenhauserheide), Марк Е. Касида (Mark E. Casida), Скотт Тайлер (Scott Tyler), Гордон Шеферд (Gordon Shepherd), Эндрю Тёрнер (Andrew Turner), Адам Хобарт (Adam Hobart), Дэрил Хэммонд (Daryl Hammond) и Сара Циммерман (Sarah Zimmerman), Джордж Сасс (George Sass), Брайан Бингхэм (Brian Bingham), Леа Энгельберт-Фентон (Leah Engelbert-Fenton), Джо Функе (Joe Funke), Чао-Чао Чень (Chao-chao Chen), Джефф Пэйн (Jeff Paine), Любош Пинтес (Lubos Pintes), Грег Линд (Gregg Lind) и Эбигейл Хейтхофф (Abigail Heithoff), Макс Хэйлперин (Max Hailperin), Чотипат Порнавалай (Chotipat Pornavalai), Станислав Антол (Stanislaw Antol), Эрик Пашман (Eric Pashman), Мигель Асеведо (Miguel Azevedo), Цзянь-Хуа Лю (Jianhua Liu), Ник Кинг (Nick King), Мартин Цутер (Martin Zuther), Адам Циммерман (Adam Zimmerman), Ратнакар Тивари (Ratnakar Tiwari), Анураг Гёль (Anurag Goel), Келли Кратцер (Kelli Kratzer), Марк Гриффитс (Mark Griffiths), Ройдан Онжи (Roydan Ongie), Патрик Воловиец (Patryk Wolowiec), Марк Чонофски (Mark Chonofsky), Рассел Коулман (Russell Coleman), Вэй Хуан (Wei Huang), Карен Барбер (Karen Barber), Нам Нгуен (Nam Nguyen), Стефани Морен (Stéphane Morin), Фернандо Тардио (Fernando Tardio) и Пол Ступ (Paul Stoop).

Приложение B

Подробная информация о защите авторского права

Я бы предпочел лицензировать книгу под менее строгой лицензией CC-BY-SA. Но, к сожалению, есть несколько недобросовестных организаций, которые находят свободно лицензированные книги, а затем публикуют и продают практически неизмененные копии таких книг на сервисах печати по запросу, таких как LuLu или KDP. Сервис KDP (к счастью) добавил политику, которая оказывает предпочтение пожеланиям фактического правообладателя перед теми, кто не владеет авторскими правами, но пытается опубликовать свободно лицензируемое произведение. К сожалению, существует множество сервисов печати по запросу, и очень немногие из них так же хорошо продумали политику, как сервис KDP.

Я был вынужден добавить элемент NC в лицензию на эту книгу, чтобы иметь возможность обратиться в судебные органы в том случае, если кто-то попытается копировать эту книгу и продавать ее на коммерческой основе. К сожалению, добавление NC ограничивает такое использование этого материала, которое я хотел бы разрешить. Поэтому я добавил особый раздел документа, чтобы описать конкретные случаи, в которых я заранее даю свое разрешение на использование материала из этой книги в ситуациях, которые некоторые могут счесть коммерческими.

- Если вы распечатываете ограниченное количество копий всей или части этой книги для использования в учебном курсе (например, как учебный пакет), то для этой цели вам предоставляется лицензия CC-BY на эти материалы.
- Если вы преподаватель в университете и переводите эту книгу на язык, отличный от английского, и преподаете, используя переведенную книгу, то вы можете связаться со мной, и я предоставлю вам лицензию CC-BY-SA на эти материалы с учетом публикации вашего перевода. В частности, вам будет разрешено продавать переведенную книгу на коммерческой основе.

Если вы собираетесь перевести книгу, то, возможно, пожелаете связаться со мной, чтобы мы совместно могли убедиться, что у вас есть все необходимые материалы курса, и вы также могли их перевести.

Разумеется, вы можете связаться со мной и попросить разрешения, если перечисленных выше пунктов недостаточно. Во всех случаях разрешение на повторное использование и переработку этого материала будет предоставляться при условии, что существует явная добавленная стоимость или выгода для студентов или преподавателей, которая возникает в результате новой работы.

Чарльз Северанс (Charles Severance)
www.dr-chuck.com
Ann Arbor, MI, USA
9 сентября 2013 г.

Предметный указатель

Р

С

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Чарльз Р. Северанс

Python для всех

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Счастин А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 21,29. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Обработка данных с использованием Python 3

Python — простой в изучении и практическом использовании язык программирования, который свободно доступен на компьютерах под управлением Mac OS, Windows и Linux. Изучив Python, вы сможете использовать его в своей профессиональной деятельности, не покупая какое-либо программное обеспечение.

Данная книга представляет собой курс программирования на языке Python. Краткий и четкий стиль изложения и многочисленные упражнения позволят достаточно быстро овладеть основными навыками программирования и методами обработки данных.

В числе рассматриваемых тем:

- базовые свойства языка Python;
- использование простых и составных структур данных;
- взаимодействия с базами данных;
- часто применяемые алгоритмы и шаблоны программирования;
- отладка программ (обнаружение и исправление ошибок).

Издание предначено широкому кругу читателей, которые не являются программистами и хотели бы освоить язык Python с нуля.

Чарльз Северанс (www.dr-chuck.com) — доцент Школы информации при Мичиганском университете; также преподавал информатику в Университете штата Мичиган. Активно сотрудничает с открытыми образовательными платформами, ведет ряд бесплатных курсов по Python и веб-технологиям на Coursera. Работал в должности исполнительного директора Фонда Sakai и главного архитектора Проекта Sakai (www.sakaiproject.org).

ЛАНЬ®

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
издательство
www.дмк.рф

ISBN 978-5-93700-104-7

9 785937 001047 >