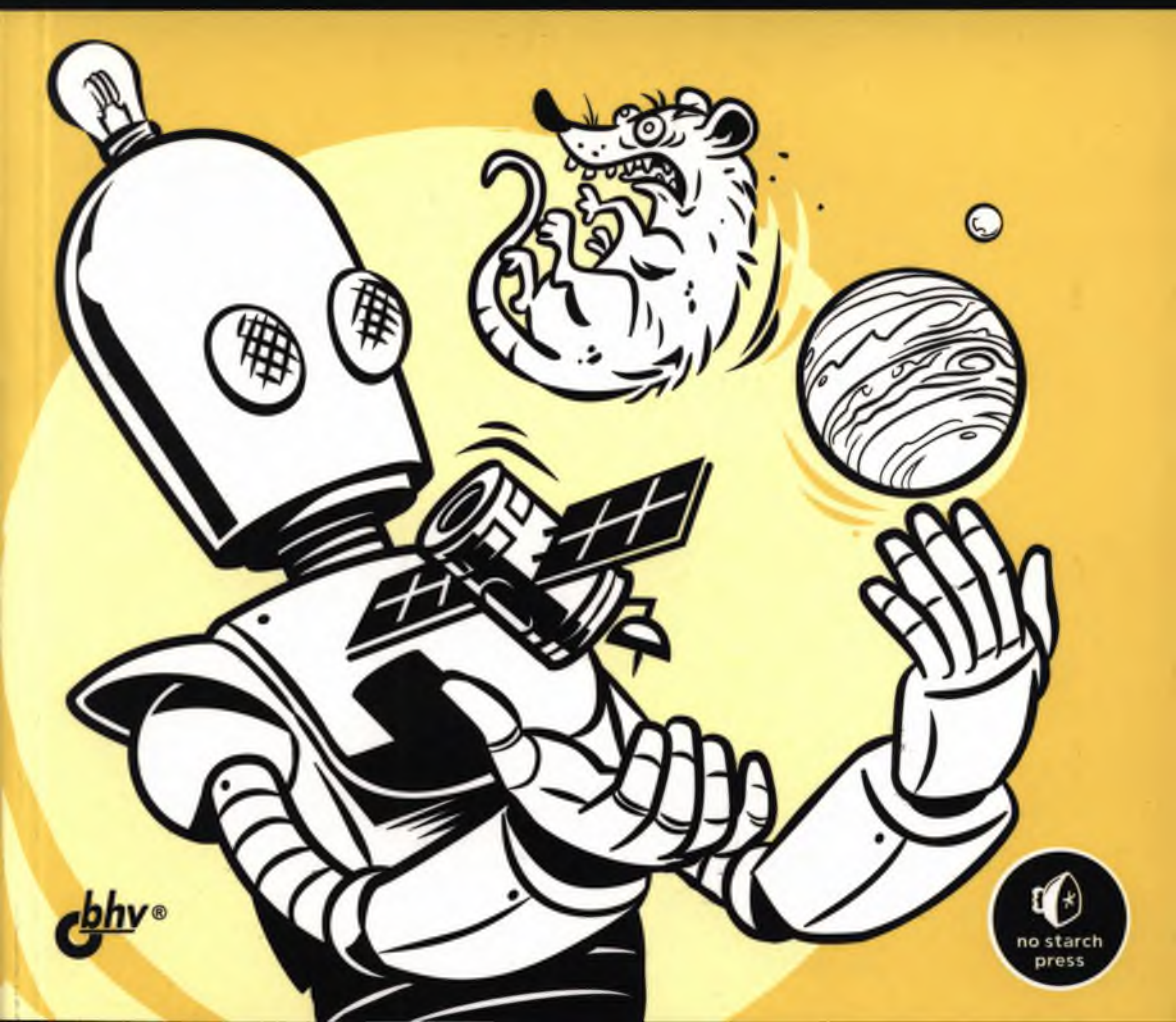
«НЕПРАКТИЧНЫЙ» PYTHON

занимательные проекты для тех, **кто хочет поумнеть**

ЛИ ВОГАН



bhv®

no starch press

IMPRACTICAL PYTHON PROJECTS

IMPRACTICAL PYTHON PROJECTS

Playful Programming Activities to Make You Smarter

by Lee Vaughan

no starch press

San Francisco

ЛИ ВОГАН

«НЕПРАКТИЧНЫЙ» PYTHON

занимательные проекты для тех, кто хочет поумнеть

Санкт-Петербург
«БХВ-Петербург»
2021

УДК 004.438 Python
ББК 32.973.26-018.1
В61

Воган Л.

В61 «Непрактичный» Python: занимательные проекты для тех, кто хочет поумнеть: Пер. с англ. — СПб.: БХВ-Петербург, 2021. — 464 с.: ил.

ISBN 978-5-9775-6751-0

Книга поможет читателям, самостоятельно осваивающим язык Python, отточить уже имеющиеся навыки программирования, выработать новые и получить удовольствие от этого процесса. Ее можно рассматривать как свою вторую книгу по языку Python. Книга организована так, чтобы дополнять обычный учебник для начинающих или вводный курс. Для этого применен проектный подход, позволяющий обойтись без тщательного "пережевывания" материла, который вы уже изучили.

По мере работы над проектами читатели будут использовать анализ марковских цепей для написания хокку, метод Монте-Карло для моделирования финансовых рынков, наложение снимков для улучшения астрофотосъемки и генетические алгоритмы для разведения армии гигантских крыс. В итоге читатели получат практический опыт работы с такими модулями, как pygame, Pylint, pydocstyle, tkinter, python-docx, matplotlib и pillow.

Для программистов на языке Python

УДК 004.438 Python
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Перевод с английского	*Андрея Логунова*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 01 12 20
Формат 70×100$^{1}/_{16}$ Печать офсетная. Усл. печ. л. 37,41
Тираж 1500 экз. Заказ №12630
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-59327-890-8 (англ.)
ISBN 978-5-9775-6751-0 (рус.)

ОГЛАВЛЕНИЕ

Для случайных программистов,
преданных своему делу неспециалистов,
копателей и дилетантов, как то:
всех непрофессионалов, которые
сами пишут свой код каждый день.
Да пусть она поможет вам на вашем пути

ОБ АВТОРЕ

Ли Воган (Lee Vaughan) — геолог с более чем 30-летним опытом работы в нефтяной промышленности. Будучи старшим техническим специалистом по геологическому моделированию в крупной международной нефтяной компании, он принимал участие в создании и обзоре компьютерных моделей, разработке, тестировании и коммерциализации программного обеспечения, а также в подготовке геологов и инженеров. Защитник непрограммистов, которым приходится использовать программирование в своей карьере, он написал эту книгу для того, чтобы помочь тем, кто самостоятельно оттачивает свои навыки в языке Python.

О ТЕХНИЧЕСКОМ РЕЦЕНЗЕНТЕ

Джереми Кун (Jeremy Kun) окончил Университет штата Иллинойс в Чикаго с докторской степенью по математике. Он ведет блог "Математика ∩ Программирование" (Math ∩ Programming, **https://jeremykun.com/**) и в настоящее время работает над оптимизацией центра обработки данных в Google.

БЛАГОДАРНОСТИ

Написание книги — семейное дело, и я не смог бы добиться успеха без поддержки со стороны моей настоящей семьи, так и со стороны моей "второй" семьи — единомышленников из издательства No Starch Press. Прежде всего, спасибо моей жене Ханне и дочерям Саре и Лоре за их понимание, терпение и бесконечную помощь в редактировании.

Спасибо Биллу Поллоку и Тайлеру Ортману за то, что они приняли мое предложение; Заку Лебовски за понимание того, что я пытался сказать; Джанель Людовиз за высокопрофессиональную работу по производственному редактированию; Рэйчел Монаган и Пауле Флеминг за то, что взяли на себя трудную работу по копированию и технической проверке книги; Дэвиду Ван Нессу за композицию и Серене Янг и Джошу Эллингсону за потрясающий дизайн обложки. Спасибо также моим техническим рецензентам Джереми Куну, Майклу Контравеосу и Микеле Пратусевичу за значительное улучшение книги с бесценными предложениями и исправлениями.

Сара Воган, Эрик Эвенчик, Сяо-Хуэй Ву, Брукс Кларк, Брайан Проетт, Брент Фрэнсис и Гленн Крум оказали значительную техническую поддержку.

Наконец, спасибо Марку Натерну за то, что познакомил меня с языком программирования Python, и в первую очередь Гвидо ван Россуму за то, что он изобрел эту замечательную вещь!

ВВЕДЕНИЕ

Добро пожаловать в книгу "“Непрактичный” Python: занимательные проекты для тех, кто хочет поумнеть"! Здесь вы будете использовать язык программирования Python для изучения Марса, Юпитера и самых дальних уголков галактики, сердец поэтов, мира высоких финансов, преступного мира шпионажа и фальсификации голосов на выборах, надувательства игровых шоу и многого другого. Вы будете применять такие методы, как анализ марковских цепей для написания хокку, симулирование Монте-Карло для моделирования финансовых рынков, наложение снимков для улучшения вашей астрофотосъемки и генетические алгоритмы для разведения армии гигантских крыс — все это при получении опыта работы с такими модулями, как `pygame`, `Pylint`, `pydocstyle`, `tkinter`, `python-docx`, `matplotlib` и `pillow`. И самое главное, вы будете получать удовольствие.

Для кого предназначена эта книга

Эту книгу можно рассматривать как свою вторую книгу по языку Python. Она организована так, чтобы следовать и дополнять либо полноценную книгу для начинающих, либо вводный курс. Вы сможете продолжить самоподготовку, используя проектный подход, не тратя свои деньги (или не занимая место на полке) на тщательное "пережевывание" материла, который вы уже изучили. Однако не волнуйтесь, я не оставлю вас один на один с программами; весь исходный код аннотирован и объяснен.

Приведенные в книге проекты предназначены для тех, кто хочет применять программирование для проведения экспериментов, проверки теорий, симулирования природных явлений или просто для того, чтобы весело провести время. Это относится ко всем тем, кто использует программирование как часть своей работы (например, исследователям и инженерам), но которые не являются программистами как таковыми, а также тем, кого я называю "непреклонными неспециалистами", — дилетантам и любителям, которым нравятся программистские задачи как забавное времяпрепровождение. Если вы хотели бы поиграть с представленными здесь идеями, но вас обескураживает запуск потенциально сложных проектов с нуля либо он требует много времени, то эта книга — для вас.

Что эта книга содержит

По мере работы над проектами вы будете расширять свои знания о полезных библиотеках и модулях языка Python; узнаете больше кратких форм, встроенных функций и полезных способов и попрактикуетесь в проектировании, тестировании и оптимизации программ. Кроме того, вы сможете связать то, что вы делаете, с реальными приложениями, наборами данных и задачами.

Процитирую Ральфа Уолдо Эмерсона: "Ничто великое никогда не достигалось без энтузиазма". Сюда входит и опыт самостоятельного освоения. Конечная цель этой книги — зажечь ваше воображение и подвести вас к разработке собственных интересных проектов. Не волнуйтесь, если на первый взгляд они покажутся слишком амбициозными; чуть-чуть усердия и много гугления способны сотворить чудеса — и быстрее, чем вы думаете.

Ниже приведен обзор глав этой книги. Вам не обязательно нужно работать с ними последовательно, но самые простые проекты находятся в начале, и я объясняю новые концепции, модули и методы подробнее, когда они вводятся впервые.

Глава 1 "Генератор дурацких имен". В этом разминочном проекте представлены рекомендации PEP 8 и PEP 257 по написанию программного кода на языке Python, а также модули `Pylint` и `pydocstyle`, которые анализируют соответствие вашего кода этим рекомендациям. Конечный продукт — это генератор глупых имен, вдохновленный сериалом "Ясновидец" (Psych) американского кабельного телеканала USA Network.

Глава 2 "Поиск палиндраммных заклинаний". Вы научитесь профилировать свой код и спасете волшебницу Затанну из комиксов DC Comics от мучительной смерти. Вы отыщете волшебные палиндраммы в онлайновых словарях, которые нужны Затанне для того, чтобы победить злодея, обращающего время вспять.

Глава 3 "Решение анаграмм". Вы напишете программу, которая поможет пользователю создавать анаграмму фразы из своего имени; например, имя Клинт Иствуд (Clint Eastwood) дает "действие старого Запада" (old west action). Затем, применив лингвистическое решето, вы поможете Тому Марволо Реддлу (Tom Marvolo Riddle) вывести свою анаграмму "Я — Лорд Волдеморт" (I am Lord Voldemort).

Глава 4 "Декодирование шифров времен Гражданской войны в США". Вы разведаете и взломаете один из самых успешных военных шифров в истории, маршрутный перестановочный шифр союзников. Затем поможете шпионам с обеих сторон отправлять и расшифровывать секретные сообщения, используя зигзагообразный шифр.

Глава 5 "Кодирование шифров времен Гражданской войны в Англии". Вы прочитаете сообщение, скрытое у всех на виду, расшифровав потайной шифр времен Гражданской войны в Англии. Затем спасете голову Марии, королевы Шотландии, разработав и реализовав код для выполнения более сложной задачи написания нулевого шифра.

Глава 6 "Написание текста невидимыми чернилами". Вы поможете корпоративному "кроту" предать отца Шерлока Холмса и избежать обнаружения с помо-

щью невидимых электронных чернил. Эта глава основана на эпизоде телевизионного сериала "Элементарно" (Elementary) американского телеканала CBS.

Глава 7 "Разведение гигантских крыс с помощью генетических алгоритмов". Вы примените генетические алгоритмы, обусловленные эволюцией Дарвина, для того, чтобы развести расу суперкрыс размером с самку бульмастифа. Затем поможете Джеймсу Бонду в мгновение ока взломать сейф с шифром на 10 млрд возможных комбинаций.

Глава 8 "Подсчет слогов в стихотворениях хокку". Вы научите свой компьютер подсчитывать слоги на английском языке в качестве прелюдии к написанию японских стихотворений в стиле хокку в следующей главе.

Глава 9 "Написание стихотворений хокку с помощью анализа марковских цепей". Вы научите свой компьютер писать хокку, объединив модуль подсчета слогов из *главы 8* с анализом цепи Маркова и тренировочным корпусом из нескольких сотен древних и современных хокку.

Глава 10 "Мы одни? Разведывание парадокса Ферми". Вы изучите вопрос, как выяснить наличие или отсутствие инопланетных радиосигналов, используя уравнение Дрейка, размеры галактики Млечного Пути и допущения о размере обнаруживаемых "пузырей излучения". Вы изучите и примените популярный модуль `tkinter` для создания графического изображения галактики и собственного радиопузыря Земли.

Глава 11 "Задача Монти Холла". Вы поможете умнейшей в мире женщине выиграть спор о задаче Монти Холла. Затем вы примените объектно-ориентированное программирование (ООП) для того, чтобы построить версию знаменитой игры в Монти с забавным графическим интерфейсом.

Глава 12 "Обеспечение сохранности пенсионных сбережений". Вы спланируете свою (или ваших родителей) обеспеченную жизнь на пенсии с использованием финансовой симуляции Монте-Карло.

Глава 13 "Симуляция инопланетного вулкана". Вы примените библиотеку `pygame` для симуляции извержения вулкана на Ио, одной из лун Юпитера.

Глава 14 "Картографирование Марса с помощью орбитального спутника". Вы построите аркадную игру на основе гравитации и подтолкнете спутник на круговую орбиту картографирования без исчерпания топлива или сгорания в атмосфере. Вы выведите на экран показания ключевых параметров, отследите орбитальные траектории, добавите тени планеты и медленное вращение Марса вокруг своей оси, все это время изучая орбитальную механику!

Глава 15 "Улучшение астрофотосъемки с помощью наложения снимков планеты". Вы откроете облачные полосы Юпитера и Большое красное пятно, оптически совместив некачественные видеоснимки с помощью библиотеки обработки изображений Python. Вы узнаете, как работать с файлами, папками и каталожными путями с помощью встроенных модулей `os` и `shutil`.

Глава 16 "Выявление мошенничества с помощью закона Бенфорда". Вы примените закон Бенфорда для расследования случаев фальсификации во время голо-

сования на президентских выборах 2016 года в США. Вы задействуете библиотеку `matplotlib` для подытоживания результатов на графике.

Каждая глава заканчивается по крайней мере одним практическим или одним сложным проектом. Каждый практический проект сопровождается решением. Это вовсе не означает, что указанное решение является лучшим, — вы можете придумать оптимальное решение самостоятельно, так что не забегайте вперед!

Однако со сложными проектами вы действительно предоставлены сами себе. Когда в 1519 г. Кортес вторгся в Мексику, он сжег свои каравеллы. Это было сделано для того, чтобы его конкистадоры поняли, что пути назад нет и им придется столкнуться с ацтеками, имея лишь мрачную и непоколебимую решимость. Таким образом, английское выражение "сжечь свои корабли" стало означать безоглядность или полную приверженность делу. Именно так вы должны решать сложные проекты — как если бы ваш корабль был сожжен, — и если вы его выполните, то из этих упражнений вы, скорее всего, узнаете больше, чем из любой другой части книги!

Версия Python, платформа и интегрированная среда разработчика

Каждый проект в этой книге построен с помощью языка Python версии 3.5 в среде операционной системы Microsoft Windows 10. Нет никаких проблем, если вы используете другую операционную систему: где это уместно, для других платформ предлагаются совместимые модули.

Используемые в этой книге примеры кода и снимки экрана взяты либо из текстового редактора Python IDLE, либо из интерактивной оболочки. Среда IDLE (integrated development and learning environment) — это интегрированная среда разработки и самообучения. Она представляет собой обычную интегрированную среду разработки (IDE) с добавлением буквы L, в результате давая аббревиатуру, которая таким образом косвенно ссылается на Эрика Айдла (Eric Idle) из знаменитого комик-шоу Монти Пайтон (Monty Python). Интерактивная оболочка, также именуемая интерпретатором, представляет собой окно, которое позволяет немедленно выполнять команды и тестировать программный код без необходимости создания файла.

Среда IDLE имеет ряд недостатков, таких как отсутствие столбца с номером строки в текстовом редакторе, но она является бесплатной и поставляется в комплекте с языком Python, поэтому каждый имеет к ней доступ. Вы можете использовать любую интегрированную среду разработки, которую вы захотите. В Интернете есть много вариантов, таких как Geany (произносится "джини"), PyCharm и PyScripter. Geany работает с широким спектром операционных систем, включая UNIX, macOS и Windows. PyCharm работает с Linux, Windows и macOS. PyScripter работает с Windows. Для получения подробного списка доступных инструментов разработки на Python и совместимых платформ посетите веб-страницу **https://wiki.python.org/moin/DevelopmentTools/**.

Код

В книге приведены все строки кода для каждого проекта, и я рекомендую вам набирать их вручную всякий раз, когда это возможно. Мне запомнилось выражение моего школьного учителя, который однажды сказал, что мы "учимся своими руками", и должен согласиться, что ввод кода заставляет нас уделять максимальное внимание тому, что происходит.

Но если вы хотите быстро завершить проект или же если вы случайно удалили всю свою работу, то вы можете скачать весь код, включая решения практических проектов, с веб-сайта книги по адресу **https://www.nostarch.com/impracticalpython/**.

Стиль написания программного кода

Эта книга посвящена решению занимательных задач и предназначена для новичков, поэтому исходный код может иногда отклоняться от передовых практических решений и максимальной эффективности. Иногда вы, возможно, будете использовать операцию включения в список или специальный оператор, но по большей части вы сосредоточитесь на простом, доступном для понимания коде, который легко усвоить.

Программирующим непрограммистам, читающим эту книгу, важно придерживаться простоты. Значительная часть их кода, возможно, будет "кодом Kleenex" — его используют один-два раза для определенной цели, а затем выбрасывают. Это тот тип кода, которым можно делиться с коллегами или который можно навязать им во время кадровых изменений, поэтому он должен быть легким в понимании и усвоении.

Весь главный код проекта аннотируется и объясняется автономно, и он обычно следует рекомендациям PEP 8 (Python Enhancement Proposal) по написанию программного кода на языке Python. Подробная информация о рекомендациях PEP 8 и программном обеспечении, которое поможет вам соблюдать эти рекомендации, приведена в *главе 1*.

Где получить помощь

Берясь за решение сложной программистской задачи, вы можете столкнуться с трудностями. Написание программного кода — это не всегда то, что можно понять интуитивно, даже с помощью такого дружественного языка, как Python. В последующих главах я буду приводить ссылки на полезные источники информации, но в случае проектов, которые вы формулируете самостоятельно, ничто не может превзойти поиск в Интернете.

Ключ к успешному поиску — знать, о чем спрашивать. Поначалу он может оказаться безуспешным, вызывая досаду, но подумайте о нем как об игре в двадцать вопросов. Продолжайте с каждым последующим поисковым запросом оттачивать свои ключевые слова до тех пор, пока не найдете ответ или не достигнете точки уменьшения отдачи.

Если книги и онлайновый поиск оказываются безуспешными, то следующий шаг — спросить у кого-то еще. Вы можете сделать это онлайн, за отдельную плату либо на бесплатных форумах, таких как Stack Overflow (**https://stackoverflow.com**), но имейте в виду: члены этих веб-сайтов не в восторге от человеческой глупости. Перед опубликованием обязательно прочитайте их страницы с рекомендациями о том, как задать хороший вопрос (которые так и называются "How do I ask a good question?"); например, подобную страницу можно найти на веб-сайте Stack Overflow по адресу **http://stackoverflow.com/help/how-to-ask/**.

Поехали!

Спасибо, что нашли время прочитать данное введение! Вы явно хотите получить от этой книги как можно больше, и вы на правильном пути. Когда вы достигнете последней страницы, вы будете более искусны в языке Python и лучше подготовлены к решению сложных реальных задач. Давайте же приступим к работе.

1

ГЕНЕРАТОР ДУРАЦКИХ ИМЕН

Американский кабельный телеканал USA Network однажды транслировал детективный сериал под названием "Ясновидец" (Psych), в котором гипернаблюдательный сыщик-любитель Шон Спенсер решал дела, притворяясь, что пользуется экстрасенсорными способностями. Фирменным знаком данного сериала было то, как Шон представлял своего напарника Гаса дурацкими именами, выдумываемыми им на лету, такими как Галилео Хампкинс, Лавандер Гумз и Плохие Новости Марвин Барнс. Это произвело на меня сильное впечатление, ибо много лет назад один мой приятель, который работал в Бюро переписи населения, дал мне список настоящих имен, таких же странных, как те, которые придумывал Шон.

Проект 1: генерирование псевдонимов

В этом разминочном проекте вы напишете простую программу на языке Python, которая генерирует чудаковатые имена, случайным образом комбинируя имена и фамилии. Если повезет, то вы создадите обилие псевдонимов, которые наполнят гордостью любого закадычного друга. Вы также познакомитесь с рекомендациями по написанию программного кода и примените внешние программы, которые помогут вам писать код, соответствующий этим рекомендациям.

Сериал "Ясновидец" не в вашем стиле? Замените имена в моем списке в коде собственными шутками или темой. Вы можете так же легко превратить этот проект в генератор имен "Игры престолов", или, возможно, вы захотите изобрести свое

имя типа Бенедикт Камбербэтч; мое любимое — Бендилик Крикетбэт (Bendylick Cricketbat, что на русском будет примерно звучать как Похотлиз Дубомоль).

Цель

Случайно сгенерировать смешные имена закадычных друзей, используя код на Python, который соответствует установленным стилевым рекомендациям.

Планирование и конструирование проекта

Время на планирование никогда не тратится впустую. Не имеет значения, программируете ли вы для получения удовольствия или заработка, в какой-то момент вам нужно будет оценить — довольно точно, — сколько времени займет проект, с какими препятствиями вы можете столкнуться и какие инструменты и ресурсы вам понадобятся для выполнения работы. И для этого вам в первую очередь нужно понять, что конкретно вы пытаетесь создать!

Помню слова одного успешного менеджера, который как-то сказал мне, что секрет его успеха заключается в том, чтобы просто задавать много вопросов. Что ты пытаешься сделать? Зачем ты это делаешь? Почему ты делаешь это таким образом? Сколько у тебя времени? Сколько денег? Ответы на эти вопросы чрезвычайно полезны для процесса проектирования и дают вам четкий ракурс.

В своей книге "Думай по-питоновски" (Downey A. Think Python. O'Reilly, 2015) Аллен Дауни описывает два типа планов разработки программного обеспечения: "прототип и заплатка" и "проектная разработка". В случае прототипа и заплатки вы начинаете с простой программы, а затем накладываете заплатки или отредактированный код для решения проблем, возникающих при тестировании. Этот подход может быть неплохим, когда вы работаете над сложной задачей, которую вы не очень хорошо понимаете. Но он также может создавать усложненный и ненадежный код. Если у вас есть четкое представление о задаче и о том, как вы хотите ее решить, то во избежание будущих проблем и их последующих исправлений вы должны применять план проектной разработки. Этот подход еще больше может упростить и поднять эффективность написания программного кода, и он обычно приводит к более прочному и надежному программному коду.

В этой книге вы будете начинать все проекты с четко определенной задачи или цели, которые лягут в основу ваших проектных решений. Затем мы будем обсуждать стратегию, которая поможет лучше понять возникающие вопросы и создавать план проектной разработки.

Стратегия

Здесь вы начнете с двух списков смешных имен и фамилий. Списки будут относительно короткими, поэтому они не будут потреблять много памяти, не будут нуждаться в динамическом обновлении и не должны представлять никаких сложностей во время выполнения. Поскольку единственное, что от вас потребуется, — это про-

читать имена из списка, вы задействуете кортеж, который будет использован в качестве контейнера.

С помощью двух кортежей имен вы создадите новые имена — попарно соединяя имена с фамилиями — одним нажатием кнопки. Благодаря этому пользователь сможет легко повторять этот процесс до тех пор, пока не появится достаточно смешное имя.

Вы также должны каким-то образом выделить имя в окне интерпретатора так, чтобы оно отличалось в командной строке. Интерактивная оболочка IDLE дает не слишком много вариантов шрифтов, но вы, вероятно, — как никто другой — знаете, что ошибки выводятся на экран красным цветом. По умолчанию функция `print()` выводит печать в стандартный канал вывода, но, загрузив модуль `sys`, можно перенаправлять вывод в канал ошибок с его фирменным красным цветом, используя параметр `file`:

```
print(печатаемое_сообщение, file=sys.stderr)
```

Наконец, вы выясните, какие, собственно, существуют стилевые рекомендации по написанию программного кода на языке Python. Эти рекомендации должны касаться не только кода, но и размещаемой в коде документации.

Псевдокод

"Можно всегда рассчитывать на то, что американцы сделают все правильно после того, как они испробовали все остальное". Эта цитата, приписываемая Уинстону Черчиллю, подводит итог тому, как многие люди подходят к написанию псевдокода.

Псевдокод — это высокоуровневый, неформальный способ описания компьютерных программ с использованием структурированного английского или любого другого человеческого языка. Он должен напоминать упрощенный язык программирования и содержать ключевые слова и соответствующие отступы. Разработчики используют его для того, чтобы игнорировать весь тайный синтаксис истинных языков программирования и сосредоточиться на базовой логике. Невзирая на свое широкое применение, псевдокод не имеет официальных стандартов — только руководящие принципы.

Если вы обнаружили, что широким шагом движетесь в сторону разочарования, то это может быть только потому, что вы не нашли времени написать псевдокод. Я искренне верю в псевдокод, т. к. он безошибочно приводил меня к решениям тогда, когда я оказывался блуждающим по лесу. Следовательно, в большинстве проектов этой книги вы будете использовать некоторую форму псевдокода. По крайней мере, надеюсь, что вы увидите его полезность, но я также надеюсь, что вы дисциплинируете себя и будете писать его в собственных проектах.

Очень высокоуровневый псевдокод нашего генератора забавных имен может выглядеть так:

```
Загрузить список имен
Загрузить список фамилий
Выбрать имя наугад
```

```
Назначить имя переменной
Выбрать фамилию наугад
Назначить имя переменной
Напечатать имена на экране по порядку и красным шрифтом
Запросить у пользователя, что делать дальше: выйти из программы или сыграть еще раз
Если пользователь решит сыграть еще раз:
  Повторить
Если пользователь решит выйти:
  Завершить и выйти из программы
```

Если только вы не пытаетесь воплотить здесь программный класс либо предоставить четкие инструкции другим, то сосредоточьтесь на *цели* псевдокода; не старайтесь рабски соблюдать (нестандартные) рекомендации по его написанию. И не зацикливайтесь только на программировании — процесс написания псевдокода можно применить к гораздо большему кругу задач. Как только вы с ним освоитесь, вы обнаружите, что он поможет вам и в других вещах, таких как уплата налогов, планирование инвестиций, строительство дома или подготовка к походу. Это отличный способ сосредоточиться и перенести успехи в программировании в реальную жизнь. О, если бы только его применяли в конгрессе!

Код

В листинге 1.1 приведен код для генератора смешных имен (файл pseudonyms.py), который составляет и печатает список псевдонимов из двух кортежей имен. Если вы не хотите набирать все имена на клавиатуре, то можете набрать их подмножество или же скачать код с веб-сайта книги по адресу **https://nostarch.com/impracticalpython/**.

Листинг 1.1. Генерирует глупые псевдонимы из кортежей имен. Файл pseudonyms.py

```
❶ import sys, random
❷ print("Добро пожаловать в 'Подбор имен для напарника.' как в сериале
'Ясновидец'\n")
  print("Имя, наподобие того, которое Шин подбирал для Гаса:\n\n")

  first = ('Baby Oil', 'Bad News', 'Big Burps', "Bill 'Beenie-Weenie'",
           "Bob 'Stinkbug'", 'Bowel Noises', 'Boxelder', "Bud 'Lite' ",
           'Butterbean', 'Buttermilk', 'Buttocks', 'Chad',
           'Chesterfield', 'Chewy', 'Chigger", "Cinnabuns', 'Cleet',
           'Cornbread', 'Crab Meat', 'Crapps', 'Dark Skies',
           'Dennis Clawhammer', 'Dicman', 'Elphonso', 'Fancypants',
           'Figgs', 'Foncy', 'Gootsy', 'Greasy Jim', 'Huckleberry',
           'Huggy', 'Ignatious', 'Jimbo', "Joe 'Pottin Soil'", 'Johnny',
           'Lemongrass', 'Lil Debil', 'Longbranch', '"Lunch Money"',
           'Mergatroid', '"Mr Peabody"', 'Oil-Can', 'Oinks',
           'Old Scratch', 'Ovaltine', 'Pennywhistle', 'Pitchfork Ben',
           'Potato Bug', 'Pushmeet','Rock Candy', 'Schlomo',
```

```
            'Scratchensniff', 'Scut', "Sid 'The Squirts'",
            'Skidmark', 'Slaps', 'Snakes', 'Snoobs', 'Snorki',
            'Soupcan Sam', 'Spitzitout', 'Squids', 'Stinky', 'Storyboard',
            'Sweet Tea', 'TeeTee', 'Wheezy Joe', "Winston 'Jazz Hands'",
            'Worms')

    last = ('Appleyard', 'Bigmeat', 'Bloominshine', 'Boogerbottom',
            'Breedslovetrout', 'Butterbaugh', 'Clovenhoof', 'Clutterbuck',
            'Cocktoasten', 'Endicott', 'Fewhairs', 'Gooberdapple',
            'Goodensmith', 'Goodpasture', 'Guster', 'Henderson',
            'Hooperbag', 'Hoosenater', 'Hootkins', 'Jefferson', 'Jenkins',
            'Jingley-Schmidt', 'Johnson', 'Kingfish', 'Listenbee',
            "M'Bembo", 'McFadden', 'Moonshine', 'Nettles', 'Noseworthy',
            'Olivetti', 'Outerbridge', 'Overpeck', 'Overturf', 'Oxhandler',
            'Pealike', 'Pennywhistle', 'Peterson', 'Pieplow', 'Pinkerton',
            'Porkins', 'Putney', 'Quakenbush', 'Rainwater', 'Rosenthal',
            'Rubbins', 'Sackrider', 'Snuggleshine', 'Splern', 'Stevens',
            'Stroganoff', 'Sugar-Gold', 'Swackhamer', 'Tippins',
            'Turnipseed', 'Vinaigrette', 'Walkingstick', 'Wallbanger',
            'Weewax', 'Weiners', 'Whipkey', 'Wigglesworth', 'Wimplesnatch',
            'Winterkorn', 'Woolysocks')

❸ while True:
    ❹ firstName = random.choice(first)

    ❺ lastName = random.choice(last)

      print("\n\n")
    ❻ print("{} {}".format(firstName, lastName), file=sys.stderr)
      print("\n\n")

    ❼ try_again = input("\n\nПопробуете еще? (Нажмите Enter либо n, чтобы
выйти)\n ")
      if try_again.lower() == "n":
          break

❽ input("\nНажмите Enter для завершения работы.")
```

Сначала импортируйте модули `sys` и `random` ❶. Модуль `sys` вы будете использовать для доступа к системным функциям сообщений об ошибках с целью окраски своего результата в привлекательный красный цвет в окне интерпретатора IDLE. А модуль `random` позволяет вам выбирать наугад элементы из списков имен.

Инструкции `print` в ❷ знакомят пользователя с программой. Команда новой строки `\n` заставляет появиться новую строку, а одинарные кавычки " позволяют использовать кавычки в распечатке без необходимости прибегать к управляющему символу обратной косой черты, который уменьшает читаемость кода.

Далее определите свои кортежи имен. Затем инициируйте цикл `while` ❸. Выражение `while = True`, в сущности, означает "продолжать выполнять до тех пор, пока я не скажу вам остановиться". Впоследствии для завершения цикла вы примените инструкцию `break`.

Цикл начинает со случайного выбора имени из первого кортежа, а затем назначает это имя переменной `firstName` ❹. Он использует метод `choice` модуля `random`, который возвращает случайный элемент из непустой последовательности — в данном случае кортежа имен.

Далее случайно выберите фамилию из кортежа `last` и назначьте ее переменной `lastName` ❺. Теперь, когда у вас есть оба имени, напечатайте их и хитростью заставьте интерактивную оболочку IDLE использовать красный шрифт "ошибки", указав в инструкции `print` необязательный аргумент `file=sys.stderr` ❻. Для преобразования переменных с именами в символьную цепочку[1] используйте более новый *метод* форматирования символьных цепочек, а не старый *оператор* форматирования (`%`). Дополнительные сведения о новом методе см. на веб-странице **https://docs.python.org/3.7/library/string.html**.

После того как имя будет выведено на экран, попросите пользователя выбрать дальнейшее действие: сыграть еще раз либо выйти из программы, применив для этого инструкцию `input` с описанием в кавычках. В этом случае задействуйте несколько пустых строк для того, чтобы сделать смешное имя в окне интерпретатора IDLE очевиднее. Если пользователь отвечает нажатием клавиши <Enter>, то в переменную `try_again` ❼ ничего не возвращается. Если ничего не возвращается, то условие в инструкции `if` не удовлетворяется, цикл `while` продолжается, и печатается новое имя. Если вместо этого пользователь нажимает клавишу <N>, то инструкция `if` приводит к команде `break`, и цикл завершается, потому что инструкция `while` больше не оценивается как `True`. Используйте метод `lower()` перевода символьной цепочки в нижний регистр для того, чтобы учесть ситуацию, когда у игрока задействована клавиша <Caps Lock>. Другими словами, не имеет значения, в нижнем или верхнем регистре пользователь вводит символ N, потому что программа всегда будет читать его в нижнем регистре.

Наконец, попросите пользователя выйти из программы, нажав клавишу <Enter> ❽. Нажатие клавиши <Enter> не задает переменной значение, возвращаемое из функции `input()`, программа завершается, и окно консоли закрывается. Нажатие клавиши <F5> в окне текстового редактора IDLE приводит к исполнению законченной программы.

Этот код работает, но только работы недостаточно — программы на языке Python должны работать в принятом стиле.

---

[1] В настоящем переводе термин string намеренно переводится как символьная цепочка, реже — как символьная последовательность, а не как "строка", что нередко можно встретить в подобной переводной литературе. Это сделано намеренно с целью избежать путаницы при переводе таких терминов, как line of code (строка кода) или row (строка таблицы), которые часто встречаются вместе в одном абзаце. — *Прим. перев.*

Использование руководства сообщества Python по стилю написания кода

Согласно Дзен языка Python (**https://www.python.org/dev/peps/pep-0020/**), "должен существовать один — и желательно только один — очевидный способ сделать это". В духе предоставления единственного очевидного "правильного способа" делать вещи и в целях достижения консенсуса вокруг этих практических приемов сообщество Python выпускает рекомендации по улучшению языка Python, которые представляют собой правила написания программного кода на Python, в состав которых входит стандартная библиотека главного дистрибутива Python. Наиболее важными из них являются рекомендации PEP 8, руководство по написанию программного кода на языке Python. Время идет, и PEP 8 регулярно эволюционирует, поскольку выявляются новые правила, а прошлые устаревают из-за изменений в языке.

Рекомендации PEP 8 (**https://www.python.org/dev/peps/pep-0008/**) устанавливают стандарты для правил именования, использования пустых строк, отступов и пробелов, максимальной длины строки, комментариев и т. д. Цель состоит в том, чтобы улучшить читаемость кода и сделать его единообразным между широким спектром программ на Python. Когда вы только начинаете программировать, то должны стремиться научиться и следовать принятым правилам до того, как укоренятся вредные привычки. Программный код в этой книге будет точно соответствовать рекомендациям PEP 8, но из уважения к издательской индустрии я переопределил некоторые правила (например, за счет меньшего объема комментированного кода, меньшего числа пустых строк и более коротких литералов документирования).

Стандартизованные имена и процедуры особенно важны, когда вы работаете в кросс-функциональных группах. При переводе с языка ученых на язык инженеров многое может потеряться, как в 1999 г., когда инженеры потеряли климатический орбитальный спутник Марса, потому что разные группы разработчиков использовали разные единицы измерения. В течение почти двух десятилетий я строил компьютерные модели Земли, которые трансформировались в инженерную функцию. Инженеры использовали мои скрипты для загрузки этих моделей в собственные программы. От проекта к проекту они делились этими скриптами между собой, тем самым повышая эффективность и помогая неопытным. Поскольку эти "командные файлы" были специально настроены под каждый проект, то по понятным причинам инженеры были не в восторге, когда во время обновлений моделей имена атрибутов менялись. По сути дела, одним из их внутренних принципов было "Упрашивай, подкупай или запугивай — лишь бы твой разработчик моделей применял единообразные имена свойств!".

Проверка кода с помощью программы Pylint

Непременно следует ознакомиться с правилами PEP 8, но вы все равно будете делать ошибки, а сверка своего кода с рекомендацией будет серьезным препятствием. К счастью, такие программы, как Pylint, pycodestyle и Flake8 способны помочь вам

легко следовать рекомендациям PEP 8. В этом проекте вы будете использовать программу Pylint.

Установка программы Pylint

Pylint — это средство проверки исходного кода, наличия ошибок и качества для языка программирования Python. Для скачивания его бесплатной копии перейдите по ссылке **https://www.pylint.org/#install** в разделе **Install** (Установить) найдите вашу платформу. Там будет отображена команда для установки программы Pylint. Так, в Windows перейдите в папку, содержащую вашу копию Python (например, C:\Python35), при нажатой клавише <Shift> щелкните правой кнопкой мыши для того, чтобы открыть контекстное меню, и в нем выберите команду **Открыть командное окно здесь** или **Открыть окно PowerShell здесь**, в зависимости от используемой версии Windows. Выполните `pip install pylint` (pip3, если установлены Python 2 и 3).

Выполнение программы Pylint

В Windows программа Pylint запускается из окна интерпретатора или же, в случае более новых систем, из интерактивной оболочки PowerShell (оба открываются при нажатой клавише <Shift> и щелчке правой кнопкой мыши в папке, содержащей модуль Python, который вы хотите проверить). Для запуска программы наберите `pylint` ***имя_файла*** (рис. 1.1). Расширение *py* является необязательным, и ваш путь к каталогу будет отличаться от показанного. В macOS или другой системе на базе UNIX используйте эмулятор терминала.

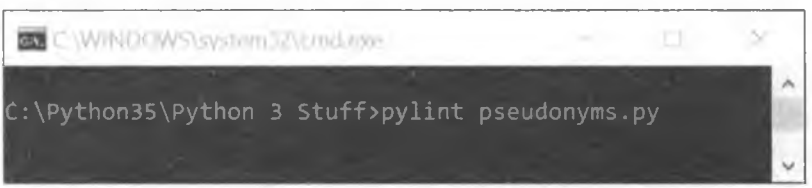


Рис. 1.1. Окно командной строки Windows с командой для запуска программы Pylint

В окне интерпретатора будут показаны результаты работы программы Pylint. Ниже приведен пример полезного результата:

```
C:\Python35\Python 3 Stuff\Psych>pylint pseudonyms.py
No config file found, using default configuration
************* Module pseudonyms
C: 45, 0: No space allowed around keyword argument assignment
    print(firstName, lastName, file = sys.stderr)
                                    ^ (bad-whitespace)
C:  1, 0: Missing module docstring (missing-docstring)
C:  2, 0: Multiple imports on one line (sys, random) (multiple-imports)
C:  7, 0: Invalid constant name "first" (invalid-name)
C: 23, 0: Invalid constant name "last" (invalid-name)
C: 40, 4: Invalid constant name "firstName" (invalid-name)
```

```
C: 42, 4: Invalid constant name "lastName" (invalid-name)
C: 48, 4: Invalid constant name "try_again" (invalid-name)
```

Заглавная буква в начале каждой строки — это код сообщения. Например, `C: 15, 0` относится к нарушению стандарта написания программного кода в строке 15 и столбце 0. В случае разнообразных кодов сообщений программы Pylint можно ссылаться на следующие ключи:

- `R` — реорганизация программного кода (рефакторинг); соответствует нарушению показателя "общепринятого на практике приема";
- `C` — правило (конвенция); соответствует нарушению стандарта написания программного кода;
- `W` — предупреждение; соответствует стилистическим проблемам или незначительным проблемам в написании программного кода;
- `E` — ошибка; соответствует важным проблемам в написании программного кода (т. е., скорее всего, дефектам);
- `F` — фатальная ошибка; соответствует ошибкам, которые препятствуют дальнейшей обработке.

Pylint завершит свой отчет, оценив соответствие вашей программы рекомендациям PEP 8. В данном конкретном случае ваш код получил отметку 4 из 10 возможных:

```
Global evaluation
-----------------
Your code has been rated at 4.00/10 (previous run: 4.00/10, +0.00)
```

Обработка ложных ошибок константных имен

Возможно, вы заметили, что программа Pylint неправильно предполагает, что все имена переменных в глобальном пространстве относятся к константам и поэтому должны начинаться с заглавных букв. Этот недостаток можно обойти несколькими способами. Первый из них — разместить свой программный код в функции `main()` (листинг 1.2); благодаря этому он будет находиться вне глобального пространства.

Листинг 1.2. Определяет и вызывает функцию `main()`

```
    def main():
        некий вмененный код
        некий вмененный код
        некий вмененный код
❶ if __name__ == "__main__":
    ❷ main()
```

Имя `__name__` является специальной встроенной переменной, которую можно использовать для оценивания того, как выполняется программа: в автономном режиме либо как импортированный модуль; помните, что модуль — это просто программа на Python, используемая внутри другой программы на Python. Если вы

выполняете программу напрямую, то переменная `__name__` имеет значение `__main__`. В листинге 1.2 переменная `__name__` используется для обеспечения того, чтобы при импортировании программы функция `main()` не выполнялась до тех пор, пока вы не вызовете ее намеренно, однако, когда вы выполняете программу напрямую, условие в инструкции `if` удовлетворено не будет ❶ и функция `main()` будет вызвана автоматически ❷. Это правило не всегда вам потребуется. Например, если ваш программный код всего лишь определяет функцию, то вы можете загрузить его как модуль и вызвать ее без необходимости в переменной `__name__`.

Давайте встроим все это в программу pseudonyms.py под функцией `main()`, за исключением инструкции `import`, а затем вставим вызов функции `main()` под инструкцией `if`, как в листинге 1.2. Соответствующие изменения можно внести самостоятельно либо скачать программу pseudonyms_main.py с веб-сайта. Повторно выполните программу Pylint. В окне интерпретатора вы должны получить следующие ниже результаты.

```
C:\Python35\Python 3 Stuff\Psych>pylint pseudonyms_main
No config file found, using default configuration
************* Module pseudonyms_main
C: 47, 0: No space allowed around keyword argument assignment
        print(firstName, lastName, file = sys.stderr)
                                         ^ (bad-whitespace)
C:  1, 0: Missing module docstring (missing-docstring)
C:  2, 0: Multiple imports on one line (sys, random) (multiple-imports)
C:  4, 0: Missing function docstring (missing-docstring)
C: 42, 8: Invalid variable name "firstName" (invalid-name)
C: 44, 8: Invalid variable name "lastName" (invalid-name)
```

Теперь раздражающие комментарии о недопустимых именах констант исчезли, но вы еще не вышли из леса. Как бы мне это ни нравилось, но правила языка Python не допускают "горбатых" имен, таких как `firstName`.

Конфигурирование программы Pylint

При вычислении небольших скриптов я предпочитаю использовать настройки по умолчанию программы Pylint и игнорировать ложные ошибки "имен констант". Я также люблю запускать опцию `-rn` (аббревиатура для `-reports=n`) для подавления большого объема посторонней статистики, которую программа Pylint возвращает:

```
C:\Python35\Python 3 Stuff\Psych>pylint -rn pseudonyms_main.py
```

Обратите внимание, что использование `-rn` отключит параметр оценивания программного кода.

Еще одна проблема с программой Pylint заключается в том, что ее максимальная длина строки по умолчанию составляет 100 символов, но PEP 8 рекомендует 79 символов. В целях соблюдения рекомендаций PEP 8 программу Pylint можно выполнить со следующей настройкой:

```
C:\Python35\Python 3 Stuff\Psych>pylint --max-line-length=79 pseudonyms_main
```

Теперь вы увидите, что выделение имен функции `main()` отступами привело к тому, что некоторые строки превысили пределы, установленные рекомендациями:

```
C: 12, 0: Line too long (80/79) (line-too-long)
C: 14, 0: Line too long (83/79) (line-too-long)
--обрезано--
```

Наверняка вы не захотите конфигурировать программу Pylint всякий раз при ее выполнении, и, к счастью, вам это не нужно делать. Вместо этого с помощью команды `-generate-rcfile` можно создать собственный конфигурационный файл. Например, для того чтобы подавить создание отчетов и установить максимальную длину строки, равной 79, в командной строке введите следующее:

```
имя вашего пути>pylint -rn --max-line-length=79 --generate-rcfile >
                                                        name.pylintrc
```

Поместите необходимые изменения перед инструкцией `--generate-rcfile > name.pylintrc` и укажите свое имя перед расширением .pylintrc. Конфигурационный файл можно создать автономно, как только что было показано, либо одновременно с оцениванием программы Python. Файл .pylintrc автоматически сохраняется в текущем рабочем каталоге, хотя есть возможность добавить каталожный путь (более подробную информацию см. на веб-страницах **https://pylint.org** и **https://pylint.readthedocs.io/en/latest/user_guide/run.html**).

Индивидуально настроенный конфигурационный файл можно применить повторно с помощью параметра `--rcfile`, за которым должны находиться имя вашего личного файла конфигурации и имя оцениваемой программы. Например, для того чтобы выполнить конфигурацию myconfig.pylintrc на программе pseudonyms_main.py, введите следующее:

```
C:\Python35\Python 3 Stuff\Psych>pylint --rcfile myconfig.pylintrc
                                                    pseudonyms_main
```

Описание своего кода с помощью литералов документирования

Программа Pylint обнаруживает, что в программе pseudonyms_main.py отсутствует литерал документирования. Согласно рекомендациям по написанию программного кода на языке Python PEP 257 (**https://www.python.org/dev/peps/pep-0257/**), литерал документирования docstring — это символьный литерал, который появляется в качестве первой инструкции в определении модуля, функции, класса или метода. Литерал документирования представляет собой в основном краткое описание того, что делает ваш код, и может включать в себя конкретные аспекты программного кода, такие как необходимые входные данные. Здесь в тройных кавычках приведен пример однострочного литерала документирования для функции:

```
def circ(r):
    """Вернуть длину окружности радиусом r."""
    c = 2 * r * math.pi
    return c
```

Приведенный выше литерал документирования просто описывает то, что конкретно функция делает, но литералы документирования могут быть длиннее и содержать намного больше информации. Например, ниже приведен многострочный литерал документирования для той же функции, который показывает информацию о входе и выходе функции:

```
def circ(r):
    """Вернуть длину окружности радиусом r.

    Аргументы:
    r - радиус круга

    Возвращает:
        float: окружность круга
    """
    c = 2 * r * math.pi
    return c
```

К сожалению, литералы документирования специфичны тем, что прочно привязаны к человеку, проекту и компании, и можно найти целый ряд конфликтующих рекомендаций. Компания Google имеет свой формат и превосходное стилевое руководство. Некоторые члены научного сообщества используют стандарт литералов документирования библиотеки `NumPy`. А текстовый формат облегченной разметки RST (reStructured Text) используется в основном в сочетании с инструментом Sphinx, который применяет литералы документирования с целью создания документации для проектов на Python в файлах таких форматов, как HTML и PDF. Если вы когда-нибудь читали документацию (**https://readthedocs.org/**) по какому-нибудь модулю Python, то видели Sphinx в действии. Ссылки на руководства для некоторых из этих стилей можно найти в *разд. "Дальнейшее чтение" далее в этой главе.*

С помощью бесплатного инструмента pydocstyle можно проверить, насколько хорошо ваша документация соответствует рекомендациям PEP 257. Для того чтобы установить его в Windows или любой другой операционной системе, откройте окно интерпретатора и выполните команду **`pip install pydocstyle`** (используйте pip3, если установлены Python 2 и 3).

Для выполнения pydocstyle откройте окно интерпретатора в папке, содержащей код, который вы хотите проверить. Если не указать имя файла, то pydocstyle будет работать на всех находящихся в папке программах Python и предоставит отклик:

```
C:\Python35\Python 3 Stuff\Psych>pydocstyle
.\OLD_pseudonyms_main.py:1 at module level:
     D100: Missing docstring in public module
.\OLD_pseudonyms_main.py:4 in public function `main`:
     D103: Missing docstring in public function
.\ pseudonyms.py:1 at module level:
     D100: Missing docstring in public module
.\ pseudonyms_main_broken.py:1 at module level:
     D200: One-line docstring should fit on one line with quotes (found 2)
```

```
.\ pseudonyms_main_broken.py:6 in public function `main`:
     D205: 1 blank line required between summary line and description
(found 0)
```

Если же указать файл, в котором нет проблем с литералом документирования, то pydocstyle ничего не вернет:

```
C:\Python35\Python 3 Stuff\Psych>pydocstyle pseudonyms_main_fixed.py

C:\Python35\Python 3 Stuff\Psych>
```

Во всех проектах этой книги я буду использовать довольно простые литералы документирования, с тем чтобы уменьшить визуальный шум в аннотированном программном коде. Если вы захотите попрактиковаться, можете свободно их расширить. Свои результаты всегда можно проверить с помощью инструмента pydocstyle.

Проверка стиля программного кода

Когда я был еще подростком, мой дядя ездил в большой город из нашего сельского городка делать себе "стильную" стрижку. Я никогда не понимал, чем она отличается от обычной стрижки, но зато я знаю, как "стилизовать" наш программный код генератора забавных имен так, чтобы он соответствовал рекомендациям PEP 8 и PEP 257.

Сделайте копию программы pseudonyms_main.py под названием pseudonyms_main_fixed.py и сразу же оцените ее с помощью программы Pylint, используя вот эту команду:

```
ваш_путь>pylint --max-line-length=79 pseudonyms_main_fixed
```

Не подавляйте отчет с помощью опции `-rn`. В нижней части окна интерпретатора вы должны увидеть вот этот результат:

```
Global evaluation
-----------------
Your code has been rated at 3.33/10
```

Теперь исправьте программный код на основе результата работы программы Pylint. В следующем ниже примере исправления выделены **жирным шрифтом**. Я внес изменения в кортежи с именами, исправив проблемы с длиной строк. Исправленный программный код pseudonyms_main_fixed.py можно также скачать с веб-страницы книги по адресу **https://www.nostarch.com/impracticalpython/**.

Файл pseudonyms_main_fixed.py

```
"""Сгенерировать забавные имена, случайно комбинируя имена
   из 2 отдельных списков."""
import sys
import random
```

```
def main():
    """Случайно выбрать имена из 2 кортежей имен и напечатать на экране."""
    print("Добро пожаловать в 'Подбор имен для напарника.' как в сериале
                                                        'Ясновидец'\n")
    print("Имя, наподобие того, которое Шин подбирал для Гаса:\n\n")

    first = ('Baby Oil', 'Bad News', 'Big Burps', "Bill 'Beenie-Weenie'",
             "Bob 'Stinkbug'", 'Bowel Noises', 'Boxelder', "Bud 'Lite'",
             'Butterbean', 'Buttermilk', 'Buttocks', 'Chad',
             'Chesterfield', 'Chewy', 'Chigger', 'Cinnabuns', 'Cleet',
             'Cornbread', 'Crab Meat', 'Crapps', 'Dark Skies',
             'Dennis Clawhammer', 'Dicman', 'Elphonso', 'Fancypants',
             'Figgs', 'Foncy', 'Gootsy', 'Greasy Jim', 'Huckleberry',
             'Huggy', 'Ignatious', 'Jimbo', "Joe 'Pottin Soil'", 'Johnny',
             'Lemongrass', 'Lil Debil', 'Longbranch', '"Lunch Money"',
             'Mergatroid', '"Mr Peabody"', 'Oil-Can', 'Oinks',
             'Old Scratch', 'Ovaltine', 'Pennywhistle', 'Pitchfork Ben',
             'Potato Bug', 'Pushmeet', 'Rock Candy', 'Schlomo',
             'Scratchensniff', 'Scut', "Sid 'The Squirts'", 'Skidmark',
             'Slaps', 'Snakes', 'Snoobs', 'Snorki', 'Soupcan Sam',
             'Spitzitout', 'Squids', 'Stinky', 'Storyboard', 'Sweet Tea',
             'TeeTee', 'Wheezy Joe', "Winston 'Jazz Hands'", 'Worms')

    last = ('Appleyard', 'Bigmeat', 'Bloominshine', 'Boogerbottom',
            'Breedslovetrout', 'Butterbaugh', 'Clovenhoof', 'Clutterbuck',
            'Cocktoasten', 'Endicott', 'Fewhairs', 'Gooberdapple',
            'Goodensmith', 'Goodpasture', 'Guster', 'Henderson',
            'Hooperbag', 'Hoosenater', 'Hootkins', 'Jefferson', 'Jenkins',
            'Jingley-Schmidt', 'Johnson', 'Kingfish', 'Listenbee',
            "M'Bembo", 'McFadden', 'Moonshine', 'Nettles', 'Noseworthy',
            'Olivetti', 'Outerbridge', 'Overpeck', 'Overturf',
            'Oxhandler', 'Pealike', 'Pennywhistle', 'Peterson', 'Pieplow',
            'Pinkerton', 'Porkins', 'Putney', 'Quakenbush', 'Rainwater',
            'Rosenthal', 'Rubbins', 'Sackrider', 'Snuggleshine', 'Splern',
            'Stevens', 'Stroganoff', 'Sugar-Gold', 'Swackhamer',
            'Tippins', 'Turnipseed', 'Vinaigrette', 'Walkingstick',
            'Wallbanger', 'Weewax', 'Weiners', 'Whipkey', 'Wigglesworth',
            'Wimplesnatch', 'Winterkorn', 'Woolysocks')

    while True:
        first_name = random.choice(first)
        last_name = random.choice(last)

        print("\n\n")
        # хитростью заставить IDLE использовать конфигурацию
        # для "фатальной ошибки" с целью распечатки имени в красном цвете.
        print("{} {}".format(first_name, last_name), file=sys.stderr)
        print("\n\n")
```

```
        try_again = input("\n\nПопробуете еще? (Нажмите Enter либо n,
                                                    чтобы выйти)\n ")

        if try_again.lower() == "n":
            break

    input("\nНажмите Enter для завершения работы.")

if __name__ == "__main__":
    main()
```

Пересмотренный программный код получит от программы Pylint оценку 10 из 10:

```
Global evaluation
-----------------
Your code has been rated at 10.00/10 (previous run: 3.33/10, +6.67)
```

Как вы видели в предыдущем разделе, выполнение инструмента pydocstyle на программе pseudonyms_main_fixed.py не дает никаких ошибок, но не обманывайте себя: это не означает, что программа является хорошей или даже адекватной. Например, вот этот литерал документирования также проходит:

```
"""ksjkdls lskjds kjs jdi wllk sijkljs dsdw noiu sss."""
```

Писать разрозненные, краткие и действительно полезные литералы документирования и комментарии довольно трудно. Рекомендации PEP 257 помогут с литералами документирования, но комментарии имеют более свободный стиль и "открытый диапазон". Слишком много комментариев создают визуальный шум, они могут отталкивать пользователя своим видом, в них нет необходимости, и так же, как хорошо написанный программный код, в значительной степени являются самодокументирующимися. Вескими причинами для добавления комментариев может быть уточнение намерений и устранение потенциальных ошибок пользователей, например когда требуются конкретные единицы измерения или форматы входных данных. При поиске правильного баланса в комментариях обратите внимание на хорошие примеры, когда вы с ними сталкиваетесь. Кроме того, подумайте о том, что конкретно вы хотели бы увидеть, доведись вам открыть собственный код после пятилетнего перерыва!

Инструменты Pylint и pydocstyle просты в установке, легки в запуске и помогут вам усвоить и соблюдать стандарты написания программного кода, принятые в сообществе Python. Просеивание программного кода через программу Pylint до его публикации на веб-форумах также является хорошим практическим приемом, в особенности когда вы ищете помощь и ожидаете получить "более добрые и мягкие" ответы!

Резюме

Теперь вы должны знать, как писать код и документировать его, чтобы его комментарии соответствовали ожиданиям сообщества языка Python. А еще (и это важнее) вы создали несколько по-серьезному смешных имен для напарника, гангстера, информатора и кого угодно.

Вот несколько моих любимых:

```
Pitchfork Ben Pennywhistle          'Bad News' Bloominshine
Chewy Stroganoff                    'Sweet Tea' Tippins
Spitzitout Winterkorn               Wheezy Joe Jenkins
'Big Burps' Rosenthal               Soupcan Sam Putney
Bill 'Beenie-Weenie' Clutterbuck    Greasy Jim Wigglesworth
Dark Skies Jingley-Schmidt          Chesterfield Walkingstick
Potato Bug Quakenbush               Jimbo Woolysocks
Worms Endicott                      Fancypants Pinkerton
Cleet Weiners                       Dicman Overpeck
Ignatious Outerbridge               Buttocks Rubbins
```

Дальнейшее чтение

Для получения интерактивной версии этих ресурсов посетите веб-сайт книги по адресу **https://www.nostarch.com/impracticalpython/**.

Псевдокод

Описания некоторых довольно формальных стандартов псевдокода можно найти по адресам **http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html** и **http://www.slideshare.net/sabiksabz/pseudo-code-basics/**.

Стилевые руководства

Вот список стилевых руководств, к которым вы можете обращаться при создании программ на языке Python.

- Рекомендации PEP 8 по написанию программного кода на языке Python можно найти по адресу **https://www.python.org/dev/peps/pep-0008/**.
- Рекомендации PEP 257 по написанию программного кода на языке Python можно найти по адресу **https://www.python.org/dev/peps/pep-0257/**.
- Google имеет собственное руководство по форматированию и стилю, которое размещено по адресу **https://google.github.io/styleguide/pyguide.html**.
- Примеры стиля Google можно найти по адресу **https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html**.
- Стандарты библиотеки `NumPy` по написанию литералов документирования находятся по адресу **https://numpydoc.readthedocs.io/en/latest/**.
- Примеры литералов документирования библиотеки `NumPy` можно найти по адресу **https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html**.
- О формате RST (reStructured Text) можно узнать по адресам **https://docs.python.org/devguide/documenting.html**, **https://docs.python.org/3.1/documenting/rest.html** и **https://wiki.python.org/moin/reStructuredText/**.

- Руководство автостопщика по Python (The Hitchhiker's Guide to Python. **http://docs.python-guide.org/en/latest/writing/style/**) содержит раздел о стилях программного кода и инструменте autopep8, который автоматически переформатирует код согласно рекомендаций PEP 8 (в какой-то степени).
- Книга Бретта Слаткина "Эффективный Python" (Slatkin B. Effective Python. Addison-Wesley, 2015) содержит полезный раздел по документированию программ.

Сторонние модули

Ниже приведено несколько ресурсов для использования сторонних модулей.

- Подробная информация о программе Pylint находится по адресу **https://docs.pylint.org/en/1.8/tutorial.html**.
- Подробную информацию об инструменте pydocstyle можно найти по адресу **http://www.pydocstyle.org/en/latest/**.

Практические проекты

Попробуйте приведенные ниже проекты для работы с символьными цепочками. Мои собственные решения доступны в приложении к книге.

"Поросячья" латынь

Для того чтобы сформировать "поросячью" латынь, берут английское слово, которое начинается с согласной, эту согласную перемещают в конец, а затем в конец слова добавляют "ay". Если слово начинается с гласной, то в конец слова добавляют "way". Одной из самых известных фраз на "поросячьей" латыни всех времен является "ixnay on the ottenray"[2], произнесенная Марти Фельдманом (Marty Feldman) в комедийном шедевре Мела Брукса (Mel Brooks) "Молодой Франкенштейн" (Young Frankenstein).

Напишите программу, которая на входе принимает слово и использует индексацию и операцию среза для возврата своего эквивалента "поросячьей" латыни. Примените инструменты Pylint и pydocstyle к своему коду и исправьте все стилевые ошибки. Решение задачи можно найти в приложении к книге либо скачать программу pig_latin_practice.py с веб-сайта **https://www.nostarch.com/impracticalpython/**.

Столбчатый график бедняка

Шесть наиболее часто используемых в английском языке букв можно запомнить с помощью мнемоники "etaoin" (произносится eh-tay-oh-in). Напишите скрипт на Python, который на входе принимает предложение (символьную цепочку) и воз-

[2] Соответствует словосочетанию nix on the rotten. — *Прим. перев.*

вращает результат в виде простого столбчатого графика, как показано на рис. 1.2. Подсказка: я использовал структуру данных "словарь" и два модуля, которые я еще не рассматривал: `pprint` и `collections/defaultdict`.

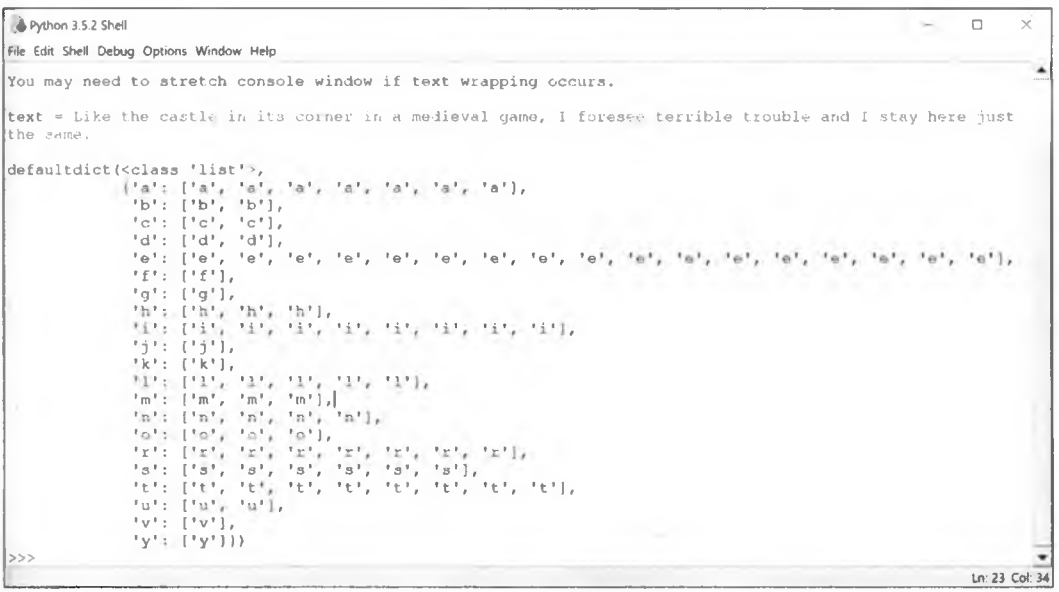


Рис. 1.2. Результат программы ETAOIN_practice.py из приложения к книге в виде столбчатого графика

Сложные проекты

Для сложных проектов не предусмотрено никаких решений. Ожидается, что вы справитесь с ними самостоятельно!

Столбчатый график бедного чужака

Примените онлайновый переводчик для того, чтобы изменить свой текст на другую систему латинского письма (например, испанскую или французскую), повторно выполните свой код из проекта о столбчатом графике бедняка и сравните результаты. Например, испанская версия текста из рис. 1.2 дает результаты на рис. 1.3.

В испанском предложении в два раза больше букв L и в три раза больше букв U. Для того чтобы сделать столбчатые графики прямо сопоставимыми для различных входных данных, измените код так, чтобы каждая буква алфавита имела ключ и выводилась на экран, даже если нет значений.

Среднее имя

Перепишите код генератора забавных имен так, чтобы включить средние имена (отчества). Сначала создайте новый кортеж `middle_name`, затем разделите сущест-

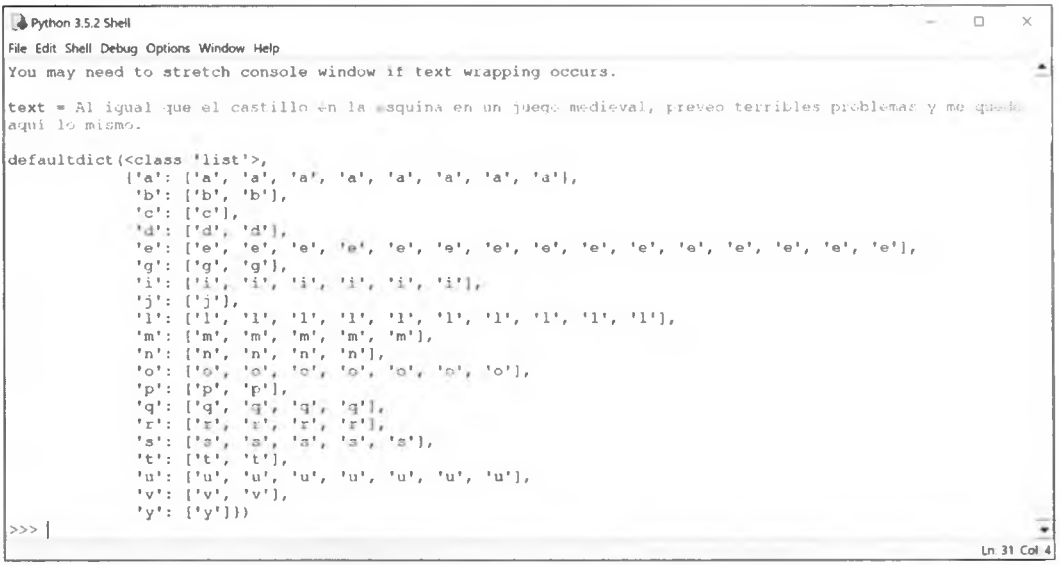


Рис. 1.3. Результаты выполнения программы EATOIN_challenge.py перевода текста из рис. 1.2 на испанский язык

вующие пары "имя — отчество" (например, " Joe 'Pottin Soil '" или " Sid 'The Squirts'") и добавьте их в кортеж. Вы также должны переместить в кортеж `middle_name` некоторые очевидные прозвища (например, "Oil Can"). Наконец, добавьте несколько новых средних имен (таких, как "The Big News", или "Grunts", или "Tinkie Winkie"). Примените модуль `random` языка Python для того, чтобы среднее имя выбиралось только в половине или трети случаев.

Что-то совершенно другое

Создайте собственный список смешных имен и добавьте его в генератор смешных имен. Подсказка: титры к кинофильмам являются богатым полем для поиска!

2

ПОИСК ПАЛИНГРАММНЫХ ЗАКЛИНАНИЙ

Радар. Заказ. Ротатор. Комок. Kayak. Sexes. Что общего у всех этих слов? Это *палиндромы* (перевертыши) — слова, которые читаются одинаково слева направо и справа налево. *Палинграммы* еще лучше[1]. Это целые фразы, которые ведут себя таким же образом. Наполеон является автором самой известной палинграммы. Когда он впервые увидел Эльбу, остров своего изгнания, он изрек: "Able was I ere I saw Elba", что примерно можно перевести на русский, как "Я был в состоянии, прежде чем увидел Эльбу".

В 2011 г. издательство DC Comics опубликовало интересную историю, которая подняла использование палинграмм на новый интеллектуальный уровень. Супергероиня, волшебница Затанна, была проклята и в результате могла произносить заклинания, только говоря палиндромами. Ей удалось придумать ровно столько словосочетаний из двух слов, таких как "nurses run, stack cats и puff up" (медсестры бегают, складывают кошек и раздуваются), что этого хватило для того, чтобы победить своего меченосного противника. Это заставило меня задаться вопросом: а сколько всего существует "боевых" палинграмм? И есть ли среди них для Затанны получше?

В этой главе вы скачаете файлы словарей из Интернета и примените Python для обнаружения в них сначала палиндромов, а затем более сложных палинграмм. Затем

---

[1] *Палинграмма* (palingram), или *сложный палиндром*, — это предложение, в котором буквы, слоги или слова читаются одинакового как слева направо, так и справа налево. Например, "a dog, a plan, a canal: pagoda". См. **https://www.sandipn.com.np/2017/05/collection-of-palingram-sentence.html**. — *Прим. перев.*

вы примените инструмент под названием `cProfile` для анализа программного кода генерирования палинграмм с целью сделать его производительнее. Наконец, вы просеете палинграммы через решето для того, чтобы посмотреть, сколько из них имеют "агрессивную" природу.

Поиски и открытие словаря

Все проекты этой главы требуют списков слов в формате текстового файла, обычно именуемого файлом словаря, поэтому давайте начнем с того, что узнаем, как загрузить один из них.

Несмотря на свое название, файлы словарей содержат только слова — без произношения, числа слогов, без определений и т. д. Эта новость радует, т. к. на нашем пути возникали бы всяческие сложности. Но что еще лучше, файлы словарей доступны онлайн бесплатно.

Подходящие файлы словарей можно найти в местах, перечисленных в табл. 2.1. Скачайте один из файлов или, если он открывается напрямую, скопируйте и вставьте содержимое в текстовый редактор, например Notepad или WordPad (TextEdit в macOS), и запишите его в файл с расширением .txt. Сохраните словарь в той же папке, что и код Python. Для подготовки этого проекта я использовал файл 2of4brif.txt. Его можно найти в скачиваемом архиве 12dicts-6.0.2.zip на веб-сайте, приведенном первым в табл. 2.1.

***Таблица 2.1.** Скачиваемые файлы словарей*

Файл	Число слов
http://wordlist.aspell.net/12dicts/	60 388
https://inventwithpython.com/dictionary.txt	45 000
http://www-personal.umich.edu/~jlawler/wordlist.html	69 903
http://greenteapress.com/thinkpython2/code/words.txt	113 809

В дополнение к файлам в табл. 2.1 операционные системы UNIX и UNIX-подобные системы поставляются в комплекте с большим файлом слов с разделением строк символом новой строки. В нем более чем 200 тыс. слов. Он обычно хранится в папке /usr/share/dict/words или /usr/dict/words. В операционной системе Debian GNU/Linux списки слов находятся в папке /usr/share/opendict/dictionaries.

Словари macOS обычно находятся в папке /Library/Dictionaries, и туда включены неанглийские словари. В случае, если вы захотите использовать один из этих файлов, то вам, возможно, придется выполнить поиск в Интернете с запросом по вашей операционной системе и соответствующей версии для того, чтобы найти точный путь к каталогу.

Некоторые файлы словарей исключают буквы *a* и *I* как слова. Другие могут включать каждую букву в словаре как однословный "заголовок" (например, d в начале

слов, начинающихся с d). В следующих ниже проектах однобуквенные палиндромы будут проигнорированы, поэтому подобного рода проблемы не должны вызывать затруднений.

Обработка исключений при открытии файлов

Всякий раз, когда вы загружаете внешний файл, ваша программа должна автоматически выполнять проверку на наличие проблем ввода-вывода, таких как отсутствующие файлы или неправильные имена файлов, и сообщить вам об этих проблемах.

Используйте следующие ниже инструкции `try` и `except` для перехвата и обработки исключений, т. е. ошибок, обнаруживаемых во время исполнения:

```
❶ try:
    ❷ with open(file) as in_file:
            сделать что-то
  except IOError❸ as e:
    ❹ print("{}\nОшибка при открытии {}. Завершение программы.".
            format(e, file), file=sys.stderr)
    ❺ sys.exit(1)
```

Сначала исполняется раздел `try` ❶. Инструкция `with` автоматически закроет файл после вложенного блока кода, независимо от того, как блок завершает работу ❷. Закрытие файлов перед завершением процесса является хорошим практическим решением. Если вы не закроете эти файлы, то можете исчерпать файловые дескрипторы (в основном эта проблема характерна для крупных скриптов, которые работают в течение длительного времени), заблокировать файл от дальнейшего доступа в Windows, повредить файлы либо потерять данные, если вы пишете в файл.

Если что-то идет не так и тип ошибки соответствует имени исключения после ключевого слова `except` ❸, то остальная часть раздела `try` пропускается, и исполняется раздел `except` ❹. Если все проходит нормально, то исполняется раздел `try`, а предложение `except` пропускается. Инструкция `print` в разделе `except` позволяет узнать о наличии проблемы, а аргумент `file=sys.stderr` окрашивает инструкцию `error` в красный цвет в окне интерпретатора IDLE.

Инструкция `sys.exit(1)` ❺ используется для завершения работы программы. Единица в `sys.exit(1)` указывает на то, что программа испытала ошибку и не закрылась успешно.

Если возникает исключение, которое *не соответствует* именованному исключению в разделе `except`, то оно передается наверх любым внешним разделам `try` или исполнению главной программы. Если обработчик не найден, то *необработанное исключение* вызывает остановку программы со стандартным сообщением об ошибке в отчете об обратной трассировке ("traceback").

Загрузка файла словаря

Листинг 2.1 загружает файл словаря в виде списка. Введите этот скрипт вручную либо скачайте его в форме файла load_dictionary.py с веб-сайта **https://nostarch.com/impracticalpython/**.

Этот файл можно импортировать в другие программы в виде модуля и запустить его с помощью однострочной инструкции. Помните, что модуль — это просто программа Python, которая может быть использована в другой программе Python. Как вы, вероятно, знаете, модули представляют собой форму *абстракции*. Абстракция означает, что вам не нужно беспокоиться обо всех деталях написания программного кода. Принцип абстракции состоит в *инкапсуляции*, акте сокрытия деталей. Мы инкапсулируем программный код загрузки файлов в модуль, поэтому в другой программе вам не нужно видеть или беспокоиться о деталях его программного кода.

Листинг 2.1. Модуль загрузки файла словаря в виде списка. Файл load_dictionary.py

```
"""Загрузить текстовый файл как список.

   Аргументы:
   - имя текстового файла (и каталожный путь, если необходимо)
   Исключения:
   - IOError, если имя файла не найдено.
   Возвращает:
   - список всех слов в текстовом файле в нижнем регистре.
   Требует - import sys
"""
❶ import sys

❷ def load(file):
      """Открыть текстовый файл и вернуть список цепочек символов
      в нижнем регистре."""
      try:
          with open(file) as in_file:
          ❸ loaded_txt = in_file.read().strip().split('\n')
          ❹ loaded_txt = [x.lower() for x in loaded_txt]
            return loaded_txt
      except IOError as e:
          ❺ print("{}\nОшибка при открытии {}. Завершение программы.".
                format(e, file), file=sys.stderr)
            sys.exit(1)
```

После литерала документирования мы импортируем системные функции с помощью модуля `sys` для того, чтобы заработал ваш программный код обработки ошибок ❶. Следующий блок кода определяет функцию, опираясь на предыдущее обсуждение темы открытия файлов ❷. Указанная функция в качестве аргумента принимает имя файла.

Если исключения не возникают, то в текстовом файле удаляются пробелы, а его элементы разбиваются на отдельные строки и добавляются в список ❺. Вы хотите, чтобы каждое слово стало отдельным элементом в списке перед тем, как этот список будет возвращен. И поскольку для языка Python регистр букв имеет значение, слова в списке конвертируются в нижний регистр посредством операции включения в список ❹. Включение в список — это сокращенный способ преобразования списка или другой итерируемой последовательности в другой список. В этом случае указанная операция заменяет цикл `for`.

При обнаружении ошибки ввода-вывода программа выводит на экран стандартное сообщение об ошибке, обозначаемое спецификатором `e`, а также сообщение, описывающее событие и информирующее пользователя об окончании работы ❺. Команда `sys.exit(1)` завершает работу программы.

Данный пример кода предназначен для иллюстративных целей, с тем что показать, как все эти шаги работают вместе. Как правило, инструкция `sys.exit()` не будет вызываться из модуля, т. к. вы можете захотеть, чтобы ваша программа сделала что-то еще до завершения, например написала в журнальный файл. В последующих главах для ясности и контроля мы переместим оба раздела `try-except` и команду `sys.exit()` в функцию `main()`.

Проект 2: поиск палиндромов

Вы начнете с отыскания однословных палиндромов в словаре, а затем перейдете к более сложным палиндромным словосочетаниям.

Цель

Применить язык Python для отыскания палиндромов в файле словаря английского языка.

Стратегия и псевдокод

Прежде чем заняться программным кодом, сделайте шаг назад и подумайте о том, что вы хотите сделать концептуально. Идентифицировать палиндромы легко: надо лишь сравнить слово с ним сами, только пройденном в обратном порядке. Ниже приведен пример прохождения по слову, начиная с первой буквы до последней и затем начиная с последней буквы до первой:

```
>>> word = 'NURSES'
>>> word[:]
'NURSES'
>>> word[::-1]
'SESRUN'
```

Если во время прохождения по символьной цепочке (или любого другого индексируемого типа) значения не указаны, то по умолчанию используется начало цепочки, конец цепочки и положительный шаг, равный 1.

На рис. 2.1 показан процесс обратного прохождения. Я предоставил начальную позицию 2 и шаг –1. Поскольку конечный индекс не предоставлен (между двоеточиями нет индекса или пробела), то подразумевается, что нужно возвращаться назад (т. к. шаг индекса равен –1) до тех пор, пока больше не останется символов.

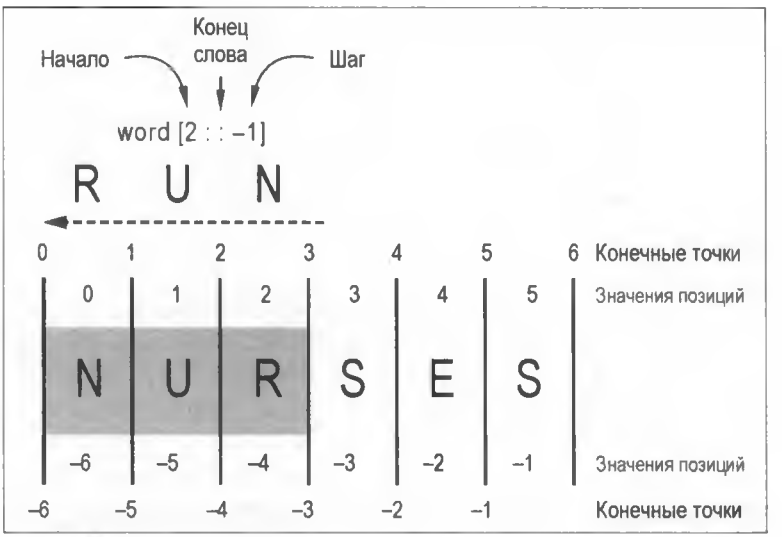


Рис. 2.1. Пример прохождения в обратном порядке для `word = 'NURSES'`

Прохождение в обратном порядке не ведет себя точно так же, как прямое прохождение, а положительные и отрицательные значения позиций и конечные точки являются асимметричными. Это может привести к путанице, поэтому давайте ограничимся нашим прохождением в обратном порядке в простом формате `[::-1]`.

Отыскание палиндромов в словаре займет меньше строк кода, чем загрузка файла словаря! Вот псевдокод:

```
Загрузить файл цифрового словаря в виде списка слов
Создать пустой список для хранения палиндромов
Перебрать каждое слово в списке слов:
  Если слово, пройденное прямо, совпадает со словом, пройденным обратно:
    Добавить слово в список палиндромов
Напечатать список палиндромов
```

Код отыскания палиндромов

Листинг 2.2 считывает файл словаря английского языка, определяет, какие слова являются палиндромами, сохраняет их в списке и распечатывает список одним махом как стопку элементов. Этот код можно скачать с веб-страницы книги по адресу **https://www.nostarch.com/impracticalpython/**. Вам также понадобится файл load_dictionary.py и файл словаря; сохраните все три файла в одной папке.

Листинг 2.2. Отыскание палиндромов в загруженном файле словаря. Файл palindromes.py

```
   """Отыскивает палиндромы (буквенные палинграммы) в файле словаря."""
❶ import load_dictionary
❷ word_list = load_dictionary.load('2of4brif.txt')
❸ pali_list = []

❹ for word in word_list:
      if len(word) > 1 and word == word[::-1]:
          pali_list.append(word)

      print("\nЧисло обнаруженных палиндромов = {}\n".
            format(len(pali_list)))
❺ print(*pali_list, sep='\n')
```

Начните с импорта файла load_dictionary.py как модуля ❶. Обратите внимание, что расширение .py при импорте не используется. Кроме того, указанный модуль находится в той же папке, что и этот скрипт, поэтому указывать путь к каталогу модуля не нужно. И поскольку модуль содержит необходимую строку `import sys`, вам не нужно ее здесь повторять.

Для того чтобы заполнить ваш список словами из словаря, вызовите функцию `load()` из модуля `load_dictionary` с помощью точечной формы записи ❷. Передайте ей имя внешнего файла словаря. Опять же, вам не нужно указывать путь, если файл словаря находится в той же папке, что и скрипт Python. Имя используемого вами файла может отличаться в зависимости от скачанного словаря.

Далее создайте пустой список для хранения палиндромов ❸ и начните перебирать слова в списке `word_list` ❹, сравнивая прямой фрагмент с обратным фрагментом. Если два фрагмента идентичны, то добавьте слово в список `pali_list`. Обратите внимание, что допускаются только те слова, которые имеют более одной буквы `(len(word) > 1)`, что соответствует самому строгому определению палиндрома.

Наконец, напечатайте палиндромы привлекательным способом — стопкой и без кавычек или запятых ❺. Это можно сделать, циклически перебирая каждое слово в списке, но для этого существует способ поэффективнее. Вы можете применить унарный оператор, обозначаемый звездочкой *[2], который на входе принимает список и в вызове функции разворачивает его в позиционные аргументы. Последний аргумент является разделителем, используемым между несколькими значениями списка для печати. Разделителем по умолчанию является пробел `(sep=' ')`, но вместо этого мы печатаем каждый элемент в новой строке `(sep='\n')`.

Однословные палиндромы встречаются редко, по крайней мере, в английском языке. Используя файл словаря из 60 тыс. слов, вам повезет найти около 60, или всего

[2] В английской терминологии такой оператор именуется splat (шлеп, плюх) operator, т. е. такой, который выполняет сложную операцию одним махом. — *Прим. перев.*

0,1% всех слов. Однако, несмотря на их редкость, с помощью языка Python их достаточно легко отыскать. Итак, давайте перейдем к более интересным и более сложным вещам — палинграммам.

Проект 3: поиск палинграмм

Отыскание палинграмм требует немного больше усилий, чем поиск однословных палиндромов. В этом разделе мы спланируем и напишем программный код для поиска палинграмм словарных пар.

Цель

Применить язык Python для отыскания двухсловных палинграмм в словаре английского языка. Проанализировать и оптимизировать программный код генерирования палинграмм с помощью инструмента cProfile.

Стратегия и псевдокод

Примером палинграммы словарных пар является nurses run (медсестры бегут) и stir grits (мешают крупу)[3]. (Если вам интересно, то grits — это блюдо для завтрака из молотой кукурузы, похожее на итальянскую поленту.)

Как и палиндромы, палинграммы читаются одинаково слева направо, и наоборот. Мне нравится думать о них как о стержневых словах, таких как nurses, из которых выводятся палиндромная последовательность и перевернутое слово (рис. 2.2).

Рис. 2.2. Рассечение палинграмм словарных пар

Наша программа будет экзаменовать стержневое слово. Основываясь на рис. 2.2, о стержневом слове можно сделать следующие выводы:

1. Оно может иметь четное либо нечетное число букв.
2. Одна смежная часть слова буквально содержит реальное слово, когда читается назад.

[3] Примеры палинграмм на русском языке: *ежу хуже* и *лев осовел*. — *Прим. перев.*

3. Эта смежная часть может занимать часть или все стержневое слово целиком.
4. Другая смежная часть содержит палиндромную последовательность букв.
5. Палиндромная последовательность может занимать часть или все стержневое слово целиком.
6. Палиндромная последовательность не обязательно должна быть реальным словом (если только она не занимает все слово целиком).
7. Эти две части не могут перекрываться либо использовать буквы совместно.
8. Последовательность обратима.

ПРИМЕЧАНИЕ

Если перевернутое слово занимает все стержневое слово и не является палиндромом, то оно называется *семорднилапом* (semordnilap). Семорднилап похож на палиндром, за исключением одного ключевого отличия: вместо того, чтобы содержать буквально одинаковое слово при чтении назад, он содержит другое слово. Примеры на английском языке — bats и stab (летучие мыши и колоть), wolf и flow (волк и поток). Кстати, семорднилап — это слово palindromes, написанное задом наперед.

На рис. 2.3 показано произвольное слово из шести букв. Крестики представляют собой часть слова, которая может образовывать реальное слово при чтении назад (например, run в слове nurses). Нолики представляют собой возможную палиндромную последовательность (например, ses в слове nurses). Слово, представленное в левой колонке на рис. 2.3, ведет себя как nurses на рис. 2.2, где перевернутое слово находится в начале. Слово, представленное правой колонкой, ведет себя как grits, где перевернутое слово находится в конце. Обратите внимание, что число комбинаций в каждом столбце — это суммарное число букв в слове плюс единица; заметьте также, что верхняя и нижняя строки представляют одинаковые ситуации.

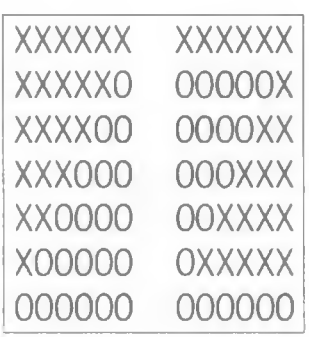


Рис. 2.3. Возможные позиции для букв перевернутого слова (X) и палиндромной последовательности (0) в шестибуквенном стержневом слове

Верхняя строка в каждом столбце представляет собой семорднилап. Нижняя строка в каждом из них — палиндром. Они оба являются перевернутыми словами, просто разными типами перевернутых слов. Следовательно, они считаются одной сущностью и оба могут быть выявлены с помощью одной строки кода в одной итерации цикла.

Для того чтобы увидеть эту диаграмму в действии, рассмотрим рис. 2.4, на котором изображены палинграммы devils lived (дьяволы жили) и retro porter (ретро-портье). Оба слова, devils и porter, являются стержневыми словами и зеркальными отражениями друг друга по отношению к палиндромным последовательностям и перевернутым словам. Сравните это с семорднилапом evil (зло) и палиндромом kayak (каяк).

```
DEVILS  PORTER
XXXXXO  OXXXXX

EVIL    LIVE      KAYAK   KAYAK
XXXX    XXXX      XXXXX   XXXXX
                  00000   00000
```

Рис. 2.4. Перевернутые слова (X) и палиндромные последовательности (0) в словах, семорднилапах и палиндромах

Палиндромы — это и перевернутые слова, и палиндромные последовательности. Поскольку они имеют тот же шаблон из крестиков, что и семорднилапы, их можно обрабатывать тем же кодом, который используется для семорднилапов.

С точки зрения стратегии вам нужно будет перебрать каждое слово в словаре и оценить его на предмет всех комбинаций на рис. 2.3. Исходя из того, что словарь имеет 60 тыс. слов, программа должна будет предпринять около 500 тыс. проходов по словарю.

Для понимания работы циклов взгляните на стержневое слово для палинграммы stack cats на рис. 2.5. Ваша программа должна перебрать буквы в слове, начиная с конечной буквы, и каждую итерацию добавлять букву. Для того чтобы отыскать такие палинграммы, как stack cats, она одновременно будет оценивать слово на наличие палиндромной последовательности в конце стержневого слова, stack, и пере-

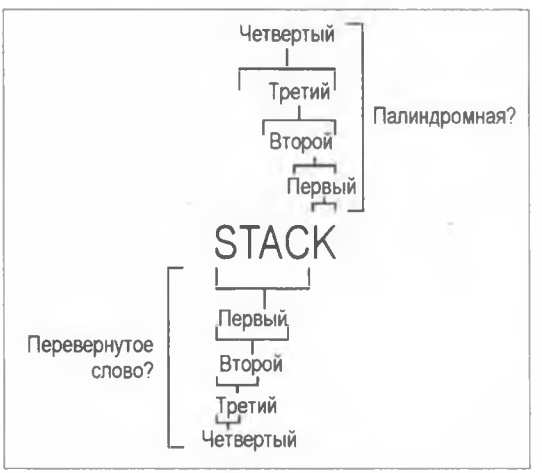


Рис. 2.5. Пример прокручивает стержневое слово, одновременно ища палиндромы и перевернутые слова

вернутого слова в начале. Обратите внимание, что первый цикл на рис. 2.5 будет успешным, т. к. в данной ситуации одиночная буква (k) может служить палиндромом.

Но вы еще не закончили. Для того чтобы уловить поведение "зеркального отражения" на рис. 2.3, вы должны выполнить циклы в обратном порядке, ища палиндромные последовательности в начале слова и перевернутые слова в конце. Это позволит найти такие палинграммы, как stir grits.

Вот псевдокод алгоритма отыскания палинграмм:

```
Загрузить цифровой словарь в виде списка слов
Создать пустой список, который будет содержать палинграммы
Для слова в списке слов:
  Получить длину слова,
  Если длина > 1:
    Перебрать буквы в слове:
      Если перевернутый фрагмент слова в начале слова
      находится в списке слов и последующие буквы
      образуют палиндромную последовательность:
        Добавить слово и перевернутое слово в список палинграмм
      Если фрагмент перевернутого слова в конце слова
      находится в списке слов и предыдущие буквы формируют
      палиндромную последовательность:
        Добавить перевернутое слово и слово в список палинграмм
Отсортировать список палинграмм в алфавитном порядке
Напечатать палинграммы словарных пар из списка палинграмм
```

Код отыскания палинграмм

Листинг 2.3 (файл palingrams.py) перебирает список слов, выявляет, какие слова образуют палинграммы словарных пар, сохраняет эти пары в списке и печатает список как стопку элементов. Код программы можно скачать с веб-страницы **https://www.nostarch.com/impracticalpython/**. Для начала предлагаю вам использовать файл словаря 2of4brif.txt, благодаря чему ваши результаты будут соответствовать моим. Сохраните словарь и файл load_dictionary.py в той же папке, что и скрипт palingrams.py.

Листинг 2.3. Отыскивает и печатает палинграммы словарных пар загруженного словаря. Файл palingrams.py

```
"""Отыскать все палинграммы словарных пар в файле словаря."""
import load_dictionary

word_list = load_dictionary.load('2of4brif.txt')

# отыскать палинграммы словарных пар
❶ def find_palingrams():
    """Отыскать палинграммы в словаре."""
```

```
        pali_list = []
        for word in word_list:
          ❷ end = len(word)
          ❸ rev_word = word[::-1]
          ❹ if end > 1:
              ❺ for i in range(end):
                  ❻ if word[i:] == rev_word[:end-i]and rev_word[end-i:] in
word_list:
                        pali_list.append((word, rev_word[end-i:]))
                  ❼ if word[:i] == rev_word[end-i:]and rev_word[:end-i] in
word_list:
                        pali_list.append((rev_word[:end-i], word))
      ❽ return pali_list

❾ palingrams = find_palingrams()

    # отсортировать палинграммы по первому слову
    palingrams_sorted = sorted(palingrams)

    # показать список палинграмм
❿ print("\nЧисло палинграмм = {}\n".format(len(palingrams_sorted)))
    for first, second in palingrams_sorted:
        print("{} {}".format(first, second))
```

После повторения шагов, которые вы использовали в программном коде palindromes.py для загрузки файла словаря, вы определяете функцию, которая будет выполнять поиск палинграмм ❶. Использование указанной функции позже позволит вам изолировать код и засечь время, требуемое для обработки всех слов в словаре.

Сразу же создайте список под названием `pali_list` для хранения всех палинграмм, обнаруженных программой. Затем начните цикл `for` для оценивания слов в списке `word_list`. Для каждого слова найдите его длину и назначьте ее переменной `end` ❷. Длина слова определяет индексы, которые программа использует для прохождения по слову в поисках каждой возможной комбинации "перевернутое слово — палиндромная последовательность", как на рис. 2.3.

Далее пройдите слово в обратном порядке и назначьте результаты переменной `rev_word` ❸. Альтернативой выражению `word[::-1]` является выражение `''.join(reversed(word))`, его некоторые считают более читаемым.

Поскольку вы ищете палинграммы словарных пар, исключите однобуквенные слова ❹. Затем вложите еще одну инструкцию `for` для того, чтобы перебрать буквы в текущем слове ❺.

Теперь выполните условное выражение, требующее, чтобы конец слова был палиндромным, а начало слова — перевернутым словом в списке слов (другими словами,

"реальным" словом) ❻. Если слово проверку проходит, то оно добавляется в список палинграмм, за которым сразу следует перевернутое слово.

Основываясь на рис. 2.3, вы знаете, что вам нужно повторить условие, но изменить направление прохождения и порядок слов для того, чтобы перевернуть результат. Другими словами, вы должны улавливать палиндромные последовательности в начале слова, а не в конце ❼. Завершая функцию, вы возвращаете список палинграмм ❽.

Определив функцию, вы ее вызываете ❾. Поскольку порядок, в котором слова словаря добавляются в список палинграмм, переключается во время цикла, палинграммы не будут в алфавитном порядке. Поэтому вы сортируете список так, чтобы первые слова в словарной паре были в алфавитном порядке. Далее печатаете длину списка ❿, а затем показываете каждую словарную пару в отдельной строке.

В том виде, как она написана, программе palingrams.py потребуется около трех минут на то, чтобы выполнить работу на файле словаря из порядка 60 тыс. слов. В следующих разделах мы разведаем причину этого длительного времени выполнения и посмотрим, что можно сделать для того, чтобы его сократить.

Профилирование палинграмм

Профилирование — это аналитический процесс, который собирает статистические данные о поведении программы (например, о числе и продолжительности вызовов функций) в ходе ее исполнения. Профилирование является ключевой частью процесса оптимизации. Оно точно говорит вам о том, какие части программы занимают бóльшую часть времени или памяти. Благодаря ему вы будете знать, на каких местах следует сосредоточить усилия с целью повышения производительности.

Профилирование с помощью *cProfile*

Профиль — это результат измерения — регистрация того, как долго и как часто выполняются части программы. Стандартная библиотека Python предоставляет удобный интерфейс профилирования, `cProfile`, являющийся расширением языка C, который подходит для профилирования длительных программ.

В функции `find_palingrams()`, вероятно, есть что-то, что объясняет относительно длительное время работы программы palingrams.py. Для подтверждения давайте запустим `cProfile`.

Скопируйте следующий ниже фрагмент кода в новый файл с именем cprofile_test.py и сохраните его в той же папке, что и palingrams.py и файл словаря. Указанный программный код импортирует `cProfile` и программу palingrams и запускает профилировщик `cProfile` на функции `find_palingrams()`, вызываемой при помощи точечной формы записи. Еще раз обратите внимание на то, что указывать расширение .py не нужно.

```
import cProfile
import palingrams
cProfile.run('palingrams.find_palingrams()')
```

Выполните скрипт cprofile_test.py, и после его завершения (в окне интерпретатора вы увидите >>>) вы должны увидеть что-то похожее на следующее:

```
     62622 function calls in 199.452 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.000   0.000 199.451 199.451 <string>:1(<module>)
     1 199.433 199.433 199.451 199.451 palingrams.py:7(find_palingrams)
     1   0.000   0.000 199.452 199.452 {built-in method builtins.exec}
 60388   0.018   0.000   0.018   0.000 {built-in method builtins.len}
  2230   0.001   0.000   0.001   0.000 {method 'append' of 'list' objects}
```

На моей машине вся циклическая обработка, прохождение слов и поиск заняли 199 452 секунды, но, разумеется, ваше время может отличаться от моего. Вы также получите дополнительную информацию о некоторых встроенных функциях, и поскольку каждая палинграмма вызывала встроенную функцию `append()`, вы даже увидите число найденных палинграмм (2230).

ПРИМЕЧАНИЕ

Общепринятым способом запуска профилировщика `cProfile` является его запуск непосредственно в интерпретаторе. Это позволяет сбрасывать результат в текстовый файл и просматривать его с помощью веб-браузера. Для получения дополнительной информации посетите веб-страницу **https://docs.python.org/37library/profile.html**.

Профилирование со временем

Еще один способ засечь время выполнения программы — использовать метод `time.time()`, который возвращает временную метку эпохи — число секунд с 12 часов 1 января 1970 года UTC (временную метку эпохи UNIX). Скопируйте файл palingrams.py в новый файл, сохраните его как palingrams_timed.py и вставьте следующий ниже фрагмент кода в самом верху:

```
import time
start_time = time.time()
```

Теперь перейдите в конец файла и добавьте следующий фрагмент кода:

```
end_time = time.time()
print("Время выполнения этой программы составило {} секунд.
                                                ".format(end_time - start_time))
```

Сохраните и запустите файл. Внизу окна интерпретатора вы должны получить следующий ниже отклик — плюс-минус несколько секунд:

```
Время выполнения этой программы составило 222.73954558372498 секунд.
```

Время выполнения будет больше, чем раньше, т. к. теперь вы оцениваете всю программу, включая печать, а не только функцию `find_palingrams()`.

В отличие от профилировщика `cProfile`, функция `time()` не предоставляет подробную статистику, но, как и `cProfile`, она может выполняться на отдельных компо-

нентах программного кода. Отредактируйте файл, который вы только что выполнили, переместив инструкции начального и конечного времени (как показано ниже жирным шрифтом) так, чтобы они обрамляли нашу длительную функцию `find_palingrams()`. Оставьте инструкции импорта и печати без изменений соответственно вверху и внизу файла.

```
start_time = time.time()
palingrams = find_palingrams()
end_time = time.time()
```

Сохраните и запустите файл. Внизу окна интерпретатора вы должны получить следующий ниже отклик:

```
Время выполнения этой программы составило 199.42786622047424 секунды.
```

Теперь он соответствует первоначальным результатам, полученным с помощью профилировщика `cProfile`. Если повторно выполнить программу или применить другой таймер, то получить в точности такое же время не удастся, но не зацикливайтесь на этом. Указанное время является *относительным*, и оно важно для регулирования процесса оптимизации кода.

Оптимизация палиндромных заклинаний

Простите, но три минуты моей жизни — это слишком долго, чтобы ждать, когда появятся палиндромные заклинания. Вооружившись нашими результатами профилирования, мы знаем, что функция `find_palingrams()` занимает бóльшую часть времени обработки. Вероятно, это имеет какое-то отношение к чтению списков и записи в них, нарезке списков или поиску в списках. Использование альтернативной спискам структуры данных — к примеру, кортежей, множеств или словарей — могло бы ускорить работу указанной функции. Множества, в частности, значительно быстрее списков при использовании ключевого слова `in`. Для выполнения очень быстрого поиска в множествах используется хэш-таблица. При хешировании символьные цепочки с текстом преобразуются в уникальные числа, которые намного меньше, чем адресуемый текст, и гораздо эффективнее для поиска. С другой стороны, в списке необходимо выполнять линейный поиск, перебирая каждый элемент.

Подумайте об этом так: если дома вы ищете свой потерянный мобильный телефон, то список можно эмулировать, просматривая каждую комнату, прежде чем вы его найдете (по закону подлости в пресловутом самом последнем месте, куда вы заглянете). Но, эмулируя множество, вы, в сущности, можете набрать номер своего мобильного телефона с другого телефона, услышать мелодию звонка и перейти прямо в нужную комнату.

Недостатком использования множеств является то, что порядок следования элементов в множестве не контролируется и дублирующие значения не допускаются. В списках порядок сохраняется, и дубликаты разрешены, но поиск занимает больше времени. К счастью для нас, порядок следования или дубликаты нас не интересуют, поэтому множества — это как раз то, что нам нужно!

Листинг 2.4 представляет собой функцию `find_palingrams()` из оригинальной программы palingrams.py, отредактированную для использования множества слов вместо списка слов. Указанную функцию можно найти в новой программе palingrams_optimized.py, которую можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**, либо просто внесите следующие ниже изменения в свою копию программы palingrams_timed.py, в случае если хотите проверить новое время выполнения самостоятельно.

Листинге 2.4. Функция `find_palingrams()`, оптимизированная для работы с множествами

```
def find_palingrams():
    """Отыскать палинграммы в словаре."""
    pali_list = []
❶   words = set(word_list)
❷   for word in words:
        end = len(word)
        rev_word = word[::-1]
        if end > 1:
            for i in range(end):
❸               if word[i:] == rev_word[:end-i] and rev_word[end-i:] in words:
                    pali_list.append((word, rev_word[end-i:]))
❹               if word[:i] == rev_word[end-i:] and rev_word[:end-i] in words:
                    pali_list.append((rev_word[:end-i], word))
    return pali_list
```

Изменены всего четыре строки. Определите новую переменную `words`, которая представляет множество `word_list` ❶. Затем выполните обход множества ❷, а не списка, как раньше, в поисках членства фрагментов слова в этом множестве ❸❹.

Вот новое время выполнения функции `find_palingrams()` в программе palingrams_optimized.py:

```
Время выполнения этой программы составило 0.4858267307281494 секунды.
```

Ух ты! С более 3 минут до менее 1 секунды! Вот это оптимизация! И разница лишь в структуре данных. Проверка принадлежности слова в списке была бы просто убийственной.

Почему я с самого сначала показал вам "неправильный" способ? Потому что именно так все происходит в реальном мире. Вы доводите программный код до рабочего состояния, а затем его оптимизируете. Опытный программист реализовал бы этот простой пример с пол-оборота, но он символизирует общую концепцию оптимизации: заставить его работать как можно лучше, а затем сделать его еще лучше.

Резюме

Вы написали программный код для поиска палиндромов и палинграмм, профилированный код с применением инструмента `cProfile` и оптимизированный код с использованием структуры данных, лучшим образом соответствующей задаче. Ну и как же мы поступим с Затанной? Есть ли у нее шанс побороться?

Здесь я перечислил несколько наиболее "агрессивных" палинграмм, найденных в файле словаря 2of4brif — тут все, начиная с неожиданной sameness enemas (клизмы одинаковости) и суровой torsos rot (торсы гниют) до близкой для меня как геолога: eroded ore (эродированная руда).

```
dump mud          drowsy sword       sameness enemas
legs gel          denims mined       lepers repel
sleet eels        dairy raid         slam mammals
eroded ore        rise sir           pots nonstop
strafe farts      torsos rot         swan gnaws
wolfs flow        partner entrap     nuts stun
slaps pals        flack calf         knobs bonk
```

Дальнейшее чтение

Книга "Думай по-питоновски" (Downey A. Think Python. O'Reilly, 2015) Аллена Дауни содержит краткое и ясное описание хэш-таблиц и причин, почему они являются такими эффективными. Она также является отличным справочником по языку Python.

Практический проект: очистка словаря

Файлы данных, доступные в Интернете, не всегда находятся в формате plug and play (подключи и играй). Прежде чем применять их в вашем проекте, вы вполне можете обнаружить, что вам необходимо немного отрегулировать данные. Как упоминалось ранее, некоторые файлы онлайновых словарей содержат каждую букву алфавита в качестве слова. Это вызовет проблемы, если в палинграммах вы захотите разрешить использовать однобуквенные слова, как, например, в "acidic a". Их всегда можно удалить из словаря, непосредственно отредактировав текстовый файл, но такой способ утомителен и для неудачников. Вместо него напишите короткий скрипт, который удаляет их после загрузки словаря в Python. Для проверки того, что все работает, отредактируйте файл словаря так, чтобы он включал несколько однобуквенных слов, таких как *b* и *c*. Решение см. в приложении к книге либо найдите копию (dictionary_cleanup_practice.py) онлайн по адресу **https://www.nostarch.com/impracticalpython/**.

Сложный проект: рекурсивный подход

В языке Python, как правило, существует более одного способа расколоть орех. Взгляните на обсуждение и псевдокод на веб-сайте Khan Academy:

https://www.khanacademy.org/computing/computer-cience/algorithms/recursive-algorithms/a/using-recursion-to-determine-whether-a-word-is-a-palindrome/.

Затем перепишите программу palindrome.py так, чтобы для выявления палиндромов в ней использовалась рекурсия.

3

РЕШЕНИЕ АНАГРАММ

Анаграмма — это слово, образованное путем перестановки букв другого слова. Например, Elvis дает жуткое трио из evils (зло), lives (жизни) и veils (завесы). Означает ли это, что Элвис по-прежнему жив, но скрывает свое злое существование? В книге "Гарри Поттер и Тайная комната" предложение "I am Lord Voldemort" (Я — Лорд Волдеморт) является анаграммой настоящего имени злого волшебника, Тома Марволо Реддла (Tom Marvolo Riddle). Имя "Lord Earldom Vomit" также является анаграммой имени Тома Марволо Реддла, но у автора Джоан Роулинг хватило здравого смысла ее переделать (vomit означает рвота).

В этой главе вы сначала найдете все анаграммы для заданного слова или имени. Затем вы напишете программу, которая позволит пользователю интерактивно строить анаграммное словосочетание из собственного имени. Наконец, вы поиграете в компьютерного мастера слова и увидите, что нужно для того, чтобы извлечь "I am Lord Voldemort" из "Tom Marvolo Riddle".

Проект 4: поиск однословных анаграмм

Вы начнете с анализа простых однословных анаграмм и выяснения того, как выявлять их программно. Выполнив это, вы будете готовы взяться за анаграммы словосочетаний в следующем разделе.

Цель

Применить язык Python и файл словаря для отыскания всех однословных анаграмм для заданного английского слова или одного имени. Инструкции по отысканию и загрузке файлов словарей можно прочитать в начале *главы 2*.

Стратегия и псевдокод

Более 600 газет и 100 интернет-сайтов содержат анаграмму под названием Jumble (Беспорядок). Созданная в 1954 г., она теперь стала самой признанной в мире игрой для беспорядочного перемешивания букв в словах (скремблирования слов). Игра Jumble может по-настоящему доводить до исступления, однако отыскивать анаграммы почти так же легко, как отыскивать палиндромы — просто нужно знать общее свойство всех анаграмм: они должны иметь одинаковое число одинаковых букв.

Идентификация анаграммы

Python не содержит встроенного оператора анаграмм, но вы можете легко его написать. В проектах этой главы вы загрузите файл словаря из *главы 2* в виде списка символьных цепочек. Так что программе необходимо лишь удостовериться, что две символьные цепочки являются анаграммами друг друга.

Давайте рассмотрим пример. Слово pots является анаграммой слова *stop*, и с помощью функции `len()` вы можете проверить, что *stop* и *pots* имеют одинаковое число букв. Но Python не знает, имеют или нет две символьные цепочки одинаковое число любого символа в слове — по крайней мере, без преобразования цепочек в другую структуру данных либо с помощью функции подсчета. Поэтому вместо того, чтобы рассматривать эти два слова просто как символьные цепочки, вы можете представить их как два списка, содержащих односимвольные цепочки. Создайте эти списки в интерактивной оболочке, к примеру в окне интерпретатора IDLE, и назовите их `word` и `anagram`, как я сделал здесь:

```
>>> word = list('stop')
>>> word
['s', 't', 'o', 'p']
>>> anagram = list('pots')
>>> anagram
['p', 'o', 't', 's']
```

Эти два списка соответствуют нашему описанию пары анаграмм, т. е. они содержат одинаковое число одинаковых букв. Но если вы попытаетесь сравнить их с помощью оператора сравнения `==`, то результат будет ложным.

```
>>> anagram == word
False
```

Проблема заключается в том, что оператор == рассматривает два списка как эквивалентные только в том случае, если они имеют одинаковое число одинаковых элементов и эти элементы выстроены в одинаковом порядке. Эту проблему можно легко решить с помощью встроенной функции `sorted()`, которая в качестве аргумента может принимать список и переупорядочивать его содержимое в алфавитном порядке. Итак, если вызвать `sorted()` дважды — по одному разу для каждого списка, а затем сравнить отсортированные списки, то они будут эквивалентными. Другими словами, оператор == вернет `True`.

```
>>> word = sorted(word)
>>> word
['o', 'p', 's', 't']
>>> anagram = sorted(anagram)
>>> anagram
['o', 'p', 's', 't']
>>> anagram == word
True
```

Кроме того, в функцию `sorted()` можно передать символьную цепочку, в результате создав отсортированный список, как в предыдущем фрагменте кода. Это окажется полезным для преобразования слов из файла словаря в отсортированные списки односимвольных цепочек.

Теперь, когда вы знаете, как проверить, что анаграмма найдена, давайте создадим скрипт в полном объеме — от загрузки словаря и запроса слова (или имени) у пользователя до отыскания и печати всех анаграмм.

Использование псевдокода

Напомним, что планирование с помощью псевдокода поможет вам определить потенциальные проблемы, а раннее выявление этих проблем сэкономит вам время. Следующий ниже псевдокод должен помочь вам лучше понять скрипт anagrams.py, который мы напишем в следующем разделе.

```
Загрузить файл цифрового словаря в виде списка слов
Принять слово от пользователя
Создать пустой список для хранения анаграмм
Отсортировать буквы введенного пользователем слова
Перебрать каждое слово в списке слов:
  Отсортировать буквы слова
  Если отсортированное слово тождественно отсортированному слову пользователя:
    Добавить слово в список анаграмм
Напечатать список анаграмм
```

Указанный скрипт начнется с загрузки слов из файла словаря в список в виде символьных цепочек. Прежде чем обойти словарь в цикле в поисках анаграмм, вам нужно знать, анаграммы какого слова вам нужны, и вам требуется место для хранения анаграмм, когда вы их найдете. Поэтому сначала попросим пользователя ввести слово, а затем создадим пустой список для хранения анаграмм. После того как программа переберет каждое слово в словаре, она напечатает этот список анаграмм.

Код отыскания анаграмм

Листинг 3.1 загружает файл словаря, принимает указанное в программе слово или имя и отыскивает для этого слова или имени все анаграммы в файле словаря. Вам также понадобится программный код загрузки словаря из *главы 2*. Указанный код можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/** соответственно в виде файлов anagrams.py и load_dictionary.py. Сохраните оба файла в од-

ной папке. Можно использовать тот же файл словаря, который вы использовали в *главе 2*, либо загрузить другой (рекомендации см. в табл. 2.1).

Листинг 3.1. С учетом слова (либо имени) и файла словаря эта программа отыскивает и распечатывает список анаграмм. Файл anagrams.py

```
❶ import load_dictionary

❷ word_list = load_dictionary.load('2of4brif.txt')

❸ anagram_list = []

  # ниже ввести ОДНО слово либо ОДНО имя для отыскания анаграмм(ы):
❹ name = 'Foster'
  print("Входное имя = {}".format (name))
❺ name = name.lower()
  print("Используя имя = {}".format(name))

  # отсортировать имя и отыскать анаграммы
❻ name_sorted = sorted(name)
❼ for word in word_list:
      word = word.lower()
      if word != name:
          if sorted(word) == name_sorted:
              anagram_list.append(word)
  # распечатать список анаграмм
  print()
❽ if len(anagram_list) == 0:
      print("Вам нужны более крупный словарь либо новое имя!")
  else:
  ❾   print("Анаграммы =", *anagram_list, sep='\n')
```

Начните с импорта модуля `load_dictionary`, созданного в *главе 2* ❶. Указанный модуль откроет текстовый файл словаря и с помощью функции `load()` загрузит все слова в список ❷. Используемый вами txt-файл может отличаться в зависимости от того, какой файл словаря вы загрузили *(см. разд. "Отыскание и открытие словаря" в главе 2)*.

Далее создайте пустой список под названием `anagram_list` для хранения любых анаграмм, которые вы найдете ❸. Попросите пользователя добавить одно слово, например свое имя ❹. Указанное слово не обязательно должно быть именем собственным, но в программном коде мы будем ссылаться на него как на имя для того, чтобы отличать его от слова словаря. Напечатайте это имя для того, чтобы пользователь смог увидеть, какое имя было введено.

Следующая строка кода предвосхищает проблемное действие пользователя. Люди, как правило, вводят свое имя с заглавной буквы, но файлы словарей могут не со-

держать заглавные буквы, а для Python это имеет значение. Поэтому с помощью метода символьных цепочек `lower()` сначала переведите все буквы в нижний регистр ❺.

Теперь отсортируйте буквы имени ❻. Как упоминалось ранее, в функцию `sorted()` можно передавать символьную цепочку или список.

Имея список отсортированных по алфавиту данных, самое время отыскать анаграммы. Начните цикл с обходом каждого слова в списке слов словаря ❼. Для подстраховки конвертируйте слово в нижний регистр, т. к. операции сравнения чувствительны к регистру. После конвертирования сравните слово с неотсортированным именем, т. к. слово не может быть анаграммой самого себя. Затем отсортируйте слово словаря и сравните его с отсортированным именем. Если сравнение проходит, то добавьте это слово словаря в список анаграмм `anagram_list`.

Теперь покажите результаты. Сначала проверьте, является ли список анаграмм пустым. Если это так, то распечатайте причудливый ответ, для того чтобы не оставить пользователя сидеть перед пустым экраном ❽. Если программа нашла хотя бы одну анаграмму, то напечатайте список с помощью унарного оператора "звездочка" (`*`). Как вы помните из *главы 2*, указанный оператор позволяет печатать каждый элемент списка на отдельной строке ❾.

Ниже приведен пример результата работы этой программы с использованием входного имени Foster.

```
Входное имя = Foster
Используя имя = foster

Анаграммы =
forest
fortes
softer
```

Если вы хотите использовать другое входное значение, то в исходном коде измените значение переменной `name`. В качестве упражнения попробуйте настроить программный код так, чтобы пользователю предлагалось вводить имя (или слово); это можно сделать с помощью функции `input()`.

Проект 5: поиск фразовых анаграмм

В предыдущем проекте вы брали одно-единственное имя или слово, переставляли все буквы и отыскивали однословные анаграммы. Теперь вы будете выводить из имени несколько слов. Слова в этих фразовых анаграммах образуют только часть входного имени, и для того, чтобы исчерпать доступные буквы, вам понадобится несколько слов.

Цель

Написать программу на языке Python, которая позволяет пользователю в интерактивном режиме строить анаграммное словосочетание из букв в его имени.

Стратегия и псевдокод

Лучшие фразовые анаграммы — это те, что описывают какую-то известную особенность или действие, которые ассоциированы с носителем имени. Например, буквы в имени Clint Eastwood могут быть перестановлены так, чтобы сформировать словосочетание old west action (меры старого Запада), имя Alec Guinness даст словосочетание genuine class (подлинный класс), Madam Curie произведет предложение radium came (радий пришел), George Bush даст he bugs Gore (он прослушивает Гора), а Statue of Liberty (статуя Свободы) будет содержать словосочетание built to stay free (построенная для того, чтобы мы оставались свободными). Мое собственное имя дает a huge navel (огромный пупок), который на самом деле не является ни одной из моих особенностей.

На этом этапе вы, возможно, увидите, что перед нами возникает одна стратегическая трудность: каким образом компьютер сможет обрабатывать контекстное содержимое? Сотрудники компании IBM, которые изобрели суперкомпьютер Watson, похоже, это знают, но для остальных из нас этот валун будет поднять трудновато.

Метод грубой силы (или исчерпывающего перебора всех вариантов) — это широко применяемый подход, который используется в онлайновых генераторах анаграмм. Их алгоритмы принимают имя и возвращают большое число случайных анаграммных словосочетаний (как правило, от 100 до 10 000+). Большинство возвращаемых словосочетаний является тарабарщиной, и пролистывание сотен из них может оказаться муторным делом.

Альтернативный подход — признать факт, что люди лучше всего справляются с контекстуальными задачами, и написать программу, которая помогает человеку работать над решением этой задачи. Компьютер берет начальное имя и предоставляет слова, которые будут собраны из нескольких (или всех) его букв; пользователь выбирает слово, которое для него "имеет смысл". Затем программа пересчитывает варианты выбора слов из остальных букв в имени, повторяя процесс до тех пор, пока не будет использована каждая буква или не будут исчерпаны возможные варианты слов. Такая конструкция подыгрывает сильным сторонам обоих участников.

Вам понадобится простой интерфейс, который предложит пользователю ввести начальное имя, покажет потенциальные варианты слов и покажет все остальные буквы. Программа также должна отслеживать растущее анаграммное словосочетание и сообщать пользователю о том, когда каждая буква была использована. Вероятно, по ходу возникнет много неудачных попыток, поэтому интерфейс должен позволить пользователю перезапускать процесс в любое время.

Поскольку анаграммы имеют одинаковое число одинаковых букв, другой способ их выявления состоит в подсчете индивидуальных букв. Если подумать о своем имени как о коллекции букв, то из вашего имени можно построить слово, если, во-первых, все его буквы встречаются в вашем имени и, во-вторых, они встречаются с одинаковой частотой или меньше. Очевидно, что если буква *e* встречается в слове три раза, а в вашем имени — дважды, то из вашего имени слово получить невозможно.

Поэтому если коллекция букв, составляющих слово, не является подмножеством коллекции букв вашего имени, то это слово не может быть частью анаграммы вашего имени.

Использование контейнерного типа *Counter* для подсчета букв

К счастью для нас, Python поставляется с модулем под названием `collections`, который включает в себя несколько контейнерных типов данных. Один из таких типов, `Counter`, подсчитывает вхождения элемента. Python хранит элементы как ключи словаря, а количества — как его значения. Например, следующий ниже фрагмент кода подсчитывает количество находящихся в списке видов деревьев бонсай.

```
>>> from collections import Counter
❶ >>> my_bonsai_trees = ['maple', 'oak', 'elm', 'maple', 'elm', 'elm', 'elm',
                         'elm']
❷ >>> count = Counter(my_bonsai_trees)
>>> print(count)
❸ Counter({'elm': 5, 'maple': 2, 'oak': 1})
```

Список `my_bonsai_trees` содержит многократные значения одного и того же типа деревьев ❶. Контейнер `Counter` подсчитывает деревья ❷ и создает простой в обращении словарь ❸. Обратите внимание, что функция `print()` является необязательной и используется здесь для ясности. Если ввести просто `count`, то содержимое словаря будет показано тоже.

Контейнерный тип `Counter` можно использовать для отыскания однословных анаграмм вместо метода `sorted()`. Вместо двух отсортированных списков на выходе будут два словаря, которые также можно напрямую сравнить с помощью оператора `==`. Ниже приведен пример:

```
>>> name = 'foster'
>>> word = 'forest'
>>> name_count = Counter(name)
>>> print(name_count)
❶ Counter({'f': 1, 't': 1, 'e': 1, 'o': 1, 'r': 1, 's': 1})
>>> word_count = Counter(word)
>>> print(word_count)
❷ Counter({'f': 1, 't': 1, 'o': 1, 'e': 1, 'r': 1, 's': 1})
```

Контейнерный тип `Counter` для каждого слова создает словарь, который увязывает каждую букву в слове с количеством ее появлений ❶❷. Словари не отсортированы, но несмотря на отсутствие сортировки, Python правильно идентифицирует каждый словарь как тождественный другому, если словари содержат одни и те же буквы и одинаковые их количества:

```
>>> if word_count == name_count:
        print("Совпадение!")

Совпадение!
```

Контейнерный тип `Counter` дает вам прекрасный способ отыскивать слова, которые "укладываются" в имя. Если количество каждой буквы в слове меньше или равно количеству той же буквы в имени, то слово можно получить из имени!

Псевдокод

Теперь мы приняли два важных проектных решения: 1) дать пользователю возможность интерактивно строить свою анаграмму по одному слову за раз; 2) использовать контейнерный тип `Counter` для отыскания анаграмм. Этого достаточно для того, чтобы начать думать о высокоуровневом псевдокоде:

```
Загрузить файл словаря
Принять имя от пользователя
Установить предел равным длине имени
Создать пустой список для хранения анаграммного словосочетания
До тех пор, пока длина словосочетания < предел:
  Сгенерировать список слов словаря, которые укладываются в имя
  Показать пользователю слова
  Показать пользователю остальные буквы
  Показать пользователю текущее словосочетание
  Попросить пользователя ввести слово или начать все сначала
  Если пользователь может ввести данные из остальных букв:
    Принять от пользователя вариант нового слова или слов
    Удалить буквы в выбранном варианте букв в имени
    Вернуть выбранный вариант и остальные буквы в имени
  Если выбранный вариант не является допустимым:
    Попросить пользователя сделать другой выбор либо дать
    пользователю возможность начать сначала
  Добавить выбранный вариант в словосочетание и показать пользователю
  Сгенерировать новый список слов и повторить процесс
Когда длина словосочетания равна значению предела:
  Показать последнее словосочетание
  Попросить пользователя начать все сначала либо выйти из программы
```

Дележ обязанностей

По мере усложнения процедурного кода возникает необходимость во встраивании (инкапсуляции) значительной его части в функции. Это упрощает управление вводом и выводом, выполнение рекурсии и чтение кода.

Главная функция — это то место, где программа начинает исполнять инструкции и обеспечивает высокоуровневую организацию, а именно управление всеми кусочками и частями кода, включая работу с пользователем. В программе генерирования фразовых анаграмм главная функция обертывает все функции-работяги, принимает подавляющую часть вводимых пользователем данных, отслеживает растущее анаграммное словосочетание, определяет момент, когда словосочетание завершено, и показывает пользователю результат.

Схематичный набросок подзадач и их потока (например, "графический псевдокод") является отличным способом выяснить, что конкретно вы хотите сделать и где. На рис. 3.1 показана блок-схема с выделенными предназначениями функций. В этом случае достаточно трех функций: `main()`, `find_anagrams()` и `process_choice()`.

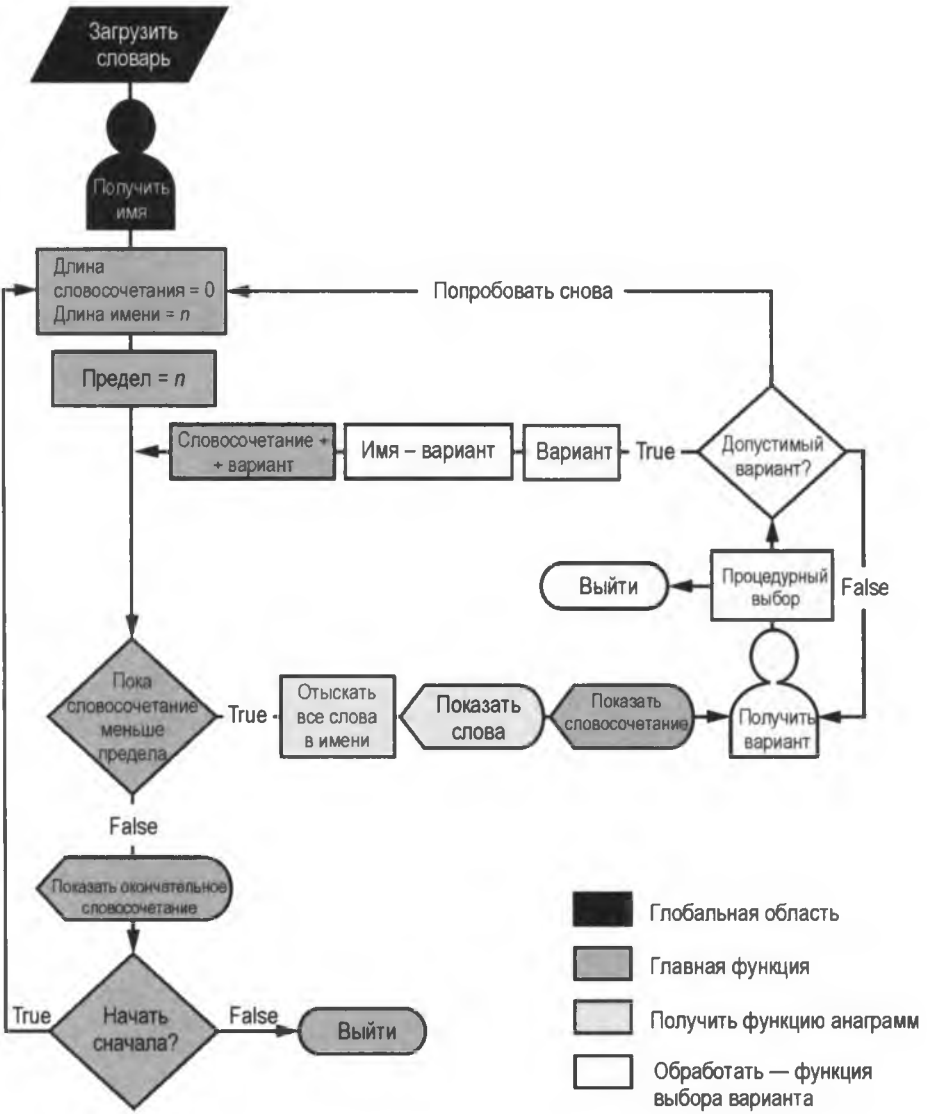


Рис. 3.1. Блок-схема отыскания фразовых анаграмм с выделенными предназначениями функций

Первостепенная задача главной функции, функции `main()`, — установить лимит количества букв и управлять циклом `while`, ответственным за построение общей фразовой анаграммы. Функция `find_anagrams()` возьмет текущую коллекцию букв, оставшихся в имени, и вернет все возможные слова, которые могут быть составлены из этих букв. Затем слова показываются пользователю наряду с текущим слово-

сочетанием, которым "владеет" и которое выводит на экран функция `main()`. Затем функция `process_choice()` предлагает пользователю начать сначала или выбрать слово для анаграммного словосочетания. Если пользователь делает выбор, то эта функция определяет наличие букв в выбранном варианте. Если их нет, то пользователю будет предложено выбрать еще раз либо начать все сначала. Если пользователь делает правильный выбор, то буквы в выбранном пользователем варианте удаляются из списка остальных букв и возвращаются выбранный вариант и список остальных букв. Функция `main()` добавляет возвращенный вариант в существующее словосочетание. Если достигнут предел, то печатается завершенная фразовая анаграмма и пользователю предлагается начать все сначала либо выйти из программы.

Обратите внимание, что начальное имя запрашивается не в функции `main()`, а в *глобальной* области. Это позволяет пользователю в любое время начать все заново без необходимости повторно вводить имя. На данный момент, в случае если пользователь захочет выбрать совершенно новое имя, ему придется выйти из программы и начать все сначала. В *главе 9* вы примените систему меню, которая позволяет пользователю, не выходя из программы, полностью сбрасывать то, что он делает.

Код генерирования анаграммных словосочетаний

Программный код этого раздела принимает от пользователя имя и помогает ему построить анаграммное словосочетание этого имени. Весь сценарий можно скачать с веб-страницы **https://www.nostarch.com/impracticalpython/** как phrase_anagrams.py. Вам также необходимо скачать программу load_dictionary.py. Сохраните оба файла в одной папке. Можно использовать тот же файл словаря, который вы использовали в *разд. "Проект 4: поиск однословных анаграмм" главы 3*.

Настройка и отыскание анаграмм

Листинг 3.2 импортирует модули, используемые в скрипте phrase_anagrams.py, загружает файл словаря, запрашивает у пользователя входное имя и определяет функцию `find_anagrams()`, которая выполняет бóльшую часть работы, связанной с отысканием анаграмм.

Листинг 3.2. Импортирует модули, загружает словарь и определяет функцию `find_anagrams`. Файл phrase_anagrams.py, часть 1

```
import sys
from collections import Counter
import load_dictionary

dict_file = load_dictionary.load('2of4brif.txt')
# обеспечить включение букв "a" и "i" (обе в нижнем регистре)
dict_file.append('a')
dict_file.append('i')
dict_file = sorted(dict_file)
```

```
❸ ini_name = input("Введите имя: ")

❹ def find_anagrams(name, word_list):
      """Прочитать имя и файл словаря и показать все анаграммы
      в имени."""
  ❺   name_letter_map = Counter(name)
      anagrams = []
  ❻   for word in word_list:
  ❼       test = ''
  ❽       word_letter_map = Counter(word.lower())
  ❾       for letter in word:
              if word_letter_map[letter] <= name_letter_map[letter]:
                  test += letter
              if Counter(test) == word_letter_map:
                  anagrams.append(word)
  ❿   print(*anagrams, sep='\n')
      print()
      print("Остальные буквы = {}".format(name))
      print("Число остальных букв = {}".format(len(name)))
      print("Число остальных анаграмм (реальных слов) = {}".
            format(len(anagrams)))
```

Начните с инструкций импорта ❶, используя рекомендуемый порядок следования: стандартная библиотека языка Python, сторонние модули, а затем локально разработанные модули. Модуль `sys` нужен для окраски конкретных результатов красным цветом в окне интерпретатора IDLE и для обеспечения пользователя возможностью выхода из программы нажатием клавиши. Для идентификации анаграмм входного имени вы будете использовать контейнерный тип `Counter`.

Далее загрузите файл словаря с помощью импортированного модуля ❷. Аргумент `filename` должен быть именем файла используемого вами словаря. Поскольку некоторые файлы словарей пропускают буквы *a* и *I*, добавьте их (если это необходимо) и отсортируйте список так, чтобы их можно было найти в соответствующих алфавитных местах, а не в конце списка.

Теперь получите от пользователя имя и назначьте его переменной `ini_name` (или "начальному имени") ❸. Из этого начального имени вы будете выводить переменную `name`, постепенно ее изменяя по мере того, как пользователь строит анаграмму имени. Хранение начального имени в качестве отдельной переменной позволит вам все обнулить, если пользователь захочет начать все сначала либо повторить попытку.

Следующим блоком кода является функция отыскания анаграмм в имени `find_anagrams()` ❹. Параметры этой функции состоят из имени и списка слов. Указанная функция начинает работу с использования контейнерного типа `Counter` для подсчета количества появлений заданной буквы в имени, а затем назначает это количество переменной `name_letter_map` ❺; контейнерный тип `Counter` использует

структуру словаря, в которой буква является ключом, а количество — значением. Затем эта функция создает пустой список для хранения анаграмм и начинает цикл `for`, перебирая каждое слово в файле словаря ❻.

Цикл `for` начинается с создания пустой символьной цепочки с именем `test` ❼. Используйте эту переменную для накопления всех букв в слове, которые "укладываются" в переменную `name`. Затем создайте для текущего слова тип `Counter`, как вы сделали для переменной `name`, и назовите его `word_letter_map` ❽. Переберите буквы в слове `word` ❾, проверяя, чтобы количество каждой буквы совпадало либо было меньше, чем количество в `name`. Если буква этому условию соответствует, то она добавляется в тестовую цепочку. Поскольку некоторые буквы могли быть отклонены, завершите цикл, выполнив `Counter` на тестовой цепочке `test` и сравнив ее с `word_letter_map`. Если они совпадают, то добавьте слово в список анаграмм.

Указанная функция заканчивается выводом на экран списка слов, используя в инструкции `print` унарный оператор "звездочка", вместе с некоторой статистикой для пользователя ❿. Обратите внимание, что `find_anagrams()` ничего не возвращает. Именно тут на сцену выходит взаимодействие с человеком. Программа будет продолжать работать, но ничего не будет происходить до тех пор, пока пользователь не выберет слово из выводимого на экран списка.

Обработка выбранного варианта

В листинге 3.3 определяется функция `process_choice()` из программы phrase_anagrams.py, которая принимает выбранный пользователем вариант слова (либо варианты слов), проверяет его на остальные буквы в переменной `name` и возвращает приемлемые варианты в функцию `main()` — вместе с любыми остальными буквами. Как и функция `main()`, указанная функция общается с пользователем напрямую.

Листинг 3.3. Определяет функцию `process_choice()`. Файл phrase_anagrams.py, часть 2

```
❶ def process_choice(name):
      """Проверить вариант пользователя на допустимость,
      вернуть вариант и остальные буквы."""
      while True:
          msg = '\nВыберите либо введите Enter, чтобы начать сначала, либо #
                                                                  для выхода '
       ❶ choice = input(msg)
          if choice == '':
              main()
          elif choice == '#':
              sys.exit()
          else:
       ❸ candidate = ''.join(choice.lower().split())
       ❶ left_over_list = list(name)
```

```
    ❺ for letter in candidate:
          if letter in left_over_list:
              left_over_list.remove(letter)
    ❻ if len(name) - len(left_over_list) == len(candidate):
          break
      else:
          print("Не сработает! Попробовать еще один вариант!",
                file=sys.stderr)
❼ name = ''.join(left_over_list)  # делает выводимое на экран
                                  # более читаемым
❽ return choice, name
```

Начните с определения функции одного параметра с именем `name` ❼. В первый раз после запуска программы этот параметр будет таким же, как переменная `ini_name` — полное имя, введенное пользователем при запуске программы. После того как пользователь выбрал слово (либо слова) для использования в анаграммном словосочетании, он будет представлять остальные буквы в имени.

Начните функцию циклом `while`, который будет выполняться до тех пор, пока пользователь не сделает правильный выбор, а затем получите от пользователя входное значение ❽. У пользователя есть выбор: ввести одно или несколько слов из текущего списка анаграмм, нажать клавишу <Enter> для того, чтобы начать все сначала, либо нажать клавишу <#> для того, чтобы выйти из программы. Используйте символ #, а не слово или букву, для того чтобы не спутать его с действительным вариантом выбора.

Если пользователь делает выбор, то введенная символьная цепочка назначается переменной `candidate`, очищается от пробелов и вся переводится в нижний регистр ❼. Это значит, что ее можно напрямую сравнить с переменной `name`. После этого из переменной `name` строится список, содержащий все остальные буквы ❺.

Теперь начните цикл, для того чтобы вычесть буквы, используемые в переменной `candidate` ❺. Если выбранная буква в списке присутствует, то она удаляется.

Если пользователь ввел слово, которого в показываемом списке нет, либо он ввел несколько слов, то буква может отсутствовать в списке. Для того чтобы это проверить, вычтите остальные буквы из имени, и если результатом будет количество букв в `candidate`, то входные данные считаются допустимыми и можно выйти из цикла `while` ❻. В противном случае покажите предупреждение и для тех, кто использует окно интерактивной оболочки IDLE, выведите это предупреждение шрифтом красного цвета. Цикл `while` будет продолжать опрашивать пользователя до тех пор, пока не будет сделан приемлемый выбор.

Если проверку проходят все буквы в выбранном пользователем варианте, то список оставшихся букв конвертируется обратно в символьную цепочку и используется для обновления переменной `name` ❼. Конвертирование списка в символьную цепочку не является строго необходимым, но оно поддерживает единообразным тип переменной `name` и позволяет показывать остальные буквы в четко читаемом формате без необходимости дополнительных аргументов инструкции `print`.

Завершите действия, вернув выбранный пользователем вариант и цепочку остальных букв (name) в функцию main() ❽.

Определение функции *main()*

Листинг 3.4 определяет функцию main() в программе phrase_anagrams.py. Указанная функция обертывает предыдущие функции, запускает цикл while и определяет момент успешного создания пользователем анаграммного словосочетания.

Листинг 3.4. Определяет и вызывает функцию main(). Файл phrase_anagrams.py, часть 3

```
def main():
    """Помочь пользователю построить анаграммное словосочетание
       из своего имени."""
❶   name = ''.join(ini_name.lower().split())
    name = name.replace('-', '')
❾   limit = len(name)
    phrase = ''
    running = True

❸   while running:
❶       temp_phrase = phrase.replace(' ', '')
❶       if len(temp_phrase) < limit:
            print("Длина анаграммного словосочетания = {}".
                  format(len(temp_phrase)))

❷           find_anagrams(name, dict_file)
            print("Текущее анаграммное словосочетание =", end=" ")
            print(phrase, file=sys.stderr)

❸           choice, name = process_choice(name)
            phrase += choice + ' '

❹       elif len(temp_phrase) == limit:
            print("\n***** ГОТОВО!!! *****\n")
            print("Анаграмма имени =", end=" ")
            print(phrase, file=sys.stderr)
            print()
            msg = '\n\nПробуете еще раз? (Нажмите Enter либо "n" для выхода)\n '
❾           try_again = input(msg)
            if try_again.lower() == "n":
                running = False
                sys.exit()
            else:
                main()
```

```
⓾ if __name__ == '__main__':
      main()
```

Первым делом необходимо превратить переменную `ini_name` в непрерывную цепочку символов в нижнем регистре без пробелов ❽. Напомним, что для языка Python регистр имеет значение, поэтому переведите все символьные цепочки в нижний регистр везде, где они встречаются; благодаря этому сравнения будут работать по назначению. Python также распознает пробелы как символы, поэтому, прежде чем делать какие-либо подсчеты букв, необходимо удалить их, а также дефисы в дефисных именах. Объявляя эту новую переменную `name`, вы сохраняете начальное имя на случай, если пользователь захочет начать все сначала. В функции `process_choice()` будет изменено только переменная `name`.

Далее получите длину имени ❾ для использования в качестве ограничения в цикле `while`. Она позволит вам узнать, когда анаграммное словосочетание потребило все буквы имени и наступило время закончить цикл. Для того чтобы начальное имя использовалось полностью, сделайте это вне цикла `while`. Затем задайте переменную, которая будет хранить анаграммное словосочетание, и установите переменную `running` равной `True` для управления циклом `while`.

Теперь начинается большой цикл, который позволяет вам перебрать символы имени и построить анаграммное словосочетание ❿. Сначала подготовьте цепочку для хранения растущего словосочетания и удалите из нее пробелы ⓫. Пробелы будут считаться буквами и сбивать указанную циклическую инструкцию, когда длина словосочетания сравнится с переменной `limit`.

Далее выполните сравнение, и если длина словосочетания меньше предела `limit`, то в качестве прелюдии к взаимодействию с пользователем покажите текущую длину словосочетания ⓬.

Самое время задействовать другие функции. Вызовите функцию `find_anagrams()` ❻ и передайте ей имя и файл словаря для получения списка анаграмм в имени. Внизу выводимого на экран списка покажите пользователю текущее словосочетание. Для вывода на экран двух инструкций `print` в одной строке примените параметр `end` функции `print()`. Благодаря ему в окне интерпретатора IDLE можно применять шрифт красного цвета в словосочетании для его выделения из всей другой информации на экране.

Далее вызовите функцию `process_choice()` ❼, которая получает вариант выбранного пользователем слова, и добавьте его в растущее анаграммное словосочетание. Она также возвращает обновленную версию переменной `name`, благодаря чему программа может использовать ее в цикле `while` снова в случае, если словосочетание не завершено.

Если длина словосочетания равна переменной `limit` ⓭, то анаграмма имени завершена. Сообщите пользователю о завершении обработки и покажите словосочетание с помощью красного шрифта. Обратите внимание, что у вас нет условного выражения на случай, когда длина словосочетания больше предела в переменной `limit`. Это связано с тем, что функция `process_choice()` уже обрабатывает этот

результат (выбор большего числа букв, чем доступно, не пройдет критерий проверки).

Функция `main()` завершается вопросом к пользователю о его желании повторить попытку. Если он набирает n, то программа завершается; если он нажимает <Enter>, то функция `main()` вызывается снова ❾. Как было сказано ранее, пользователь может изменить начальное имя единственным способом — выйти из программы и ее перезапустить.

Вне функции `main()` завершите стандартными двумя строками кода с вызовом функции `main()`, когда программа не импортирована в виде модуля ❿.

Выполнение демонстрационного сеанса

В этот раздел я включил пример интерактивного сеанса, используя программу phrase_anagrams.py и имя Bill Bo. Полужирный шрифт указывает на ввод данных пользователем, полужирный курсив — на то, где на экране используется красный шрифт.

```
Введите имя: Bill Bo
Длина анаграммного словосочетания = 0
bib
bill
blob
bob
boil
boll
i
ill
lib
lilo
lo
lob
oi
oil

Остальные буквы = billbo
Число остальных букв = 6
Число остальных анаграмм (реальных слов) = 14
Текущее анаграммное словосочетание =

Выберите либо введите Enter, чтобы начать сначала, либо # для выхода: ill
Длина анаграммного словосочетания = 3
bob

Остальные буквы = bbo
Число остальных букв = 3
Число остальных анаграмм (реальных слов) = 1
Текущее анаграммное словосочетание = ill
```

```
Выберите либо введите Enter, чтобы начать сначала, либо # для выхода: Bob

***** ЗАКОНЧЕНО!!! *****

Анаграмма имени = ill Bob

Попробуете еще раз? (Нажмите Enter либо "n" для выхода)
```

Число найденных анаграмм зависит от используемого файла словаря. Если у вас возникли трудности с построением анаграммных словосочетаний, то попробуйте применить более крупный словарь.

Проект 6: поиски Волдеморта, или галльский гамбит

Вы когда-нибудь задумывались о том, как Том Реддл придумал анаграмму "Я — Лорд Волдеморт"? Приложил ли он перо к пергаменту или же просто взмахнул палочкой? А может, ему помогло волшебство языка Python?

Давайте на мгновение представим, что вы — профессор компьютерной школы чародейства и волшебства в Хогвартсе и Том Реддл, школьный префект и образцовый ученик, пришел к вам за помощью. Используя волшебство вашей программы phrase_anagrams.py из предыдущего раздела, он мог бы, к своему огромному удовольствию, уже в самом первом списке анаграмм отыскать фразу "Я — Господь". Но остальные буквы, *tmvoordle*, дают только тривиальные слова, такие как dolt, drool, looter и lover (болван, слюни, мародер и любовник). И это Реддла не обрадовало бы.

Если мы оглянемся назад, то нам покажется, что проблема выглядит очевидной: слово Voldemort — французского происхождения и не будет найдено ни в одном файле словаря английского языка. Vol de la mort на французском языке означает "полет смерти", а английское Voldemort в свободном толковании будет означать "смертельный полет". Но Реддл — стопроцентный англичанин, и до сих пор вы работали с английским языком. Без реконструкции, причин внезапно переключиться с родного английского словаря на французский не больше, чем на голландский, немецкий, итальянский или испанский.

Можно попробовать случайно перетасовать оставшиеся буквы и посмотреть, что выпадет. К сожалению, число возможных комбинаций является факториалом числа букв, деленным на факториал числа повторов (буква *о* появляется дважды): $9!/2! = 181\,440$. Если бы вам пришлось прокручивать все эти перестановки, тратя на просмотр каждой из них всего одну секунду, то на то, чтобы закончить просмотр списка, вам потребовалось бы более двух дней! И если бы вы попросили об этом Тома Реддла, то он, вероятно, сделал бы из вас крестраж![1]

---

[1] В книгах о Гарри Поттере крестраж — это предмет, в котором заключена часть души темного мага. — *Прим. перев.*

На этом этапе я хотел бы наперед разведать два логических пути. Один я называю "галльским гамбитом", а другой — "британской грубой силой". Здесь мы рассмотрим первый, а второй — в следующем разделе.

> **ПРИМЕЧАНИЕ**
>
> Марволо — это явно сфабрикованное слово, используемое для того, чтобы анаграмма Волдеморта сработала. Джоан Роулинг могла бы получить дополнительные обороты, используя Thomas вместо Tom либо отбросив такие части, как "Lord" или "I am". Такие трюки используются, когда книга переводится на неанглийские языки. В некоторых языках может потребоваться изменить одно или оба имени. По-французски анаграмма звучит как "I am Voldemort" (я — Волдеморт). По-норвежски — Voldemort the Great (Волдеморт Великий). По-голландски — "My name is Voldemort" (Меня зовут Волдеморт). В других языках, таких как китайский, использовать анаграмму вообще невозможно!

Том Реддл был одержим победой над смертью, и если вы отправитесь на поиски смерти в *tmvoordle*, то найдете и старофранцузское слово *morte* (как в знаменитой книге "Le Morte d'Arthur" сэра Томаса Мэлори — "Смерть Артура"), и современное французское mort. После удаления букв *mort* остается *vodle*, пять букв с очень управляемым числом перестановок. На самом деле, можно легко отыскать *volde* прямо в окне интерпретатора:

```
❶ >>> from itertools import permutations
  >>> name = 'vodle'
❷ >>> perms = [''.join(i) for i in permutations(name)]
❸ >>> print(len(perms))
  120
❹ >>> print(perms)
  ['vodle', 'vodel', 'volde', 'voled', 'voedl', 'voeld', 'vdole', 'vdoel',
  'vdloe', 'vdleo', 'vdeol', 'vdelo', 'vlode', 'vloed', 'vldoe', 'vldeo',
  'vleod', 'vledo', 'veodl', 'veold', 'vedol', 'vedlo', 'velod', 'veldo',
  'ovdle', 'ovdel', 'ovlde', 'ovled', 'ovedl', 'oveld', 'odvle', 'odvel',
  'odlve', 'odlev', 'odevl', 'odelv', 'olvde', 'olved', 'oldve', 'oldev',
  'olevd', 'oledv', 'oevdl', 'oevld', 'oedvl', 'oedlv', 'oelvd', 'oeldv',
  'dvole', 'dvoel', 'dvloe', 'dvleo', 'dveol', 'dvelo', 'dovle', 'dovel',
  'dolve', 'dolev', 'doevl', 'doelv', 'dlvoe', 'dlveo', 'dlove', 'dloev',
  'dlevo', 'dleov', 'devol', 'devlo', 'deovl', 'deolv', 'delvo', 'delov',
  'lvode', 'lvoed', 'lvdoe', 'lvdeo', 'lveod', 'lvedo', 'lovde', 'loved',
  'lodve', 'lodev', 'loevd', 'loedv', 'ldvoe', 'ldveo', 'ldove', 'ldoev',
  'ldevo', 'ldeov', 'levod', 'levdo', 'leovd', 'leodv', 'ledvo', 'ledov',
  'evodl', 'evold', 'evdol', 'evdlo', 'evlod', 'evldo', 'eovdl', 'eovld',
  'eodvl', 'eodlv', 'eolvd', 'eoldv', 'edvol', 'edvlo', 'edovl', 'edolv',
  'edlvo', 'edlov', 'elvod', 'elvdo', 'elovd', 'elodv', 'eldvo', 'eldov']
  >>>
❺ >>> print(*perms, sep='\n')
  vodle
  vodel
  volde
  voled
  voedl
--обрезано--
```

Начните с импорта итератора перестановок `permutations` из модуля `itertools` ❶. Модуль `itertools` — это группа функций в стандартной библиотеке Python, которые создают итераторы с целью эффективной циклической обработки. Обычно, рассуждая о перестановках, имеют в виду числа, но версия `itertools` работает с итерируемыми типами, которые включают и буквы.

После ввода имени или, в данном случае, остальных букв имени примените операцию включения в список с целью создания списка перестановок имени ❷. Объедините каждый элемент в перестановке методом `join`, так чтобы каждый элемент в окончательном списке был уникальной перестановкой букв *vodle*. Использование указанного метода дает новое имя как один элемент, `'vodle'` вместо трудночитаемого кортежа из односимвольных элементов ('v', 'o', 'd', 'l', 'e').

В качестве проверки получите длину перестановок; благодаря этому вы можете подтвердить, что речь действительно идет о факториале 5 ❸. В конце концов независимо от того, как вы его напечатаете ❶❺, *volde* легко найти.

Проект 7: поиски Волдеморта, или британская грубая сила

Теперь давайте предположим, что Том Реддл плохо разбирается в анаграммах (либо французском языке). Он не отличает mort от morte, и вы снова тысячи и тысячи раз тасуете оставшиеся девять букв в поисках комбинации букв, которая ему понравилась бы.

С другой стороны, в программном отношении это задача интереснее, чем интерактивное решение, которое вы только что увидели. Нужно просто сократить все перестановки, используя некую форму фильтрации.

Цель

Уменьшить число анаграмм цепочки tmvoordle до управляемого числа, которое попрежнему будет содержать Voldemort.

Стратегия

Во 2-м издании Оксфордского словаря английского языка в настоящее время используется 171 476 английских слов, что меньше, чем общее число перестановок в tmvoordle! Независимо от языка можно предположить, что большинство генерируемых функцией `permutations()` анаграмм являются бессмыслицей.

С помощью *криптографии*, науки о кодах и шифрах, можно без вреда устранить многие бесполезные, непроизносимые комбинации, такие как *ldtmvroeo*, и вам даже не придется проверять их визуально. Криптографы уже давно изучают языки и составляют статистику по повторяющимся шаблонам слов и букв. Для имплементации этого проекта мы могли бы применить огромное число крипто-аналитических приемов, но давайте сосредоточимся на трех: проецировании букв в согласные и гласные, частоте триграмм и частоте диграмм.

Фильтрация с помощью проекции в гласные-согласные

Проецирование букв в гласные и согласные (consonant-vowel map, c-v map) в зависимости от обстоятельств просто заменяет буквы в слове на *c* (consonant — согласная) или *v* (vowel — гласная). Слово Riddle, например, становится *cvcccv*. Можно написать программу, которая сканирует файл словаря и создает проекцию в гласные-согласные для каждого слова. По умолчанию невозможные комбинации, такие как *ccccccvvv*, будут исключены. Можно также исключить членство, удалив слова с проекциями в гласные-согласные, которые в принципе *возможны*, но имеют низкую частоту встречаемости.

Проецирование букв в согласные и гласные имеет довольно всеохватывающий характер, но это хорошо. На данный момент вариант для слова Riddle состоит в том, чтобы составить новое имя собственное, а имена собственные не обязательно должны быть как раз теми словами, которые встречаются в словаре. И поэтому вы не хотите *слишком* ограничивать себя в самом начале процесса.

Фильтрация с помощью триграмм

Поскольку начальный фильтр требует относительно широкой апертуры (проходимого отверстия), вам придется снова выполнять фильтрацию на более низком уровне для того, чтобы без вреда удалить из перестановок еще больше анаграмм. *Триграммы* — это триплеты, в состав которых входят три идущие подряд буквы. Неудивительно, что самой распространенной триграммой в английском языке является слово *the*, за которым следуют *and* и *ing*. На другом конце шкалы находятся такие триграммы, как *zvq*.

Статистику по частоте появления триграмм можно найти в Интернете, в частности по адресу **http://norvig.com/ngrams/count_3l.txt**. Для любой группы букв, например *tmvoordle*, можно сгенерировать и применить список наименее распространенных триграмм, еще больше сократив число перестановок. В этом проекте можно использовать файл least-likely_trigrams.txt, который можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**, с наименее вероятными триграммами. Этот текстовый файл содержит триграммы в *tmvoordle*, которые встречаются в нижних 10% триграмм английского языка, на основе частоты встречаемости.

Фильтрация с помощью диграмм

Диграммы (также именуемые *биграммами*) — это пары букв. Широко встречающиеся диграммы английского языка включают *an*, *st* и *er*. С другой стороны, редко встречаются такие пары, как *kg*, *vl* или *oq*. Статистику по частоте появления диграмм можно найти на таких веб-сайтах, как **https://www.math.cornell.edu/~mec/2003-2004/cryptography/subs/digraphs.html** и **http://practicalcryptography.com/**.

Таблица 3.1 была построена из коллекции букв слова *tmvoordle* и файла словаря английского языка на 60 тыс. слов. Буквы в левой части диаграммы являются начальными буквами диграмм; расположенные вверху представляют конечную букву. Например, для отыскания *vo* начните с *v* слева и читайте поперек до столбца под *o*. В случае диграмм в слове *tmvoordle* сочетание *vo* встречается только в 0,8% случаев.

Таблица 3.1. *Относительная частота диграмм из букв слова tmvoordle в словаре на 60 000 слов (черные квадраты указывают на отсутствие вхождений)*

	d	e	l	m	o	r	t	v
d		3,5%	0,5%	0,1%	1,7%	0,5%	0.0%	0,1%
e	6,6%		2,3%	1,4%	0,7%	8,9%	2,0%	0,6%
l	0,4%	4,4%			4,2%	0.0%	0,4%	0,1%
m	0.0%	2,2%	0.0%	0,1%	2,8%	0.0%	0.0%	0.0%
o	1,5%	0,5%	3,7%	3,2%	5,3%	7,1%	2,4%	1,4%
r	0,9%	6,0%	0,4%	0,7%	5,7%		1,3%	0,3%
t	0.0%	6,2%	0,6%	0,1%	3,6%	2,3%		0.0%
v	0.0%	2,5%	0.0%	0.0%	0,8%	0.0%	0.0%	

Если исходить из того, что вы ищете комбинации букв, которые являются "англоподобными", то подобного рода частотные проекции можно использовать с целью исключить пары букв, которые вряд ли встретятся. Думайте об этом как о "диграммном решете", которое пропускает только незатененные квадраты.

Для подстраховки исключите диграммы, которые встречаются просто менее 0,1% случаев. Я затенил их черным цветом. Обратите внимание, что если вы будете резать слишком близко к кости, то можно очень легко устранить нужное сочетание букв *vo* в слове Voldemort!

Свой фильтр можно сделать еще более избирательным, пометив диграммы, которые вряд ли встретятся в начале слова. Например, хотя нет ничего необычного в том, что диграмма *lm* появится внутри слова (как в словах almanac и balmy, например), вам понадобится немало удачи для того, чтобы отыскать слово, которое будет начинаться с *lm*. Для того чтобы отыскать эти диграммы, вовсе не нужна криптография; просто попробуйте произнести их! В табл. 3.2 некоторые отправные варианты для них затенены серым цветом.

Таблица 3.2. *Обновление табл. 3.1, где серые затененные квадраты указывают на то, что диграммы вряд ли появятся в начале слова*

	d	e	l	m	o	r	t	v
d		3,5%	0,5%	0,1%	1,7%	0,5%	0.0%	0,1%
e	6,6%		2,3%	1,4%	0,7%	8,9%	2,0%	0,6%
l	0,4%	4,4%		0,1%	4,2%	0.0%	0,4%	0,1%
m	0.0%	2,2%	0.0%	0,1%	2,8%	0.0%	0.0%	0.0%
o	1,5%	0,5%	3,7%	3,2%	5,3%	7,1%	2,4%	1,4%
r	0,9%	6,0%	0,4%	0,7%	5,7%		1,3%	0,3%
t	0.0%	6,2%	0,6%	0,1%	3,6%	2,3%		0.0%
v	0.0%	2,5%	0.0%	0.0%	0,8%	0.0%	0.0%	

Теперь у вас есть три фильтра, которые можно применить к 181 440 перестановкам слова *tmvoordle*: проекции в гласные-согласные, триграммы и диграммы. В качестве последнего фильтра вы должны предоставить пользователю возможность просмотра только тех анаграмм, которые начинаются с заданной буквы. Это позволит пользователю разделить оставшиеся анаграммы на более управляемые "куски" либо сосредоточиться на тех анаграммах, которые звучат более пугающе, таких, которые начинаются с *v*!

Код с использованием британской грубой силы

Приведенный ниже программный код генерирует перестановки слова tmvoordle и пропускает их через только что описанные фильтры. Затем он дает пользователю возможность просмотреть либо все перестановки, либо только те из них, которые начинаются с заданной буквы.

Все программы, которые вам понадобятся, можно скачать с **https://www.nostarch.com/impracticalpython/**. Программный код этого раздела представлен одним скриптом с именем voldemort_british.py. Вам также понадобится программа load_dictionary.py в той же папке вместе с тем же файлом словаря, который вы использовали в предыдущих проектах этой главы. Наконец, вам потребуется новый файл с именем least-likely_trigrams.txt — текстовый файл триграмм с низкой частотой встречаемости в английском языке. Скачайте все эти файлы в одну папку.

Определение функции *main()*

Листинг 3.5 импортирует модули, необходимые для программы voldemort_british.py, и определяет свою функцию `main()`. В программе phrase_anagrams.py вы определили функцию `main()` в конце кода. Здесь мы поместили ее в начало. Преимущество заключается в том, что вы с самого начала видите, что конкретно функция делает — как она выполняет программу. Недостатком является то, что вы еще не знаете, что делают все вспомогательные функции.

Листинг 3.5. Импортирует модули и определяет функцию `main()`. Файл voldemort_british.py, часть 1

```
❶ import sys
  from itertools import permutations
  from collections import Counter
  import load_dictionary

❷ def main():
      """Загрузить файлы, выполнить фильтры, показать пользователю
         анаграммы по 1-й букве."""
  ❸   name = 'tmvoordle'
      name = name.lower()

  ❹   word_list_ini = load_dictionary.load('2of4brif.txt')
      trigrams_filtered = load_dictionary.load('least-likely_trigrams.txt')
```

```
❺ word_list = prep_words(name, word_list_ini)
  filtered_cv_map = cv_map_words(word_list)
  filter_1 = cv_map_filter(name, filtered_cv_map)
  filter_2 = trigram_filter(filter_1, trigrams_filtered)
  filter_3 = letter_pair_filter(filter_2)
  view_by_letter(name, filter_3)
```

Начните с импорта модулей, которые вы использовали в предыдущих проектах ❶. Теперь определите функцию `main()` ❷. Переменная `name` представляет собой цепочку из остальных букв слова *tmvoordle* ❸. Переведите ее в нижний регистр, для того чтобы защититься от ошибки пользователя при вводе. Затем с помощью модуля `load_dictionary` загрузите файл словаря и файл триграмм в виде списков ❹. Имя файла вашего словаря может отличаться от показанного.

Наконец, вызовите все функции по порядку ❺. Вскоре я опишу каждую из этих функций, но, в сущности, вам нужно подготовить список слов, подготовить проекции в гласные-согласные, применить три фильтра и дать пользователю увидеть все анаграммы целиком либо просмотреть их подмножество на основе первой буквы анаграммы.

Подготовка списка слов

Листинг 3.6 подготавливает список слов, включая только те слова, которые содержат столько же букв, сколько и в переменной `name` (в данном случае девять). В целях поддержки единообразия слов вы также должны обеспечить, чтобы все слова находились в нижнем регистре.

Листинг 3.6. Создание списков слов, длина которых равна переменной `name`. Файл voldemort_british.py, часть 2

```
❶ def prep_words(name, word_list_ini):
      """Подготовить список слов для отыскания анаграмм."""
   ❷ print("длина первоначального списка word_list = {}".
           format(len(word_list_ini)))
      len_name = len(name)
   ❸ word_list = [word.lower() for word in word_list_ini
                  if len(word) == len_name]
   ❹ print("длина нового списка word_list = {}".format(len(word_list)))
   ❺ return word_list
```

Определите функцию `prep_words()`, которая в качестве аргументов принимает символьную цепочку с именем и список слов словаря ❶. Я предлагаю вам печатать длины ваших списков слов до и после того, как они прошли через фильтр; благодаря этому вы сможете отслеживать степень воздействия фильтров. Поэтому напечатайте длину словаря ❷. Задайте переменную для хранения длины имени, а затем

примените операцию включения в список для создания нового списка путем перебора слов в word_list_ini, сохраняя те из них, длина которых совпадает с числом букв в имени, и их конвертации в нижний регистр ❸. Затем напечатайте длину этого нового списка слов ❹ и, наконец, верните этот новый список для использования в следующей функции ❺.

Генерирование проекции в гласные-согласные

Вам нужно конвертировать подготовленный список слов в проекцию в гласные-согласные (c-v map). Напомню, что вы больше не заинтересованы в реальных словах из словаря; они были рассмотрены и отклонены. Ваша цель состоит в том, чтобы тасовать оставшиеся буквы до тех пор, пока они не образуют нечто напоминающее имя собственное.

Листинг 3.7 определяет функцию, которая для каждого слова в word_list создает проекции в гласные-согласные. Программа voldemort_british.py будет использовать проекцию в гласные-согласные для вынесения решения о том, является ли перетасованная комбинация букв разумной, на основе шаблонов следования согласных и гласных в английском языке.

Листинг 3.7. Создает проекции в гласные-согласные из слов в word_list. Файл voldemort_british.py, часть 3

```
❶ def cv_map_words(word_list):
      """Спроецировать буквы слов в согласные и гласные."""
   ❷ vowels = 'aeiouy'
   ❸ cv_mapped_words = []
   ❹ for word in word_list:
         temp = ''
         for letter in word:
             if letter in vowels:
                 temp += 'v'
             else:
                 temp += 'c'
         cv_mapped_words.append(temp)

      # определить число УНИКАЛЬНЫХ шаблонов согласная-гласная
   ❻ total = len(set(cv_mapped_words))
      # целевая доля устраняемых
   ❼ target = 0.05
      # получить число элементов в целевой доле
   ❽ n = int(total * target)
   ❾ count_pruned = Counter(cv_mapped_words).most_common(total - n)
   ❿ filtered_cv_map = set()
      for pattern, count in count_pruned:
          filtered_cv_map.add(pattern)
```

```
    print("длина множества filtered_cv_map = {}".
          format(len(filtered_cv_map)))
⑩   return filtered_cv_map
```

Определите функцию `cv_map_words()`, которая в качестве аргумента принимает готовый список слов ❶. Поскольку согласные и гласные образуют двоичную систему, вы можете определить гласные с помощью символьной цепочки ❷. Создайте пустой список для хранения проекций ❸. Затем в цикле переберите слова из списка и буквы в каждом слове, конвертируя буквы в *c* или *v* ❹. Используйте переменную `temp` для накопления проекции; затем добавьте ее в список. Обратите внимание, что `temp` инициализируется повторно всякий раз, когда цикл повторяется.

Вы хотите знать частоту появления заданного шаблона проекции (например, *cvcv*) с целью устранения тех из них, которые имеют низкую вероятность появления. Перед вычислением частоты вам нужно свести свой список к уникальным проекциям — как и сейчас, цепочка *cvcv* может повторяться очень много раз. Поэтому преобразуйте список `cv_mapped_words` в множество, удалив дубликаты, и получите его длину ❺. Теперь вы можете определить целевой процент устраняемых элементов, используя дробные значения ❻. Начните с низкого числа, такого как 0,05 — эквивалента 5%, — благодаря такому значению вы с меньшей вероятностью исключите анаграммы, которые могут образовывать полезные имена собственные. Умножьте это целевое значение на суммарную длину множества `cv_mapped_words` и назначьте результат переменной `n` ❼. Обязательно сконвертируйте `n` в целое число; оно не может быть вещественным, т. к. оно будет представлять значение количества.

Модуль `Count` имеет удобный метод `most_common()`, который возвращает наиболее распространенные элементы в списке на основе значения *количества*, которое вы предоставите; в данном случае это значение будет длиной списка проекций в гласные-согласные, `total`, минус `n`. Передаваемое вами в функцию `most_common()` значение должно быть целым числом. Если передать в функцию `most_common()` длину списка, то она вернет все элементы списка. Если вычесть количество наименее вероятных 5%, то вы практически устраните эти проекции из списка ❽.

Напомню, что объект `Count` возвращает словарь, но вам нужны лишь окончательные проекции в гласные-согласные, а не ассоциированные с ними частотности (частотные количества). Поэтому инициализируйте пустое множество с именем `filtered_cv_map` ❾ и в цикле переберите каждую пару "ключ — значение" в `count_pruned()`, добавляя в новое множество только ключ. Напечатав длину этого множества, вы увидите влияние фильтра. Затем завершите действия, вернув отфильтрованную проекцию для использования в следующей функции ❿.

Определение фильтра проекций в гласные-согласные

В листинге 3.8 применяется фильтр проекций в гласные-согласные: на основе каждой перестановки букв в переменной `name` генерируются анаграммы, а затем программа конвертирует их в проекции в гласные-согласные и сравнивает эти анаграммы с отфильтрованными проекциями, построенными с помощью функции `cv_map_words()`. Если проекция анаграммы находится в `filtered_cv_map`, то программа сохраняет анаграмму для следующего фильтра.

Листинг 3.8. Определяет функцию `cv_map_filter()`. Файл voldemort_british.py, часть 4

```
❶ def cv_map_filter(name, filtered_cv_map):
      """Удалить перестановки слов на основе маловероятных
         комбинаций гласные-согласные."""
   ❷ perms = {''.join(i) for i in permutations(name)}
      print("длина первоначального множества перестановок = {}".
            format(len(perms)))
      vowels = 'aeiouy'
   ❸ filter_1 = set()
   ❹ for candidate in perms:
          temp = ''
          for letter in candidate:
              if letter in vowels:
                  temp += 'v'
              else:
                  temp += 'c'
      ❺ if temp in filtered_cv_map:
             filter_1.add(candidate)
      print("# вариантов после фильтра filter_1 = {}".
            format(len(filter_1)))
   ❻ return filter_1
```

Определите функцию `cv_map_filter()`, принимающую два аргумента: имя, за которым следует набор проекций в гласные-согласные, возвращаемых из `cv_map_words()` ❺. Примените операцию включения в множество и итератор `permutations` для генерирования множества перестановок ❷. Этот процесс описан в *разд. "Проект 6: поиски Волеморта, или галльский гамбит" ранее в этой главе*. Используйте здесь множество, для того чтобы позже иметь возможность применять операции над множествами, например взятие разницы между двумя отфильтрованными множествами. Указанная структура данных также удаляет дубликаты, т. к. перестановки рассматривают каждую букву *о* как отдельный элемент и вместо 9!/2! возвращают 9!. Обратите внимание, что итератор `permutations` рассматривает слова *tmvoordle* и *tmvoordle* как разные символьные цепочки.

Теперь инициализируйте пустое множество для хранения содержимого первого фильтра ❸ и начните перебирать перестановки в цикле ❶. Используйте термин *"кандидат"*, т. к. большинство из них являются не словами, а просто символьными цепочками из случайных букв. Пройдите буквы каждого кандидата в цикле и спроецируйте их в *c* или *v*, как это было сделано в функции `cv_map_words()`. Проверьте каждую проекцию `temp` на членство в `filtered_cv_map`. Это одна из причин использовать множества: проверка вхождения в множества выполняется молниеносно. Если кандидат соответствует условию, то добавьте его в `filter_1` ❺. Завершите обработку, вернув свое новое множество анаграмм ❻.

Определение триграммного фильтра

Листинг 3.9 определяет триграммный фильтр, который удаляет перестановки с маловероятными трехбуквенными триплетами. Он использует текстовый файл, полученный из различных криптографических веб-сайтов, который был адаптирован к буквам в слове *tmvoordle*. Эта функция будет возвращать только те перестановки, которые содержат одну из этих триграмм; функция `main()` передаст новое множество следующей фильтровочной функции.

Листинг 3.9. Определяет функцию `trigram_filter()`. Файл voldemort_british.py, часть 5

```
❶ def trigram_filter(filter_1, trigrams_filtered):
      """Удалить маловероятные триграммы из перестановок."""
   ❷ filtered = set()
   ❸ for candidate in filter_1:
      ❹  for triplet in trigrams_filtered:
          triplet = triplet.lower()
          if triplet in candidate:
              filtered.add(candidate)
   ❺ filter_2 = filter_1 - filtered
     print("# вариантов после фильтра filter_2 = {}".
           format(len(filter_2)))
   ❻ return filter_2
```

Параметры для триграммного фильтра включают выход из фильтра проекций в гласные-согласные и внешний список маловероятных триграмм, `trigrams_filtered` ❶.

Инициализируйте пустое множество для хранения перестановок, содержащих одну из запрещенных триграмм ❷. Затем начните еще один цикл `for`, который просматривает кандидатов, переживших предыдущий фильтр ❸. Вложенный цикл `for` просматривает каждый триплет в списке триграмм ❹. Если триплет находится в кандидате, то он добавляется в фильтр.

Теперь можно применить операции над множествами с целью вычесть новый фильтр из `filter_1` ❺, а затем вернуть разницу для использования в следующем фильтре ❸.

Определение диграммного фильтра

В листинге 3.10 определен диграммный фильтр, который удаляет маловероятные буквенный пары. Некоторые будут активировать фильтр, если они появляются в любом месте в пределах перестановки; другие будут делать это только в том случае, если они появляются в начале перестановки. Запрещенные диграммы основаны на затененных ячейках в табл. 3.2. Указанная функция возвращает результаты этого фильтра для использования в конечной фильтровочной функции.

Листинг 3.10. Определяет функцию `letter_pair_filter()`. Файл voldemort_british.py, часть 6

```
❶ def letter_pair_filter(filter_2):
      """ Удалить маловероятные буквенные пары из перестановок."""
   ❷ filtered = set()
   ❸ rejects = ['dt', 'lr', 'md', 'ml', 'mr', 'mt', 'mv',
                'td', 'tv', 'vd', 'vl', 'vm', 'vr', 'vt']
   ❹ first_pair_rejects = ['ld', 'lm', 'lt', 'lv', 'rd',
                           'rl', 'rm', 'rt', 'rv', 'tl', 'tm']
   ❺ for candidate in filter_2:
      ❻  for r in rejects:
             if r in candidate:
                 filtered.add(candidate)
      ❼  for fp in first_pair_rejects:
          if candidate.startswith(fp):
              filtered.add(candidate)
   ❽ filter_3 = filter_2 - filtered
     print("# вариантов после фильтра filter_3 = {}".
           format(len(filter_3)))
   ❾ if 'voldemort' in filter_3:
         print("Волдеморт найден!", file=sys.stderr)
   ❿ return filter_3
```

Указанный фильтр в качестве аргумента принимает результаты предыдущего фильтра ❶. Пустое множество инициализируется для хранения любых отброшенных перестановок ❷. Затем два списка отклоненных пар назначаются переменным `rejects` ❸ и `first_pair_rejects` ❹. Оба списка были введены вручную. Первый представляет ячейки табл. 3.2, затененные черным цветом; второй ссылается на ячейки, затененные серым цветом. Любая перестановка, содержащая член первого списка — в любом месте, — будет отброшена; перестановки, начинающиеся с члена второго списка, не будут допущены. Эти списки можно изменить, добавив либо удалив диграммы, тем самым изменив поведение фильтра.

Начните перебирать перестановки — продолжайте называть их "кандидатами", поскольку они не обязательно являются словами ❺. Вложенный цикл `for` проходит пары в `rejects`, определяет, находятся ли они в `candidate`, и добавляет их в множество `filtered` ❻. Второй вложенный цикл `for` повторяет этот процесс для `first_pair_rejects` ❼. Вычтите `filtered` из множества `filter_2`, возвращенного предыдущей функцией ❽.

Ради удовольствия и в целях обеспечения того, чтобы вы не переусердствовали с фильтрованием, проверьте наличие слова *voldemort* в `filter_3` ❾ и распечатайте объявление с выделенным найденным словом, используя привлекательный красный шрифт для пользователей интерпретатора IDLE. Затем завершите, вернув окончательное множество `filtered` ❿.

Предоставление возможности выбирать начальную букву

Вы не можете сказать заранее, будет ли ваша фильтрация успешной или нет. Вы по-прежнему можете получить тысячи перестановок. Возможность наблюдать только подмножество результатов не уменьшит совокупное их число, но зато психологически облегчит их восприятие. Листинг 3.11 добавляет в программу voldemort_british.py возможность просмотра списка анаграмм, которые начинаются с определенной входной буквы.

Листинг 3.11. Определяет функцию `view_by_letter()`. Файл voldemort_british.py, часть 7

```
❶ def view_by_letter(name, filter_3):
      """Отфильтровать анаграммы, начинающиеся с входной буквы."""
   ❷ print("Остальные буквы = {}".format(name))
   ❸ first = input("выберите стратовую букву либо нажмите Enter,
                                                    чтобы увидеть все: ")
   ❹ subset = []
   ❺ for candidate in filter_3:
         if candidate.startswith(first):
             subset.append(candidate)
   ❻ print(*sorted(subset), sep='\n')
   print("Число вариантов, начинающихся с {} = {}".
         format(first, len(subset)))
   msg = "Попробуете еще раз? (Нажмите Enter либо любую другую клавишу
                                                        для выхода):"
   ❼ try_again = input(msg)
     if try_again.lower() == '':
     ❽ view_by_letter(name, filter_3)
     else:
     ❾ sys.exit()
```

Определите функцию `view_by_letter()`, которая в качестве аргументов принимает переменную `name` и фильтр `filter_3` ❶. Вам нужно имя, поэтому вы можете показать пользователю имеющиеся варианты букв, по которым производится фильтрация ❺. Получите от пользователя ответ относительно того, хочет ли он видеть все остальные перестановки целиком либо только те, которые начинаются с определенной буквы ❼. Затем создайте пустой список, который будет хранить подмножество второго варианта ❹.

Цикл `for` с условием проверяет, что кандидат начинается с выбранной буквы, и добавляет те буквы, которые передаются в подмножество `subset` ❺. Этот список печатается с помощью унарного оператора "звездочка" ❻. Затем программа спрашивает пользователя, хочет ли он повторить попытку либо выйти из программы ❼. Если он нажимает клавишу <Enter>, то `view_by_letter()` вызывается рекурсивно и

запускается снова с самого начала ❽. В противном случае программа завершает работу ❾. Обратите внимание, что в Python по умолчанию глубина рекурсии ограничена 1000 вызовами, и данное ограничение в этом проекте мы проигнорируем.

Выполнение функции *main()*

Вернувшись в глобальное пространство, листинг 3.12 завершает код, вызывая функцию `main()`, если пользователь запускает программу в автономном режиме в отличие от импорта в другую программу.

Листинг 3.12. Вызов функции `main()`

```
if __name__ == '__main__':
    main()
```

Пример результата, полученного из завершенной программы, показан ниже. После того как программа применила третий фильтр, осталось 248 перестановок, из которых очень управляемые 73 начинаются с *v*. Для краткости распечатка перестановок опущена. Как видно в полученном результате, слово *voldemort* после фильтрации остается.

```
длина первоначального списка word_list = 60388
длина нового списка word_list = 8687
длина множества filtered_cv_map = 234
длина первоначального множества перестановок = 181440
# вариантов после фильтра filter_1 = 123120
# вариантов после фильтра filter_2 = 674
# вариантов после фильтра filter_3 = 248
Волдеморт найден!
Остальные буквы = tmvoordle
выберите стратовую букву либо нажмите Enter, чтобы увидеть все: v
```

Интересно, что еще одной сохранившейся перестановкой является *lovedmort* (любящий смерть). С учетом того, сколько людей Волдеморт погубил (или мог бы погубить), это прозвище, возможно, будет самым подходящим из всех.

Резюме

В этой главе вы сначала написали код, который отыскал анаграммы для заданного слова или имени. Затем вы его расширили с целью отыскания анаграмм именных словосочетаний, работая в интерактивном режиме с пользователем. Наконец, вы применили крипто-аналитические методы для выуживания Волдеморта почти из 200 тыс. возможных анаграмм. По пути вы применили полезный функционал из модулей `collections` и `itertools`.

Дальнейшее чтение

Веб-сайт игры Jumble находится по адресу **http://www.jumble.com/**.

Несколько представительных онлайновых генераторов анаграмм можно найти на веб-сайтах по следующим адресам:

- **http://wordsmith.org/anagram/**;
- **https://www.dcode.fr/anagram-generator**;
- **http://www.wordplays.com/anagrammer/**.

Другие анаграммные программы представлены в книге Аллена Дауни "Думай по-питоновски" (Downey A. Think Python. O'Reilly, 2015).

Книга Эла Свейгарта "Взламывание кодов с помощью Python" (Sweigart A. Cracking codes with Python. No Cramp Press, 2018) предоставляет еще несколько примеров исходного кода для вычисления шаблонов слов, таких как те, которые используются для фильтрации в программе voldemort_british.py.

Практический проект: поиск диграмм

В поисках частотной статистики можно прочесать криптографические веб-сайты либо вывести ее самим. Напишите программу Python, которая находит все диграммы в слове *tmvoordle*, а затем подсчитывает их частоту появления в файле словаря. Обязательно протестируйте свой программный код на таких словах, как *volvo*, с тем чтобы не упустить из виду повторяющиеся диграммы в одном и том же слове. Решение данного проекта можно найти в приложении к книге либо скачать программу count_digrams_practice.py с веб-сайта

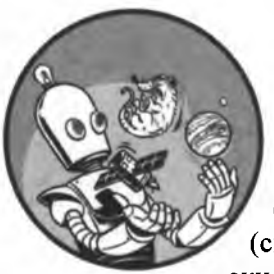
https://www.nostarch.com/impracticalpython/.

Сложный проект: автоматический генератор анаграмм

Взгляните на онлайновые генераторы анаграмм, на которые я только что сослался в *разд. "Дальнейшее чтение"*, и напишите программу на языке Python, которая имитирует один из них. Ваша программа должна автоматически генерировать фразовые анаграммы из входного имени и выводить на экран их подмножество (например, первые 500) для просмотра пользователем.

4

ДЕКОДИРОВАНИЕ ШИФРОВ ВРЕМЕН ГРАЖДАНСКОЙ ВОЙНЫ В США



Криптография — это наука о безопасном общении посредством использования *кодов* и *шифров*. Код заменяет целые слова другими словами; шифр беспорядочно перемешивает (скремблирует) или заменяет буквы в словах (поэтому технически азбука Морзе, или морзянка, — это в действительности шифр Морзе). Одной из целей криптографии является использование *ключа* для *шифровки* читаемого *открытого* текста в нечитаемый *шифротекст*, а затем *расшифровки* его обратно в открытый текст. Цель *криптоанализа* — расшифровать шифры и коды, не зная их ключа или алгоритма шифрования.

В этой главе мы рассмотрим два шифра, известных во времена Гражданской войны в США: маршрутный шифр, использовавшийся Севером, и зигзагообразный шифр (шифр жердевой изгороди), использовавшийся обеими сторонами. Мы также выясним, что сделало один из них таким успешным и как мы можем задействовать уроки, извлеченные из его применения, для того чтобы более качественно писать программы для неопытных пользователей и тех, кто не знаком с вашим программным кодом на Python.

Проект 8: маршрутный шифр

Во время Гражданской войны в Америке Союзники имели почти все преимущества перед Конфедератами, в том числе в области криптографии. У Союза были лучшие коды, лучшие шифры и лучше обученный персонал. Но, пожалуй, самое большое его преимущество заключалось в лидерстве и организации.

Главой военного телеграфного управления США был Энсон Стейджер (рис. 4.1). Как соучредитель телеграфной компании Western Union, Стейджер знал по опыту, что телеграфисты делают меньше ошибок при отправке целых слов, в отличие от цепочек случайных букв и цифр, общепринятых для большинства шифротекстов. Он также знал, что для того чтобы приказы были выполнены, военные депеши должны оставаться в секрете достаточно долгое время. Его безопасным решением была гибридная криптосистема, именуемая *маршрутным перестановочным шифром* (route transposition cipher — дословно шифр перестановки маршрута), т. е. комбинация переставленных реальных слов и кодовых слов, который стал одним из самых успешных военных шифров всех времен.

Рис. 4.1. Генерал Энсон Стейджер, телеграфный корпус США, 1865 г.

В отличие от подстановочных шифров, которые *заменяют* буквы в открытом тексте различными знаками или символами, перестановочные шифры *беспорядочно перемешивают* (скремблируют) расстановку букв или слов. На рис. 4.2 показан пример маршрутного перестановочного шифра. Сообщение записывается слева направо по ряду предопределенных столбцов и строк, важные слова открытого текста заменяются кодовыми словами, а последняя строка заполняется фиктивными словами-заполнителями. Читатель определяет порядок переупорядоченных слов, проходя вверх и вниз по этим столбцам, как показано на рисунке. Начальным словом является REST (ОТДЫХ), а дальнейший маршрут шифрования показан стрелками.

Для того чтобы полностью расшифровать это сообщение, вам нужно знать отправную точку и маршрут, используемый для прохождения по сообщению и создания окончательного шифротекста, и смысл кодовых слов.

В начале XX в. выдающийся военный криптоаналитик Уильям Фридман отзывался о маршрутном шифре Стейджера пренебрежительно. Он считал его слишком бесхитростным и находил крайне маловероятным, что конфедераты его так и не взломали. Но факт остается фактом: сотни тысяч маршрутных шифров, отправленных

Code Words

VILLAGE = Enemy	**ROANOKE** = Cavalry
GODWIN = Tennessee	**SNOW** = Rebels

Original Message in Encryption Matrix

Enemy	cavalry	heading	to
Tennessee	With	Rebels	gone
you	are	free	to
transport	your	supplies	south

Encryption Route + Code & Dummy Words

VILLAGE	**ROANOKE**	heading	to
GODWIN	With	**SNOW**	gone
you	are	free	to
transport	your	supplies	south
REST	**IS**	**JUST**	**FILLER**

Cyphertext

REST TRANSPORT YOU GODWIN VILLAGE
ROANOKE WITH ARE YOUR IS JUST SUPPLIES FREE
SNOW HEADING TO GONE TO SOUTH FILLER

Кодовые слова

ДЕРЕВНЯ = Вражеский	**РОАНОК** = Кавалерия
ГОДВИН = Теннесси	**СНОУ** = Мятежники

Исходное сообщение в матрице шифрования

Вражеская	кавалерия	направляется	в
Теннесси	Со	Мятежниками	ушли
Вы	свободно	можете	
транспортировать	свое	снаряжение	на юг

Маршрут шифрования + код и фиктивные слова

ДЕРЕВНЯ	**РОАНОК**	направляется	в
ГОДВИН	Со	**СНОУ**	ушли
вы		свободно	можете
транспортировать	ваше	снаряжение	на юг
ОТДЫХ	**ЭТО**	**ПРОСТО**	**ЗАПОЛНИТЕЛЬ**

Шифротекст

ОТДЫХ ТРАНСПОРТИРОВАТЬ ВЫ ГОДВИН ДЕРЕВНЯ
РОАНОК СО МОЖЕТЕ ВАШИ ПРОСТО СНАРЯЖЕНИЯ СВОБОДНО
СНОУ НАПРАВЛЯЕТСЯ К УШЛИ НА ЮГ ЗАПОЛНИТЕЛЬ

Рис. 4.2. Маршрутный шифр, в котором используются фактические кодовые слова союзников

во время войны, по-видимому, так и не были расшифрованы, и не от недостатка попыток. В одном из ранних примеров краудсорсинга конфедераты опубликовали закодированные сообщения в газетах, надеясь хоть на какую-то помощь в расшифровке, но эта попытка оказалась безрезультатной. Хотя историки предполагают, что данный шифр иногда удавалось взламывать, схема Стейджера преподает несколько важных уроков.

- **Проектирование с учетом человеческой ошибки.** Военные шифры должны быть простыми, т. к. в день могут отправляться сотни. Используемые в маршрутном шифре настоящие слова делали гораздо менее вероятным то, что он будет искажен телеграфистами. Стейджер знал своих клиентов и на них рассчитывал. Он осознавал ограниченности своей рабочей силы и соответствующе подстраивал свое изделие. Конфедераты, напротив, расшифровывали свои сложные сообщения с большим трудом, иногда бросая их и объезжая вокруг вражеских линий, для того чтобы поговорить лицом к лицу!
- **Инновация имеет преимущество перед изобретением.** Иногда не нужно изобретать что-то новое, просто нужно заново открыть что-то старое. Короткие шифры перестановки слов, пригодные для телеграфной передачи, были слишком слабы, для того чтобы использовать их самостоятельно, но в сочетании с кодовыми именами и дезориентирующими фиктивными словами они сбивали с толку конфедератов.
- **Обмен опытом.** Поскольку в телеграфном корпусе все использовали одну и ту же методологию, было легко опираться на существующие решения и делиться извлеченными уроками. Это позволило маршрутному шифру с течением времени эволюционировать за счет введения сленга и преднамеренных опечаток, а также растущего числа кодовых слов для топографических мест, людей и дат.

Практичный шифр Стейджера, возможно, не понравился более поздним "перфекционистам", но для того времени это схема была идеальной. Идеи, лежащие в его основе, являются вечными и легко переносятся в современные приложения.

Цель

В удостоенном наград романе Гарри Тертлдав 1992 года "Оружие Юга" (Guns of the South) путешественники во времени снабжают армии конфедератов современным оружием, меняя ход истории. Давайте представим, что вместо АК-47 вы вернулись в 1864 г. с ноутбуком, несколькими дополнительными батареями и языком Python, для того чтобы разработать алгоритм, который будет расшифровывать маршрутный шифр на основе вымышленной матрицы шифрования и пути. Вы напишете удобную для пользователя программу в духе Стейджера, которая уменьшит человеческую ошибку.

Стратегия

Когда дело касается решения шифров, все намного проще, если вы знаете, с каким типом шифра вы имеете дело. В этом случае вы знаете, что этот шифр является перестановочным, потому что он состоит из реальных слов, которые перемешаны между собой. Вы также знаете, что есть кодовые слова и присутствуют пустые слова. Ваша задача — выяснить способы расшифровки *перестановочной* части мар-

шрутного шифра, а кто-то другой пусть побеспокоится о кодовых словах, пока вы сходите за заслуженным мятным джулепом.

Создание управляющего сообщения

Для того чтобы понять, как это сделать, создайте собственное сообщение и маршрутный шифр. Назовем его управляющим сообщением:

- число столбцов — 4;
- число строк — 5;
- отправная позиция — внизу слева;
- маршрут — чередование столбцов вверх и вниз;
- открытый текст — 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19;
- шифротекст — 16 12 8 4 0 1 5 9 13 17 18 14 10 6 2 3 7 11 15 19;
- ключ — –1 2 –3 4.

Использование числовой прогрессии для открытого текста позволяет в любом месте в сообщении мгновенно определить, удалась ли вам расшифровать запись полностью либо только ее часть.

Перестановочная матрица показана на рис. 4.3. Серые стрелки указывают маршрут шифрования.

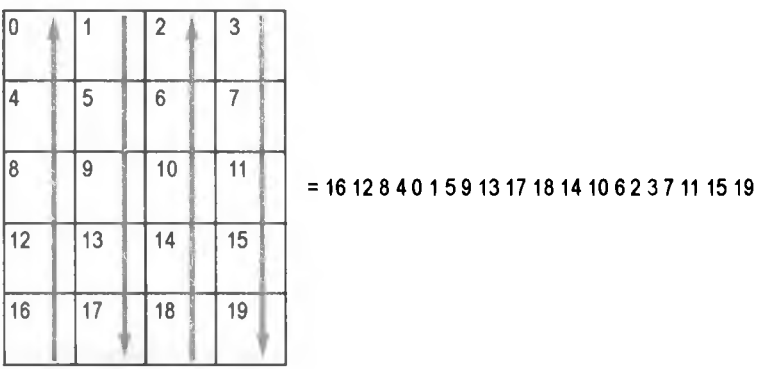


Рис. 4.3. Перестановочная матрица для управляющего сообщения с путем маршрутного шифра и результирующим шифротекстом

Ключ отслеживает как порядок, так и направление маршрута по столбцам. Маршрут не обязательно должен перемещаться по столбцам по порядку. Например, он может двигаться вниз по первому столбцу, вверх по третьему, вниз по четвертому и, наконец, вверх по второму. Отрицательные числа означают, что вы начинаете снизу и читаете столбец; положительные числа означают обратное. Для управляющего сообщения конечным ключом, используемым в программе, будет список: [–1, 2, –3, 4]. Этот список проинструктирует программу начать чтение вверх снизу столбца 1, перейти к верху столбца 2 и читать вниз, перейти к низу столбца 3 и читать вверх, а также перейти к верху столбца 4 и читать вниз.

Обратите внимание, что в ключах не следует задавать 0, потому что пользователи, будучи людьми, предпочитают начинать подсчет с 1. Разумеется, Python предпочитает начинать подсчет с 0, поэтому за кулисами вам нужно будет вычесть из значений ключа 1. В результате все выигрывают!

В разд. "Маршрутный перестановочный шифр: атака с применением грубой силы" далее в этой главе вы сможете применить эту компактную структуру ключа для того, чтобы продраться сквозь маршрутный шифр с помощью грубой силы, автоматически пробуя сотни ключей до тех пор, пока открытый текст не будет восстановлен.

Проектирование, заполнение и обнуление матрицы

Вы будете вводить шифротекст в виде непрерывной символьной цепочки. Для того чтобы ваша программа могла разгадать маршрут по этой цепочке, вам сначала нужно построить и заполнить переводную матрицу. Цепочка с шифротекстом представляет собой просто столбцы в перестановочной матрице на рис. 4.3, уложенные впритык в том порядке, в котором они были прочитаны. И поскольку в перестановочной матрице имеется пять строк, каждая группа из пяти элементов в шифротексте представляет собой отдельный столбец. Эту матрицу можно реализовать в виде списка списков:

```
>>> list_of_lists = [['16', '12', '8', '4', '0'], ['1', '5', '9', '13', '17'],
['18', '14', '10', '6', '2'], ['3', '7', '11', '15', '19']]
```

Элементы в этом новом списке теперь представляют списки — причем каждый список представляет столбец, — и пять элементов в каждом списке представляют строки, составляющие этот столбец. Это немного трудно представить, поэтому давайте напечатаем каждый из этих вложенных списков на отдельной строке:

```
>>> for i in range(len(list_of_lists)):
        print(list_of_lists[i])
[16, 12, 8, 4, 0]
[1, 5, 9, 13, 17]
[18, 14, 10, 6, 2]
[3, 7, 11, 15, 19]
```

Если вы читаете каждый список слева направо, начиная сверху, то следуете по перестановочному маршруту, который был вверх и вниз по чередующимся столбцам (см. рис. 4.3). С точки зрения Python первый столбец обозначается как `list-of-lists[0]` и отправной точкой является `list-of-lists[0][0]`.

Теперь нормализуем маршрут, считывая все столбцы в том же направлении, что и начальный столбец (вверх). Это требует изменения порядка следования элементов в каждом втором списке, как показано здесь жирным шрифтом:

```
[16, 12, 8, 4, 0]
[17, 13, 9, 5, 1]
[18, 14, 10, 6, 2]
[19, 15, 11, 7, 3]
```

Возникает шаблон. Если вы начинаете справа вверху и читаете каждый столбец вниз, заканчивающийся слева внизу, то числа находятся в числовом порядке; вы восстановили открытый текст!

Для того чтобы это повторить, скрипт может прокрутить в цикле каждый вложенный список, удаляя в этом списке последний элемент и добавляя его в новую цепочку до тех пор, пока переводная матрица не будет опустошена. Скрипт будет знать из ключа, какие вложенные списки ему нужно обратить, и порядок, в котором нужно опустошать матрицу. Результатом будет цепочка с восстановленным открытым текстом:

```
'0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19'
```

Теперь у вас должно быть очень общее представление о стратегии. Давайте опишем ее подробнее, а затем напишем псевдокод.

Псевдокод

Скрипт можно разбить на три главные части: ввод данных пользователем, заполнение переводной матрицы и расшифровка в открытый текст. Вы должны увидеть эти части в следующем ниже псевдокоде:

```
Загрузить символьную цепочку с шифротекстом.
Конвертировать шифротекст в шифросписок, разбив его на отдельные слова.
Получить входные данные с числом столбцов и строк.
Получить входные данные с ключом.
Конвертировать ключ в список, разбив на отдельные числа.
Создать новый список для переводной матрицы.
Для каждого числа в ключе:
    Создать новый список и добавить n элементов (n = число строк) из шифросписка.
    Использовать знак числа ключа, для того чтобы решить, как следует читать строку матрицы:
    вперед либо назад.
    Используя выбранное направление, добавить в матрицу новый список. Индекс каждого нового
    списка зависит от номера столбца, используемого в ключе.
Создать новую символьную цепочку для хранения результатов перевода.
Для диапазона строк матрицы:
    Для вложенного списка в переводной матрице:
        Удалить последнее слово во вложенном списке
        Добавить слово в цепочку с переводом.
Напечатать символьную цепочку с переводом.
```

Все, что предшествует первому циклу, по сути просто собирает и переформатирует шифровальные данные. Первый цикл отвечает за построение и заполнение матрицы, а второй создает из этой матрицы символьную цепочку с переводом. Наконец, цепочка с переводом печатается.

Код расшифровки маршрутного шифра

Листинг 4.1 принимает сообщение, закодированное маршрутным шифром, число столбцов и строк в перестановочной матрице и ключ, а затем показывает переве-

денный открытый текст. Он будет расшифровывать все "распространенные" маршрутные шифры, где маршрут начинается в верхней или нижней части столбца и продолжается вверх и/или вниз столбцов.

Эта версия является прототипной; после того как вы убедитесь, что она работает, вы упакуете ее для других пользователей. Указанный код можно скачать с вебсайта **https://www.nostarch.com/impracticalpython/**.

Листинг 4.1. Код расшифровки маршрутного шифра route_cipher_decrypt_prototype.py

```
❶ ciphertext = "16 12 8 4 0 1 5 9 13 17 18 14 10 6 2 3 7 11 15 19"

  # разбить элементы на слова, не на буквы
❷ cipherlist = list(ciphertext.split())
❸ # инициализировать переменные
  COLS = 4
  ROWS = 5
  key = '-1 2 -3 4' # отрицательное число означает чтение ВВЕРХ столбца,
                    # а не ВНИЗ
  translation_matrix = [None] * COLS
  plaintext = ''
  start = 0
  stop = ROWS

  # превратить key_int в список целых чисел:
❹ key_int = [int(i) for i in key.split()]
  # превратить столбцы в элементы списка списков:
❺ for k in key_int:
    ❻ if k < 0:   # читать в столбце снизу вверх
          col_items = cipherlist[start:stop]
      elif k > 0: # читать в столбце сверху вниз
          col_items = list((reversed(cipherlist[start:stop])))
  translation_matrix[abs(k) - 1] = col_items
  start += ROWS
  stop += ROWS

  print("\nшифротекст = {}".format(ciphertext))
  print("\nпереводная матрица =", *translation_matrix, sep="\n")
  print("\nдлина ключа = {}".format(len(key_int)))

  # обойти в цикле вложенные списки, передавая последний элемент
  # в новый список:
❼ for i in range(ROWS):
      for col_items in translation_matrix:
        ❽ word = str(col_items.pop())
❾ plaintext += word + ' '

  print("\nоткрытый текст = {}".format(plaintext))
```

Начните с загрузки шифротекста ❶ в виде цепочки символов. Вы хотите работать со словами, а не с буквами, поэтому разбейте символьную цепочку по пробелам, используя метод обработки символьных цепочек `split()` для создания нового списка с именем `cipherlist` ❷. Метод `split()` является обратным методу `join()`, который вы встречали раньше. Разбиение может производиться на любую символьную цепочку; указанный метод по умолчанию выполняется на отрезках пробелов, расположенных один за другим, удаляя каждый пробел, прежде чем перейти к следующему.

Теперь самое время ввести то, что вы знаете о шифре ❸: столбцы и строки, которые образуют матрицу, и ключ, который содержит маршрут. Инициализируйте числа столбцов и строк как константы. Затем создайте пустой список с именем `translation_matrix` для хранения содержимого каждого столбца в виде (вложенного) списка. Задайте значения заполнителей, умножив значения `None` на число столбцов. Индексы этих пустых элементов можно использовать с целью возврата столбцов в их правильный порядок для ключей, которые не находятся в числовом порядке.

Расшифрованное сообщение будет содержаться в пустой строке с именем `plaintext`.

Далее идут несколько параметров среза. Обратите внимание, что некоторые из них являются производными от числа строк, которое тождественно числу элементов в каждом столбце.

Теперь конвертируйте переменную `key`, являющуюся символьной цепочкой, в список целых чисел, используя операцию включения в список, т. е. сокращенный способ выполнения операций на списках ❹. Позже вы будете использовать числа в ключе в качестве индексов, поэтому они должны иметь целочисленный тип.

Следующим блоком кода является цикл `for`, который заполняет матрицу `translation_matrix`, являющуюся просто списком списков ❺. Поскольку каждый столбец становится вложенным списком, а длина списка `key_int` равна числу столбцов, диапазоном цикла является ключ, который также описывает маршрут.

Внутри цикла примените условное выражение для проверки того, является ли ключ положительным или отрицательным ❻; если ключ положительный, то направление среза меняется на противоположное. Назначьте срезу правильную позицию в `translation_matrix` на основе абсолютного значения ключа и вычтите 1 (т. к. ключи не содержат 0, а индексы списка содержат). Завершите цикл, продвинув конечные точки среза вперед на число строк и напечатав немного полезной информации.

Заключительный блок ❼ в цикле перебирает число строк (что эквивалентно числу слов в одном из вложенных списков) и каждый вложенный список. Первые два цикла показаны на рис. 4.4. Когда вы останавливаетесь в каждом вложенном списке, вы задействуете одну из моих любимых функций Python — списковый метод `pop()` ❽. Метод `pop()` выталкивает из списка и возвращает его последний элемент, если не указан определенный индекс. Он разрушает вложенный список, но он вам все равно больше не понадобится.

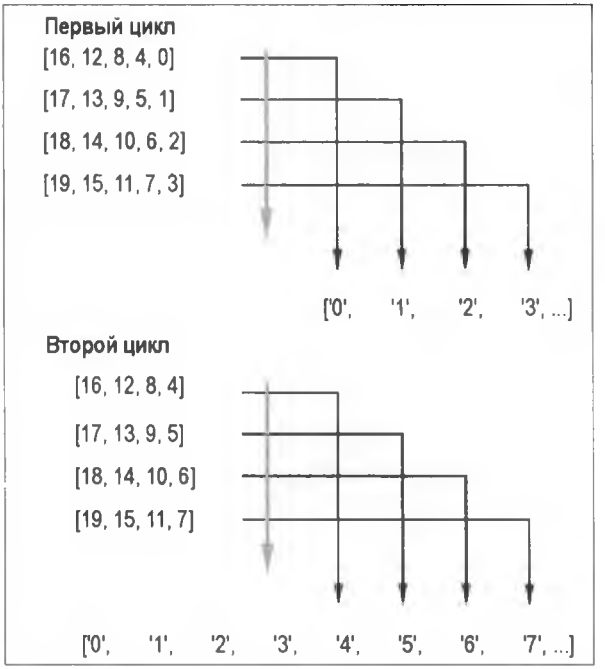


Рис. 4.4. Первый и второй циклы по вложенным спискам, удаление каждого конечного элемента и его добавление в символьную цепочку с переводом

После того как вы вытолкнули слово, присоедините его к символьной цепочке `plaintext` и добавьте пробел ❾. Осталось лишь показать расшифрованный шифротекст. Результат для тестового числового набора выглядят следующим образом:

```
открытый текст = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Похоже, все прошло успешно!

Взламывание маршрутного шифра

Приведенный выше код исходит из допущений, что вы знаете маршрут по матрице шифрования либо правильно угадали ключ. Если эти допущения являются ложными, то ваш единственный выход — попробовать все возможные расстановки ключа и матрицы. Возможность автоматизировать процесс выбора ключа — для заданного числа столбцов — вы получите в *разд. "Маршрутный перестановочный шифр: атака с применением грубой силы" далее в этой главе*. Но, как вы увидите, маршрутный шифр союзников хорошо укреплен против атак с применением грубой силы. Вы можете его взломать, но в итоге у вас получится так много данных, что вы будете чувствовать себя как собака, которая все преследовала машину и наконец ее догнала.

По мере того как сообщения становятся все длиннее, число возможных путей шифрования в перестановочном шифре становится слишком большим для того, чтобы его решить применением грубой силы, даже с использованием современных ком-

пьютеров. Например, если в матрице восемь столбцов и вы разрешаете маршруту переходить к любому столбцу, то число способов объединить столбцы является факториалом восьми: $8\times7\times6\times5\times4\times3\times2\times1=40\,320$. А это 40 320 путей. Учтите это, прежде чем начнете выбирать альтернативные маршруты по столбцам. Если маршрут может изменять направление вверх или вниз по столбцу, то число комбинаций увеличивается до 10 321 920. И если вы решите начать с любого места в столбце — а не в самом верху или внизу — и разрешите, чтобы через матрицу проходил любой маршрут (например, спираль), то все начнет действительно выходить из-под контроля!

По этой причине даже короткие перестановочные шифры могут иметь от тысяч до миллионов возможных путей. Даже если число путей поддается решению на компьютере и атака с применением грубой силы может восторжествовать, то вам все равно понадобится способ просеять несметное число результатов и либо выбрать победителя вычислительно, либо отобрать небольшое подмножество кандидатов для визуального изучения.

Для более распространенных *буквенных* перестановочных шифров легко написать функцию, которая обнаруживает английский язык, сравнивая каждую попытку расшифровки с файлом словаря. Если число расшифрованных слов, принадлежащих словарю, превышает определенный пороговый процент, то вы, возможно, взломали шифр. Схожим образом, если имеется высокая частота распространенных буквенный пар (диграмм), например *er*, *th*, *on* или *an*, то вы, возможно, нашли решение. К сожалению, этот подход не будет работать для *словарного* перестановочного шифра, подобного тому, который вы используете здесь.

Словари не способны помочь вам выяснить правильность расстановки слов. Для расстановки слов можно попробовать использовать такие подходы, как грамматические правила и вероятностные языковые модели, в частности *n*-граммы, прочесывая тысячи расшифровок и программно выбирая кандидатные результаты, но мудрое использование Стейджером кодовых имен и фиктивных слов в его маршрутном шифре значительно усложнит процесс.

Криптоаналитики считают, что, невзирая на только что описанные проблемы, короткие, прямолинейные перестановочные шифры довольно легко взломать без компьютера. Можно отыскать распространенные пары слов или букв, которые имеют смысл, и использовать их для того, чтобы угадать число строк в перестановочной матрице.

Для иллюстрации воспользуемся нашим управляющим сообщением, состоящим из чисел. На рис. 4.5 вы видите результаты шифротекста для матрицы 4×5, каждый из которых порождается, начиная с одного из четырех углов решетки и следуя чередующимся последовательным маршрутом. Все случаи включают повторение соседних чисел (на рис. 4.5 закрашены). Они указывают на то, где по решетке вы перемещаетесь в боковом направлении, и они дают наводки о схеме матрицы и пройденном по ней маршруте. Сразу видно, что строк было пять, потому что первым из каждой общей пары является пятое слово. Кроме того, зная, что в сообщении имеется 20 слов, вы узнаете, что число столбцов составило четыре ($20/5=4$). Исполь-

зуя разумное допущение о том, что текстовое сообщение было написано слева направо, можно даже угадать маршрут. Например, если вы начнете в правом нижнем углу, то вы пойдете вверх до 3, далее влево до 2, затем вниз до 18, затем влево до 17, затем вверх до 1 и влево до 0. Разумеется, со словами было бы сложнее, т. к. связь между ними не такая явная, но использование чисел действительно имеет смысл.

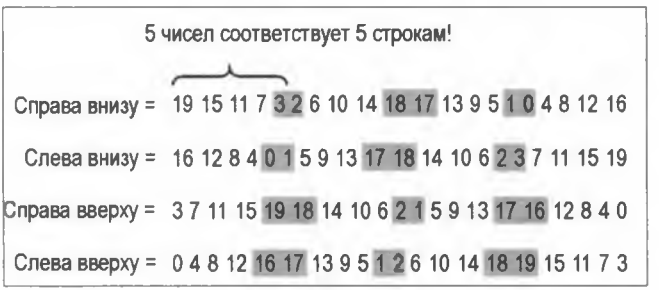


Рис. 4.5. Символы или слова в логическом порядке (затонированные) могут использоваться для угадывания маршрута шифрования

Взгляните на рис. 4.6 (на нем приведено сообщение из рис. 4.2). Конечные слова и возможные связанные слова, такие как "is just" (просто) или "heading to" (направляются), затенены.

Рис. 4.6. Взламывание маршрутного шифра из рис. 4.2 человеком. Указана пятистрочная матрица

Всего имеется 20 слов, для которых может быть 4, 5 или 10 строк матрицы. Сомнительно, что будет использоваться двухстолбцовая матрица, поэтому мы реально имеем дело с расстановкой 4×5 или 5×4. Если путь маршрутного шифра подобен тому, который показан на рис. 4.5, то мы ожидаем увидеть два незатененных слова между затененными словами для четырехстрочной матрицы и три незатененных слова для пятистрочной матрицы. Труднее придумать осмысленные словарные пары, которые соблюдают четырехстолбцовый шаблон, независимо от того, в каком направлении вы читаете шифротекст. Поэтому мы, вероятно, имеем дело с пятистолбцовым решением, которое начинается с левой стороны матрицы, поскольку связанные слова имеют смысл, если читать слева направо.

Обратите внимание, как затененные слова на рис. 4.6 заполняют верхнюю и нижнюю строки в перестановочной матрице из рис. 4.7. Это то, что мы и ожидали, т. к. путь "разворачивается" вверху и внизу каждого столбца. Графические решения — Божий дар для непонятливых!

Это выглядело просто, но опять же, мы знаем, как работает маршрутный шифр. Шифровальщики конфедерации в конце концов тоже его раскрыли, но использование кодовых слов лишило их полного входа в систему. Для того чтобы взломать

коды, им нужна была захваченная кодовая книга или крупная организация, способная собирать и анализировать большие данные, что в XIX в. было вне досягаемости конфедерации.

VILLAGE	ROANOKE	heading	to
GODWIN	With	SNOW	gone
you	are	free	to
transport	your	supplies	south
REST	IS	JUST	FILLER

Рис. 4.7. Затененные слова из рис. 4.6, помещенные в перестановочную матрицу

Добавление пользовательского интерфейса

Вторая цель этого проекта — написать программный код таким образом, чтобы уменьшить ошибки людей, в особенности тех, у кого меньше опыта (включая техников, стажеров, коллег и телеграфных служащих в 1864 г.). Разумеется, лучший способ сделать программу удобной для пользователя — это внедрить графический интерфейс пользователя (graphical user interface, GUI), но иногда это непрактично или невозможно. Например, программы для взламывания кода автоматически перебирают тысячи возможных ключей, и выполнить автогенерацию этих ключей проще, чем получить их непосредственно от пользователя.

В этом примере вы будете исходить из допущения о том, что пользователь взломает файл программы и введет в программу данные или даже внесет в код незначительные изменения. Вот некоторые рекомендации:

1. Начать с полезного литерала документирования *(см. главу 1)*.
2. Поместить вверху все необходимые данные, вводимые пользователем.
3. Использовать комментарии для уточнения требований к входным данным.
4. Четко отделить вводимые пользователем данные от остального кода.
5. Инкапсулировать большинство процессов в функциях.
6. Задействовать функции для улавливания предсказуемых ошибок пользователя.

Самое приятное в этом подходе то, что не оскорбляется ничей интеллект. Если пользователь хочет прокрутить вниз и посмотреть на код или даже изменить его, его ничто не остановит. Если он хочет лишь ввести несколько значений и получить черноящичное решение, то он тоже будет доволен. И мы отдали дань духу Энсона Стейджера, делая все проще и уменьшая вероятность ошибки.

Инструктаж пользователя и получение входных данных

В листинге 4.2 показан прототипный программный код, переупакованный для использования совместно с другими пользователями. Указанный код можно найти на веб-сайте **https://www.nostarch.com/impracticalpython/**.

Листинг 4.2. Литерал документирования, импорт и входные данные, вводимые пользователем для программы route_cipher_decrypt.py, часть 1

```
❶ """Расшифровать путь посредством маршрутного шифра союзников.

❷ Предназначено для полнословных перестановочных шифров с
переменными строками и столбцами. Исходит из допущения, что
шифрование началось вверху или внизу столбца.
Ключ указывает на порядок чтения столбцов и направление прохождения.
Отрицательные числа столбцов означают начало снизу и чтение вверх.
Положительные числа столбцов означают начало сверху и чтение вниз.

Приведенный ниже пример предназначен для матрицы 4x4 с ключом -1 2 -3 4.
Примечание: "0" не допускается.
Стрелки показывают маршрут шифрования; для отрицательных значений ключа
читать ВВЕРХ.

  1   2   3   4
___ ___ ___ ___
| ^ | | | ^ | | | СООБЩЕНИЕ ПИШЕТСЯ
|_|_|_v_|_|_|_v_|
| ^ | | | ^ | | | ПОПЕРЕК КАЖДОЙ СТРОКИ
|_|_|_v_|_|_|_v_|
| ^ | | | ^ | | | ТАКИМ ОБРАЗОМ
|_|_|_v_|_|_|_v_|
| ^ | | | ^ | | | ПОСЛЕДНЯЯ СТРОКА ЗАПОЛНЯЕТСЯ ФИКТИВНЫМИ СЛОВАМИ
|_|_|_v_|_|_|_v_|
НАЧАЛО      КОНЕЦ

Необходимые входные данные - текстовое сообщение, число столбцов,
число строк, символьная цепочка с ключом

Печатает переведенный текст

"""
❸ import sys

   #=====================================================================
❹ # ВХОДНЫЕ ДАННЫЕ ПОЛЬЗОВАТЕЛЯ:

❺ # дешифруемая символьная цепочка (введите или вставьте между
   # тройными кавычками):
   ciphertext = """16 12 8 4 0 1 5 9 13 17 18 14 10 6 2 3 7 11 15 19
   """

❻ # число столбцов в перестановочной матрице:
   COLS = 4
```

```
# число строк в перестановочной матрице:
ROWS = 5

❼ # ключ с пробелами между числами; отрицательный
# для чтения ВВЕРХ столбца
(например, -1 2 -3 4):
key = """ -1 2 -3 4 """

❽ # КОНЕЦ ВХОДНЫХ ДАННЫХ ПОЛЬЗОВАТЕЛЯ - НЕ РЕДАКТИРОВАТЬ НИЖЕ
# ЭТОЙ СТРОКИ!
#==================================================================
❾
```

Начните с многострочного литерала документирования в тройных кавычках ❶. Литерал документирования сообщает пользователю о том, что программа расшифровывает только типичный маршрутный шифр — тот, который начинается либо сверху, либо снизу столбца, — и о том, как вводить информацию о ключе ❷. Таблица включена для того, чтобы помочь уяснить смысл.

Затем импортируйте модуль `sys` для доступа к системным шрифтам и функциям ❸. Вы собираетесь проверить введенные пользователем данные по критерию приемлемости, поэтому вам потребуется возможность показывать сообщения в оболочке интерпретатора в привлекательном красном цвете. Размещение этой инструкции `import` здесь является примером взаимоисключающей ситуации. Поскольку стратегическая цель состоит в сокрытии рабочего кода от пользователя, вы, по сути дела, должны применить ее в программе гораздо позже. Однако правило языка Python по размещению всех инструкций импорта в верхней части программы является чересчур строгим, чтобы его игнорировать.

Теперь что касается раздела ввода данных. Как часто вы видели программный код или имели с ним дело, где результаты должны показываться или изменения должны вноситься на протяжении всего кода программы? Это может сбить с толку самого автора и, что еще хуже, другого пользователя. Поэтому ради удобства, пущей вежливости и предотвращения ошибок переместите все эти важные переменные наверх.

Сначала отделите раздел входных данных линией, а затем с помощью комментария в верхнем регистре сообщите пользователю о том, что он находится в указанном разделе ❹. Необходимые входные данные четко определены комментариями. Для текстовых входных данных можно использовать тройные кавычки. Это позволит лучше разместить длинные фрагменты текста. Обратите внимание, что я ввел символьную цепочку чисел из рис. 4.3 ❺. Далее пользователю необходимо добавить число столбцов и строк перестановочной матрицы ❻, а затем предложенный (или известный) ключ ❼.

Завершите раздел вводимых пользователем данных комментарием с объявлением на этот счет и предостережением не редактировать ничего ниже следующей стро-

ки ❽. Затем добавьте несколько дополнительных пустых строк для того, чтобы четче отделить раздел ввода данных от остальной части программы ❾.

Определение функции *main()*

В листинге 4.3 определена функция `main()`, которая выполняет программу и печатает открытый текст после декодирования шифра. Функция `main()` может быть определена до или после вызываемых функций, если она является последней вызываемой функцией.

Листинг 4.3. Определяет функцию `main()`. Файл route_cipher_decrypt.py, часть 2

```
def main():
    """Выполнить программу и напечатать расшифрованный открытый текст."""
❶   print("\nШифротекст = {}".format(ciphertext))
    print("Тестирование {} столбцов".format(COLS))
    print("Тестирование {} строк".format(ROWS))
    print("Тестирование ключа = {}".format(key))

    # разбить элементы на слова, не на буквы
❷   cipherlist = list(ciphertext.split())
❸   validate_col_row(cipherlist)
❹   key_int = key_to_int(key)
❺   translation_matrix = build_matrix(key_int, cipherlist)
❻   plaintext = decrypt(translation_matrix)

❼   print("Открытый текст = {}".format(plaintext))
```

Начните функцию `main()`, напечатав введенные пользователем данные в интерактивной оболочке ❶. Затем превратите шифротекст в список, разбив его на пробелы, как это было сделано в прототипном программном коде ❷.

Следующая серия инструкций вызывает функции, которые вы определите в ближайшее время. Первая проверяет, являются ли входные строки и столбцы приемлемыми для длины сообщения ❸. Вторая конвертирует переменную `key` из символьной цепочки в список целых чисел ❹. Третья строит переводную матрицу ❺, а четвертая выполняет дешифровальный алгоритм на матрице и возвращает символьную цепочку с открытым текстом ❻. Завершите `main()` печатью открытого текста ❼.

Верификация данных

Когда вы продолжите упаковывать route_cipher_decrypt.py для конечного пользователя, вам необходимо проверить допустимость входных данных. Листинг 4.4 предвосхищает распространенные ошибки пользователя и предоставляет пользователю полезные отклики и рекомендации.

Листинг 4.4. Определяет функции для проверки и подготовки введенных пользователем данных. Файл route_cipher_decrypt.py, часть 3

```
❶ def validate_col_row(cipherlist):
      """Проверить, что входные столбцы и строки приемлемы
         по отношению к длине сообщения."""
      factors = []
      len_cipher = len(cipherlist)
   ❷ for i in range(2, len_cipher): # диапазон исключает
                                    # 1 столбец шифров
          if len_cipher % i == 0:
              factors.append(i)
   ❸ print("\nДлина шифра = {}".format(len_cipher))
      print("Приемлемые значения столбцов/строк включают: {}".
            format(factors))
      print()
   ❹ if ROWS * COLS != len_cipher:
          print("\nОшибка - входные столбцы и строки не являются"
                "кратными длины шифра. Завершение программы.",
                file=sys.stderr)
          sys.exit(1)

❺ def key_to_int(key):
      """Превратить ключ в список целых чисел и проверить допустимость."""
   ❻ key_int = [int(i) for i in key.split()]
      key_int_lo = min(key_int)
      key_int_hi = max(key_int)
   ❼ if len(key_int) != COLS or key_int_lo < -COLS or key_int_hi > COLS \
         or 0 in key_int:
      ❽ print("\nОшибка - проблема с ключом. Завершение.",
              file=sys.stderr)
         sys.exit(1)
      else:
      ❾ return key_int
```

Функция `validate_col_row()` проверяет, что входные номера столбцов и строк соответствуют длине шифросписка, который передается в качестве аргумента ❶. Перестановочная матрица всегда имеет тот же размер, что и число слов в сообщении, поэтому число столбцов и число строк должны быть кратными размеру сообщения. Для того чтобы определить все допустимые кратные, сначала составьте пустой список для хранения кратных, а затем получите длину шифросписка. Используйте шифросписок, а не входной шифротекст, т. к. элементы в шифротексте являются буквами, а не словами.

Для получения кратных чисел обычно использовался бы диапазон (1, число + 1), но вы не хотите, чтобы эти конечные точки были в списке `factors`, т. к. переводная

матрица с этими размерностями была бы просто открытым текстом. Поэтому исключите эти значения из диапазона ❷. Так как кратное число делится без остатка на это число, для отыскания кратных используйте оператор деления по модулю (%) и затем добавьте их в список `factors`.

Затем покажите пользователю полезную информацию: длину шифросписка и допустимые варианты строк и столбцов ❺. Наконец, перемножьте два выбранных пользователем варианта между собой и сравните произведение с длиной шифросписка. Если они не совпадают, то покажите в оболочке интерпретатора красное предупреждающее сообщение (используя наш старый трюк `file=sys.stderr`) и завершите программу ❹. Примените `sys.exit(1)`, т. к. 1 указывает на аварийный выход из программы.

Теперь определите функцию для проверки ключа и конвертируйте его из символьной цепочки в список ❺. Передайте ей в качестве аргумента переменную `key`. Выделите каждый элемент в `key` и конвертируйте его в целое число; назовите список `key_int`, для того чтобы отличить его от введенной пользователем переменной `key` ❻. Далее определите минимальное и максимальное значения в списке `key_int`. Затем примените инструкцию `if`, с целью обеспечить, чтобы список содержал то же само число элементов, что и столбцы, и ни один из элементов в `key` не был слишком большим, слишком маленьким либо равным 0 ❼. Завершите работу программы сообщением об ошибке, если какой-либо из этих критериев не проходит ❽. В противном случае верните список `key_int` ❾.

Построение и декодирование переводной матрицы

Листинг 4.5 определяет две функции: одну для построения переводной матрицы, другую для ее декодирования, — и вызывает функцию `main()` как модуль либо в автономном режиме.

Листинг 4.5. Определяет функции для построения и декодирования переводной матрицы. Файл route_cipher_decrypt.py, часть 4

```
❶ def build_matrix(key_int, cipherlist):
      """Превратить каждые n элементов в списке в новый элемент
         в списке списков."""
      translation_matrix = [None] * COLS
      start = 0
      stop = ROWS
      for k in key_int:
          if k < 0:    # читать в столбце снизу-вверх
              col_items = cipherlist[start:stop]
          elif k > 0:  # читать в столбце сверху-вних
              col_items = list((reversed(cipherlist[start:stop])))
          translation_matrix[abs(k) - 1] = col_items
          start += ROWS
          stop += ROWS
      return translation_matrix
```

```
❷ def decrypt(translation_matrix):
      """Перебрать вложенные списки в цикле, выталкивая последний элемент
         в символьную цепочку."""
      plaintext = ''
      for i in range(ROWS):
          for matrix_col in translation_matrix:
              word = str(matrix_col.pop())
              plaintext += word + ' '
      return plaintext

❸ if __name__ == '__main__':
      main()
```

Эти две функции представляют собой инкапсуляцию кода из программы route_cipher_decrypt_prototype.py. Подробное описание см. в листинге 4.1.

Сначала определите функцию для построения переводной матрицы; передайте ей в качестве аргументов переменные `key_int` и `cipherlist` ❶. Указанная функция возвращает список списков.

Затем упакуйте дешифровальный код, где выталкивается конец каждого вложенного списка, как функция, которая в качестве аргумента использует список `translation_matrix` ❷. Верните открытый текст, который будет напечатан функцией `main()`.

Завершите программу с помощью условной инструкции, которая позволяет программе работать как модуль либо в автономном режиме ❸.

Если вы являетесь случайным или разовым пользователем этого кода, то вы оцените, насколько он прост и доступен. Если вы планируете изменить указанный код для своих собственных нужд, то вы также оцените то, что ключевые переменные легко доступны, а главные задачи модуляризованы. Вам не придется копаться в программе, чтобы вычленить важные части или понять разницу между потайными переменными, такими как `list1` и `list2`.

Вот результат работы программы с учетом шифротекста из рис. 4.3:

```
Шифротекст = 16 12 8 4 0 1 5 9 13 17 18 14 10 6 2 3 7 11 15 19

Тестирование 4 столбцов
Тестирование 5 строк
Тестирование ключа = -1 2 -3 4

Длина шифра = 20
Приемлемые значения столбцов/строк включают: [2, 4, 5, 10]

Открытый текст = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Теперь вы можете легко расшифровать маршрутный перестановочный шифр с известным ключом либо протестировать подозреваемые маршруты, используя понят-

ный и доступный интерфейс скрипта по настройке ключа. Вы получите шанс по-настоящему взломать один из этих шифров, автоматически тестируя все возможные ключи, *в разд. "Маршрутный перестановочный шифр: атака с применением грубой силы" далее в этой главе.*

Проект 9: зигзагообразный шифр

Офицеры конфедератов и шпионы в значительной степени были предоставлены сами себе всякий раз, когда дело доходило до криптографии. Это привело к нехитрым решениям, причем одним из излюбленных был зигзагообразный шифр, или шифр жердевой изгороди (rail fence cipher), названный так из-за его сходства с зигзагообразным рисунком разъемного жердевого забора (показан на рис. 4.8).

Рис. 4.8. Жердевая изгородь

Зигзагообразный шифр является простым в использовании перестановочным шифром, подобным маршрутному шифру союзников, но отличается от маршрутного шифра тем, что он переставляет буквы, а не слова, и это делает его более подверженным ошибкам. И поскольку число возможных ключей намного ограниченнее, чем число путей по маршрутному шифру, зигзагообразный шифр намного легче "снести".

И союзники, и конфедераты использовали зигзагообразный шифр как полевой шифр, а шпионы, вероятно, не очень часто использовали кодовые слова. По очевидным причинам шифровальные книги должны были строго контролироваться и,

скорее всего, хранились в военной телеграфной конторе, а не носились с собой легко раскрывавшимися агентами под прикрытием.

Иногда для важных сообщений конфедераты использовали более сложный шифр Виженера *(см. "Проект 12: сокрытие шифра Виженера" в главе 6)* — и для некоторых неважных с целью ввести в заблуждение противника, но его расшифровка выливалась в утомительную работу, а его шифрование становилось столь же трудоемким, что не подходило для оперативной полевой связи.

Несмотря на отсутствие подготовки в области криптографии, конфедераты и южане в целом были умны и изобретательны. Наиболее впечатляющим достижением в искусстве секретных сообщений было использование микрофотографии, за 100 лет до того, как она получила широкое распространение во время холодной войны.

Цель

Написать программы на языке Python, которые помогут шпиону шифровать и дешифровывать "двухрядные" (двухстрочные) зигзагообразные шифры. Вы должны написать программы так, чтобы уменьшить потенциальные ошибки неопытных пользователей.

Стратегия

Для шифровки сообщения с помощью зигзагообразного шифра выполните действия, описанные на рис. 4.9 (перевод шифруемого текста — купи еще мэнской картошки).

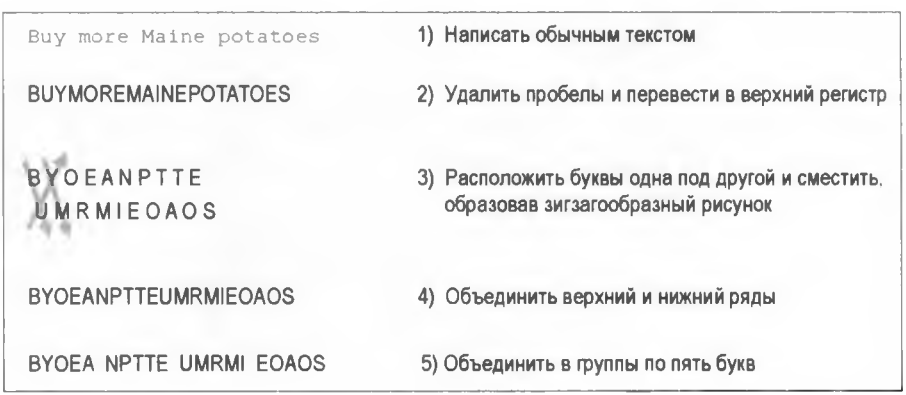


Рис. 4.9. Процесс шифрования для "двухрядного" зигзагообразного шифра

После записи открытого текста удаляются пробелы, и все буквы конвертируются в верхний регистр (шаг 2). В криптографии использование прописных букв является общепринятым правилом, поскольку оно запутывает наличие собственных имен и начала предложений, давая криптоаналитику меньше подсказок для расшифровки сообщения.

Далее сообщение записывается внакладку, где каждая вторая буква расположена ниже предыдущей буквы и сдвигается на один пробел (шаг 3). Именно здесь становится очевидной аналогия с "жердевой изгородью".

Потом в линию пишется первая строка, за которой сразу же следует вторая строка на той же линии (шаг 4), а затем буквы разбиваются на пятибуквенные группы для того, чтобы создать иллюзию отдельных слов и еще больше запутать криптоаналитика (шаг 5).

Для расшифровки зигзагообразного шифра указанный процесс выполняется в обратном порядке. Просто удалите пробелы, разделите сообщение пополам, разместите вторую половину ниже первой, смещенную на одну букву, и прочитайте сообщение, используя зигзагообразный рисунок. Если в шифротексте число букв является нечетным, то разместите в первую (верхнюю) половину лишнюю букву.

Для облегчения работы людям, которые хотят использовать зигзагообразный шифр, выполните приведенные выше шаги, написав две программы: одну для шифровки, а другую для расшифровки. По сути, рис. 4.9 является вашим псевдокодом, так что давайте перейдем к нему. И поскольку теперь вы знаете, как упаковывать код для неопытных пользователей, с самого начала примите этот подход.

Код шифровки зигзагообразным шифром

Программный код этого раздела позволяет пользователю ввести текстовое сообщение и распечатать зашифрованные результаты в окне интерпретатора. Указанный код доступен для скачивания с ресурсов книги по адресу **https://www.nostarch.com/impracticalpython/**.

Инструктаж пользователя и получение входных данных

Листинг 4.6 предоставляет инструкции пользователю вверху программы rail_fence_cipher_encrypt.py и назначает переменной открытый текст.

Листинг 4.6. Раздел документирования и данных, вводимых пользователем для rail_fence_cipher_encrypt.py, часть 1

```
❶ r"""Зашифровать текст зигзагообразным шифром времен Гражданской войны
      в США. Этот шифр является "двухрядным" и предназначен для
      коротких сообщений. Пример шифруемого текста:
      'Buy more Maine potatoes' (Купи еще мэнской картошки)
   Зигзагообразный стиль: B Y O E A N P T T E
                           U M R M I E O A O S
   Чтение зигзага:        \/\/\/\/\/\/\/\/\/\/
   Шифротекст: BYOEA NPTTE UMRMI EOSOS
   """
   #-------------------------------------------------------------------------
❷ # ВХОДНЫЕ ДАННЫЕ ПОЛЬЗОВАТЕЛЯ:

   # шифруемая символьная цепочка (вставьте между кавычками):
❸ plaintext = """Давай пересечем реку и отдохнем в тени деревьев"""
❹ # КОНЕЦ ВХОДНЫХ ДАННЫХ ПОЛЬЗОВАТЕЛЯ - НЕ РЕДАКТИРОВАТЬ
   # НИЖЕ ЭТОЙ СТРОКИ!
   #-------------------------------------------------------------------------
```

Начните с многострочного литерала документирования, поместив префикс r (от англ. *raw* — сырой) перед первым набором тройных кавычек ❶. Без этого префикса программа Pylint будет горько жаловаться на косые черты \\, используемые ниже. К счастью, инструмент pydocstyle укажет на это, так что вы сможете все исправить (прочитайте *главу 1*, где узнаете все о программах Pylint и pydocstyle). Если вы хотите побольше узнать о необработанных (сырых) символьных цепочках, то см. раздел 2.4.1 в документации по языку Python (**https://docs.python.org/3.6/reference/lexical_analysis.html#string-and-bytes-literals**).

Затем отделите литерал документирования и инструкции `import` программы от раздела входных данных пустой строкой и сообщите пользователю, что он находится в указанном разделе, с помощью комментария в верхнем регистре ❷. При помощи комментариев четко определите требования к входным данным и поместите открытый текст внутри тройных кавычек для того, чтобы лучше уместить длинный текст ❸.

Наконец, завершите раздел вводимых пользователем данных объявлением на этот счет и предостережением не редактировать ничего ниже следующей далее строки кода ❹.

Шифрование сообщения

В программу rail_fence_cipher_encrypt.py добавьте листинг 4.7, который будет заниматься процессами шифрования.

Листинг 4.7. Определяет функции для шифрования текста открытого сообщения. Файл rail_fence_cipher_encrypt.py, часть 2

```
❶ def main():
      """Выполнить программу шифрования сообщения с помощью
         двухрядного зигзагообразного шифра."""
      message = prep_plaintext(plaintext)
      rails = build_rails(message)
      encrypt(rails)

❷ def prep_plaintext(plaintext):
      """Удалить пробелы и начальные/замыкающие пробелы."""
   ❸ message = "".join(plaintext.split())
   ❹ message = message.upper() # общепринято шифротекст писать
                              # в верхнем регистре
      print("\nоткрытый текст = {}".format(plaintext))
      return message

❺ def build_rails(message):
      """Построить символьные цепочки, беря каждый второй символ
         в сообщении."""
      evens = message[::2]
      odds = message[1::2]
```

```
    ❻ rails = evens + odds
       return rails

❼ def encrypt(rails):
       """Разбить буквы в шифротексте на куски по 5 букв и сцепить
          в одну цепочку."""
    ❽ ciphertext = ' '.join([rails[i:i+5] for i in range(0,
                                                        len(rails), 5)])
       print("шифротекст = {}".format(ciphertext))

❾ if __name__ == '__main__':
       main()
```

Для начала определите функцию `main()`, которая выполняет программу ❺. Наличие функции `main()` даст вам позже возможность использовать эту программу в качестве модуля в другой программе, если возникнет такая необходимость. Указанная функция вызывает три другие функции: одну для подготовки входного открытого текста, другую — для построения "рядов", используемых шифром, и третью — для разбиения зашифрованного текста на пятибуквенные фрагменты.

Далее определите функцию, которая примет входную строку и подготовит ее к шифрованию ❸. Сюда входит удаление пробелов ❹ и конвертирование букв в верхний регистр (как в шаге 2 на рис. 4.9) ❹. Затем, после новой строки, напечатайте открытый текст на экране и верните его.

Теперь определите функцию для построения двух символьных цепочек, как в шаге 3 на рис. 4.9, нарезая сообщение на четные (начиная с 0 и шагая на 2 позиции) и нечетные (начиная с 1 и шагая на 2 позиции) символы ❻. Далее эти две символьные цепочки сцепите в новую символьную цепочку с именем `rails` ❽, которая затем возвращается.

Определите функцию шифрования, которая в качестве аргумента принимает символьную цепочку `rails` ❼. Примените операцию включения в список для разбиения шифротекста на фрагменты по пять букв (как в шаге 5 на рис. 4.9) ❽. Зашифрованный текст выведите на экран. Закончите фрагментом кода, который выполняет программу в виде модуля либо в автономном режиме ❾.

Вот вывод из этой программы:

```
открытый текст = Let us cross over the river and rest under the shade of the
                                                                        trees
шифротекст = LTSRS OETEI EADET NETEH DOTER EEUCO SVRHR VRNRS UDRHS AEFHT ES
```

Код расшифровки зигзагообразного шифра

Код этого раздела позволяет пользователю ввести сообщение, зашифрованное зигзагообразным шифром, и напечатать открытый текст в окне интерпретатора. Этот код доступен для скачивания вместе с остальными ресурсами книги по адресу **https://www.nostarch.com/impracticalpython/**.

Импорт модулей, инструктаж пользователя и получение входных данных

Листинг 4.8 начинается инструктажем, аналогичным инструктажу в программе rail_fence_cipher_encrypt.py (см. листинг 4.6), импортирует два модуля и получает от пользователя входные данные.

Листинг 4.8. Импортирует модули, инструктирует пользователя и получает от пользователя входные данные. Файл rail_fence_cipher_decrypt.py, часть 1

```
r"""Дешифровать зигзагообразный шифр времен Гражданской войны в США.

Этот шифр является "двухрядным" и предназначен для коротких сообщений.

Пример шифруемого текста: 'Buy more Maine potatoes'
                          (Купи еще мэнской картошки)

Зигзагообразный стиль: B Y O E A N P T T E
                        U M R M I E O A O S

Чтение зигзага:        \/\/\/\/\/\/\/\/\/\/

Шифротекст: BYOEA NPTTE UMRMI EOSOS
"""
❶ import math
   import itertools

#-----------------------------------------------------------------------
# ВХОДНЫЕ ДАННЫЕ ПОЛЬЗОВАТЕЛЯ:
# дешифруемая символьная цепочка (вставить между кавычками):
❷ ciphertext = """LTSRS OETEI EADET NETEH DOTER EEUCO SVRHR VRNRS UDRHS AEFHT ES

"""

# КОНЕЦ ВХОДНЫХ ДАННЫХ ПОЛЬЗОВАТЕЛЯ - НЕ РЕДАКТИРОВАТЬ НИЖЕ ЭТОЙ СТРОКИ!
#-----------------------------------------------------------------------
```

Одним из отличий здесь является то, что вам нужно импортировать модули `math` и `itertools` ❶. Модуль `math` будет использоваться для округления. Модуль `itertools` представляет собой группу функций в стандартной библиотеке Python, которые создают итераторы с целью эффективной циклической обработки. Функция `zip_longest()` из модуля `itertools` будет использоваться во время процесса расшифровки.

Другое единственное изменение заключается в том, что вместо открытого текста пользователь должен ввести шифротекст ❷.

Расшифровка сообщения

Листинг 4.9 определяет функции для подготовки и декодирования шифротекста и завершает работу программы rail_fence_cipher_decrypt.py.

Листинг 4.9. Подготовка, декодирование и печать сообщения. Файл rail_fence_cipher_decrypt.py, часть 2

```
❶ def main():
      """Выполнить программу для дешифровки двухрядного зигзагообразного
         шифра."""
      message = prep_ciphertext(ciphertext)
      row1, row2 = split_rails(message)
      decrypt(row1, row2)

❷ def prep_ciphertext(ciphertext):
      """Удалить пробелы."""
      message = "".join(ciphertext.split())
      print("\nшифротекст = {}".format(ciphertext))
      return message

❸ def split_rails(message):
      """Разбить сообщение на два, всегда округляя ВВЕРХ
         для первого ряда."""
   ❹ row_1_len = math.ceil(len(message)/2)
   ❺ row1 = (message[:row_1_len])
      row2 = (message[row_1_len:])
      return row1, row2

❻ def decrypt(row1, row2):
      """Построить список, чередуя буквы из 2 цепочек символов,
         и напечатать."""
   ❼ plaintext = []
   ❽ for r1, r2 in itertools.zip_longest(row1, row2):
          plaintext.append(r1.lower())
          plaintext.append(r2.lower())
   ❾ if None in plaintext:
          plaintext.pop()
      print("ряд 1 = {}".format(row1))
      print("ряд 2 = {}".format(row2))
      print("\nоткрытый текст = {}".format(''.join(plaintext)))

❿ if __name__ == '__main__':
      main()
```

Функция `main()` здесь ❶ аналогична функции, используемой в программе шифрования из листинга 4.7. Вызываются три функции: одна для подготовки входной

символьной цепочки, другая — для "разбиения рядов" в зигзагообразном шифре и третья — для сшивания двух рядов вместе в читаемый открытый текст.

Начните с функции, которая повторяет шаги предобработки, используемые во время шифрования ❷. Удалите пробелы между пятибуквенными фрагментами, а также любые другие пробелы, созданные во время вставки зашифрованного текста, напечатайте и верните шифротекст.

Далее вам нужно разбить сообщение обратно на две части, обратив процесс шифрования вспять ❸. Как я уже упоминал в *разд. "Стратегия" ранее в этой главе*, лишняя буква в сообщении с нечетным числом символов назначается верхнему ряду. Для улаживания нечетного случая используйте метод `math.ceil()` ❹. "Ceil" является сокращением английского слова "ceiling" и означает потолок, поэтому при делении на 2 ответ всегда округляется вверх до ближайшего целого числа. Назначьте это число переменной `row_1_len`. После того как вы узнаете длину первого ряда, используйте это значение и срез для того, чтобы разбить переменную `message` на две представляющие ряды символьные цепочки ❺. Завершите функцию, вернув переменные рядов.

Теперь это просто вопрос выбора и объединения каждой второй буквы из рядов, сшивая открытый текст вместе. Определите функцию `decrypt()` и передайте ей символьные цепочки для рядов `row1` и `row2` ❻. Начните процесс перевода, создав пустой список для хранения результатов ❼. Далее вам нужен простой способ справиться со случаями, когда шифротекст имеет нечетное число букв, что приводит к двум рядам разной длины — потому что Python не даст вам пройти по двум неравным последовательностям и вызовет ошибку "Индекс вне диапазона". Вот почему мы импортировали модуль `itertools` — его функции помогают с циклической обработкой, обходя эту проблему.

Функция `itertools.zip_longest()` в качестве аргументов принимает две символьные цепочки и выполняет их циклическое прохождение без жалоб, добавляя пустое значение (`None`) в список `plaintext`, когда она доходит до конца более короткой цепочки ❽. Вряд ли вы захотите напечатать это пустое значение, поэтому, если оно есть, удалите его с помощью метода `pop()`, который вы применили в коде маршрутного шифра ❾. Завершите процесс расшифровки, напечатав два ряда (две жерди) на экране, а затем расшифрованный шифротекст.

Завершите стандартным фрагментом кода для выполнения указанной программы в виде модуля либо в автономном режиме ❿. Результат работы программы выглядит следующим образом:

```
шифротекст = LTSRS OETEI EADET NETEH DOTER EEUCO SVRHR VRNRS UDRHS AEFHT ES

ряд 1 = LTSRSOETEIEADETNETEHDOTERE
ряд 2 = EUCOSVRHRVRNRSUDRHSAEFHTES

открытый текст = letuscrossovertheriverandrestundertheshadeofthetrees
```

Обратите внимание, что между словами не будет пробелов, но это нормально — вы же не хотите, чтобы криптоаналитик чувствовал себя совершенно бесполезным!

Резюме

На этом наш набег на шифры времен Гражданской войны в США завершается. Вы написали программу, которая помогает пользователю расшифровывать маршрутный шифр, и получили ценную информацию о том, как он работает и как его взломать. В последующих практических проектах вы можете реализовать автоматическую атаку на шифр, но помните, с учетом его многочисленных возможных путей и использования кодовых слов маршрутный шифр союзников является крепким орешком и потребуется много усилий, чтобы его взломать полностью.

Затем вы написали программы для шифрования и дешифрования двухрядных зигзагообразных шифров. Учитывая, насколько утомительны и подвержены ошибкам ручные процессы шифрования и дешифрования, наличие автоматизированного способа выполнения подавляющей части работы было бы ценным вкладом для обеих сторон во время той войны. И для дальнейшего решения этих задач вы написали свой код, удобный для неопытного криптоаналитика или шпиона.

Дальнейшее чтение

Еще больше программ Python начального уровня для работы с перестановочными шифрами есть в книге Эла Свейгарта "Взламывание кодов с помощью Python" (Sweigart A. Cracking codes with Python. No Cramp Press, 2018).

Превосходные и хорошо иллюстрированные обзоры криптографии можно найти в книге Гэри Блэквуда "Таинственные сообщения: история кодов и шифров" (Blackwood G. Mysterious messages: a history of codes and ciphers. The Penguin Group, 2009) и книге Саймона Сингха "Книга кодов: наука секретаря от Древнего Египта до квантовой криптографии" (Singh S. The code book: the science of secretary from ancient egypt to quantum cryptography. Anchor, 2000).

Веб-страницы **http://www.civilwarsignals.org/pages/crypto/crypto.html** и **http://www.mathaware.org/mam/06/Sauerberg_route-essay.html** включают описания попытки Эдварда Портера Александера (Edward Porter Alexander) решить маршрутный шифр. Александер был основателем корпуса связи армии конфедератов и блестящим военным новатором со многими впечатляющими достижениями.

Практические проекты

Отточите свои навыки криптографии с помощью приведенных ниже проектов. Решения доступны в приложении к книге и онлайн на веб-сайте книги.

Взламывание сообщения Линкольна

В своей книге "Таинственные сообщения: история кодов и шифров" Гэри Блэквуд воспроизводит фактическое сообщение, отправленное Авраамом Линкольном и зашифрованное маршрутным шифром:

THIS OFF DETAINED ASCERTAIN WAYLAND CORRESPONDENTS
OF AT WHY AND IF FILLS IT YOU GET THEY NEPTUNE THE
TRIBUNE PLEASE ARE THEM CAN UP

Примените программу route_cipher_decrypt.py для решения этой криптограммы. Число столбцов и строк должно быть кратным длине сообщения, а маршрут начинается в одном из углов, не пропускает столбцы и меняет направление при каждом изменении столбца. Определения кодового слова и решение с открытым текстом можно найти в приложении к книге.

Идентификация типов шифров

Чем скорее вы узнаете, с каким типом шифра имеете дело, тем скорее сможете его взломать. Словарные перестановочные шифры легко обнаружить, но буквенные перестановочные шифры могут выглядеть как буквенные подстановочные шифры. К счастью, отличить один от другого можно, используя частоту встречаемости букв в зашифрованном тексте. Поскольку в перестановочных шифрах буквы просто скремблируются и не заменяются, их частотное распределение будет таким же, как и для языка, на котором был написан открытый текст. Исключением, однако, являются военные сообщения, в которых используется жаргон и опускаются многие распространенные слова. Для этого вам нужна частотная таблица, построенная из других военных сообщений.

Напишите программу Python, которая на входе принимает символьную цепочку шифротекста и определяет, является ли он, скорее всего, перестановочным либо подстановочным шифром. Проверьте его с помощью файлов cipher_a.txt и cipher_b.txt, которые можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**. Решение задачи можно найти в приложении к книге и онлайн на веб-сайте книги в файле identify_cipher_type_practice.py.

Хранение ключа в форме словаря

Напишите короткий скрипт, который разбивает ключ маршрутного шифра на две части: одну для регистрации порядка следования столбцов, а другую — для регистрации направления чтения строк в столбце (вверх или вниз). Сохраните номер столбца в качестве ключа словаря и направление чтения в качестве значения словаря. Пусть программа в интерактивном режиме запросит у пользователя значение ключа для каждого столбца. Решение можно найти в приложении к книге и онлайн в файле key_dictionary_practice.py.

Автоматическое генерирование возможных ключей

Попытка расшифровать маршрутный шифр, используя любую комбинацию столбцов на своем пути, потребует знания о том, какие это комбинации. Это необходимо для того, чтобы можно было ввести их в качестве аргументов в функцию расшифровки. Напишите программу Python, которая принимает целое число (например, число столбцов) и возвращает коллекцию кортежей. Каждый кортеж должен со-

держать уникальный порядок следования номеров столбцов, например (`1, 2, 3, 4`). Включите отрицательные значения, например (`2, -3, 4, -1`) для того, чтобы уловить маршруты шифрования, которые идут по столбцам вверх в отличие от движения по столбцам вниз. Решение представлено в приложении к книге со скачиваемой версией на веб-сайте книги в файле permutations_practice.py.

Маршрутный перестановочный шифр: атака с применением грубой силы

Скопируйте и модифицируйте программу route_cipher_decrypt.py для взлома маршрутного шифра на рис. 4.2. Вместо того чтобы вводить один ключ, переберите все возможные ключи — для предполагаемого числа столбцов — и напечатайте результаты (используйте приведенный ранее перестановочный код, который генерирует ключи для этого четырехстолбцового шифра). Переключение порядка следования столбцов и разрешение восходящих и нисходящих путей по перестановочной матрице имеет сильное влияние, что четко проиллюстрировано на рис. 4.10. Пунктирная линия — это факториал числа столбцов; сплошная линия улавливает эффект чтения в столбцах вверх, а также вниз (улавливается за счет включения в ключ отрицательных значений). Если бы вам пришлось иметь дело только с факториалом 4, то ваша работа в качестве криптоаналитика была бы легкой. Но по мере того, как шифр становится длиннее, число возможных ключей претерпевает взрывной рост. И некоторые фактические маршрутные шифры союзников имели 10 столбцов и более!

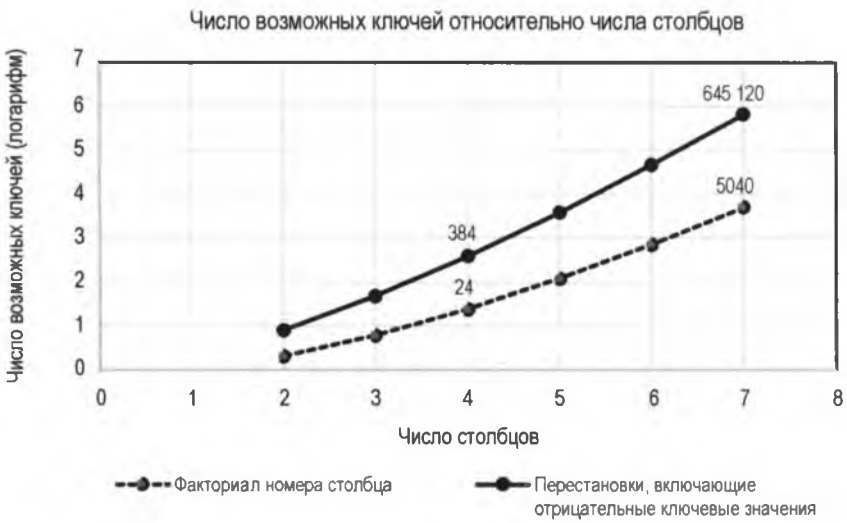


Рис. 4.10. Число возможных ключей относительно числа столбцов для маршрутного шифра

Вот четыре перевода из 384, произведенных для шифротекста на рис. 4.2:

```
с использованием ключа = [-4, -1, -2, -3]
перевод = IS HEADING FILLER VILLAGE YOUR SNOW SOUTH GODWIN ARE FREE TO YOU
WITH SUPPLIES GONE TRANSPORT ROANOKE JUST TO REST
```

```
с использованием ключа = [1, 2, -3, 4]
перевод = REST ROANOKE HEADING TO TRANSPORT WITH SNOW GONE YOU ARE FREE TO
GODWIN YOUR SUPPLIES SOUTH VILLAGE IS JUST FILLER

с использованием ключа = [-1, 2, -3, 4]
перевод = VILLAGE ROANOKE HEADING TO GODWIN WITH SNOW GONE YOU ARE FREE TO
TRANSPORT YOUR SUPPLIES SOUTH REST IS JUST FILLER

с использованием ключа = [4, -1, 2, -3]
перевод = IS JUST FILLER REST YOUR SUPPLIES SOUTH TRANSPORT ARE FREE TO YOU
WITH SNOW GONE GODWIN ROANOKE HEADING TO VILLAGE
```

Правильный ответ присутствует, но вы можете по достоинству оценить то, как трудно было бы выбрать его оперативно, учитывая использование кодовых и фиктивных слов. Тем не менее вы сделали свою работу. Пойдите и выпейте мятного джулепа или сладкого чая.

Решение этого проекта представлено в приложении к книге и по адресу **https://www.nostarch.com/impracticalpython/** в файле route_cipher_hacker.py. Вам также понадобится программа perms.py, которая основана на предыдущем практическом проекте.

Сложные проекты

Для сложных проектов решения не предусмотрены.

Кодировщик маршрутного шифра

Неопытному служащему телеграфа союзников нужно зашифровать следующее ниже сообщение, дополненное кодовыми словами (табл. 4.1). Помогите ему, написав программу, которая на входе принимает сообщение и автоматически заменяет

Таблица 4.1. *Кодовые слова*

Batteries	HOUNDS
Vicksburg	ODOR
April	CLAYTON
16	SWEET
Grand	TREE
Gulf	OWL
Forts	BAILEY
River	HICKORY
25	MULTIPLY
29	ADD
Admiral	HERMES
Porter	LANGFORD

кодовые слова, заполняет нижний ряд фиктивными словами и переставляет слова с помощью ключа [–1, 3, –2, 6, 5, –4]. Используйте матрицу 6×7 и составьте собственные фиктивные слова.

> We will run the batteries at Vicksburg the night of April 16 and proceed to Grand Gulf where we will reduce the forts. Be prepared to cross the river on April 25 or 29. Admiral Porter.
>
> *Перевод.* Мы пригоним батареи в Виксбург в ночь на 16 апреля и продолжим путь к заливу Гранд, где мы сократим форты. Будьте готовы пересечь реку 25 или 29 апреля. Адмирал Портер.

Для лексикона кодовых слов из этой таблицы рассмотрите возможность использования словаря Python.

Зигзагообразный многорядный шифр

Напишите версию зигзагообразного шифра, которая использует три ряда (три жерди) вместо двух. Подсказки можно найти на веб-странице

https://en.wikipedia.org/wiki/Rail_fence_cipher.

5

КОДИРОВАНИЕ ШИФРОВ ВРЕМЕН ГРАЖДАНСКОЙ ВОЙНЫ В АНГЛИИ

В 1587 г. Мария, королева Шотландии, потеряла голову из-за клочка бумаги. Пятьдесят пять лет спустя сэр Джон Тревanион, сторонник другого обезглавленного монарха, Карла I, спас свою голову клочком бумаги. В чем же разница? В стеганографии.

Стеганография — это проверенная временем практика сокрытия сообщений настолько хорошо, что об их существовании даже не подозревают. Название основано на греческих словах "скрытое письмо", и весьма буквальный греческий пример состоял в том, чтобы взять используемые для письма покрытые воском деревянные таблички, соскоблить воск, написать на дереве, а затем покрыть табличку новым покрытием из гладкого воска. Современный пример состоит во встраивании сообщения в изображение путем тонкого изменения его цветовых компонентов. Даже простое 8-битное изображение JPEG содержит красок больше, чем может обнаружить человеческий глаз, поэтому без цифровой обработки или фильтрации сообщение, по существу, невидимо.

В этой главе вы будете работать с нулевым шифром[1], который вообще-то шифром не является, а считается стенографическим приемом сокрытия открытого текста внутри других символьных цепочек, состоящих из не шифровального материала. "Нулевой" означает "никакой" (null), поэтому с нулевым шифром вы решаете не шифровать сообщение. Ниже приведен пример нулевого шифра, использующего

[1] Нулевой шифр (null cypher), также именуемый маскировочным шифром или тайнописью, является древней формой шифрования, где открытый текст смешивается с большим количеством нешифрованного материала. См. **https://en.wikipedia.org/wiki/Null_cipher**. — *Прим. перев.*

первую букву в каждом слове (в английском варианте зашифровано словосочетание "null cypher", что означает "нулевой шифр"):

Nice **u**ncles **l**ive **l**onger. **C**ruel, **i**nsensitive **p**eople **h**ave **e**ternal **r**egrets.

Перевод. Хорошие дяденьки живут дольше. Жестокие, бесчувственные людишки испытывают вечные сожаления.

Сначала вы напишете код, который найдет скрытое сообщение, которое спасло сэра Джона, а затем вы выполните гораздо более сложную задачу написания нулевого шифра. Наконец, вы получите возможность написать программу, которая могла бы спасти голову королевы Марии, если бы она использовала ее результат.

Проект 10: шифр Треваниона

В целях защиты своих сообщений королева Мария полагалась на стеганографию и шифрование. Стратегия была разумной, но ее применение было ошибочным. Сама того не ведая, для переправки своих сообщений она полагалась на двойного агента по имени Гилберт Гиффорд. Сначала Гиффорд передал их шпиону королевы Елизаветы, который затем взломал шифр и заменил его поддельным сообщением, соблазнившим Марию изобличить себя. Остальное, как говорится, уже история.

Для Джона Треваниона результат был радужнее. Сэр Джон, выдающийся кавалер, который помогал Карлу I против Оливера Кромвеля в Гражданской войне в Англии, был схвачен и заключен в тюрьму в Колчестерском замке. За день до казни он получил письмо от одного из своих друзей. Письмо не было переправлено контрабандой, а доставлено прямиком в руки его тюремщиков, которые осмотрели его, но не заметили никакого обмана. Прочитав его, сэр Джон попросил немного времени наедине, чтобы помолиться в часовне. Когда тюремщики вернулись за ним, он уже исчез.

Вот сообщение, полученное сэром Джоном (ниже приведен его перевод на русский язык).

Worthie Sir John: Hope, that is the beste comfort of the afflicted, cannot much, I fear me, help you now. That I would saye to you, is this only: if ever I may be able to requite that I do owe you, stand not upon asking me. 'Tis not much I can do: but what I can do, bee you verie sure I wille. I knowe that, if deathe comes, if ordinary men fear it, it frights not you, accounting for it for a high honour, to have such a rewarde of your loyalty. Pray yet that you may be spared this soe bitter, cup. I fear not that you will grudge any sufferings; onlie if bie submission you can turn them away, 'tis the part of a wise man. Tell me, an if you can, to do for you anythinge that you wolde have done. The general goes back on Wednesday. Restinge your servant to command. R.T.

Достойный сэр Джон: Надежда, которая является лучшим утешением для страждущих, боюсь, не много способна помочь Вам сейчас. Вот что я хотел бы сказать Вам: если когда-нибудь я смогу отплатить Вам за то, что я Вам должен, не просите меня об этом. Не так уж много того, что я могу сделать, но то, что я могу сделать, будьте уверены, я сделаю. Я знаю, что на пороге смерти, если обычные люди будут ее бояться, то она не испугает Вас, потому что считается высокой честью получить такую награду за свою верность. И все же помолитесь, чтобы Вы были избавлены от этой

чаши горечи. Я не боюсь, что Вы пожалеете о каких-либо страданиях; только из покорности вы можете их избежать, это часть мудрого человека. Скажите мне, и если сможете, я сделаю для Вас все, что сделали бы Вы. Генерал возвращается в среду. Прислушайтесь к своему слуге. Р.Т.

Как вы, наверное, уже догадались, это, казалось бы, невинное письмо содержит скрытое сообщение, показанное ниже жирным шрифтом:

Worthie Sir John: **H**ope, **th**at is the beste comfort of the afflicted, ca**n**not much, I **f**ear me. he**l**p you now. Th**a**t I would saye to you, is **t**his only: if **e**ver I may be able to requite that I do owe you, st**a**nd not upon asking me. 'T**i**s not much I can do: bu**t** what I can do, be**e** you verie sure I wille. I k**n**owe that, if **d**eathe comes, if **o**rdinary men fear it, it **f**rights not you, ac**c**ounting for it for a high honour, to **h**ave such a rewarde of your loyalty. Pr**a**y yet that you may be spared this soe bitter, cu**p**. I fear not that you will grudge any sufferings; on**l**ie if bie submission you can turn them away, 't**i**s the part of a wise man. Te**l**l me, an **i**f you can, to **d**o for you anythinge that you wolde have done. Th**e** general goes back on Wednesday. Re**s**tinge your servant to command. R.T.

Этот нулевой шифр использует каждую третью букву после знака препинания, сообщая сэру Джону о том, что "panel at east end of chapel slides" (панель в восточном конце часовни сдвигается). Ходят слухи, что позже были обнаружены остатки узкой лестницы в углублении в стене замка. Проход был заблокирован в момент обнаружения, но, возможно, примерно в 1642 г. для сэра Джона это был путь к спасению.

Этот побег в самую последнюю минуту не был бы возможен с помощью традиционного шифра. Только умело замаскировав послание с помощью стеганографии, автор смог так быстро передать его в руки сэра Джона. А прелесть нулевого шифра заключается в том, что даже если сэр Джон не знал схемы, но подозревал, что сообщение присутствует, он смог найти его довольно быстро.

Если бы друг сэра Джона был осторожнее, скрыв зашифрованный текст вместо открытого, то сэр Джон, вероятно, не расшифровал бы сообщение за то короткое время, что у него оставалось — если только он не был бы заранее проинформирован о типе шифра и ключе.

Цель

Написать программный код, который находит в нулевом шифре буквы, скрытые после знаков препинания, и позволяет пользователю выбрать число букв после знака препинания для поиска решения.

Стратегия и псевдокод

Нулевые шифры полагаются на повторяющийся шаблон, известный отправителю и получателю. Например, каждое третье слово может быть частью реального сообщения или, еще лучше, последней буквой каждого третьего слова. В шифре Треваниона это третья буква после знака препинания.

Для того чтобы найти шифр Треваниона, будем считать, что знаки препинания являются сигналом для начала подсчета, а затем напишем программный код, который

находит каждую *n*-ю букву после знака и сохраняет эти буквы в символьной цепочке или списке. Разобравшись в том, как это сделать, вы можете легко отредактировать программный код для работы с любой начальной точкой, такой как каждое заглавное слово, каждая вторая буква в слове или начальная буква каждого третьего слова.

Единственным реальным спорным моментом являются знаки препинания. Например, хотел ли автор нулевого шифра, чтобы знаки препинания были включены в открытый текст? Как обрабатывать второй знак препинания в желаемом диапазоне счета? Что происходит, если два знака препинания встречаются подряд?

Если внимательно посмотреть на шифр Треваниона, то вы должны увидеть, что там есть двойные знаки препинания, вызванные повторным использованием слова tis.

Кроме того, в конце послания имеется путаница знаков препинания, где автор указывает свои инициалы. Для того чтобы справиться с этим, сэр Джон и его друг, возможно, установили некоторые правила до заключения сэра Джона в тюрьму, или сэр Джон просто использовал метод проб и ошибок для их выработки.

Опираясь на концевую часть послания, знаки препинания не включают в число букв. Если бы друг сэра Джона хотел, чтобы они были включены, то скрытое послание заканчивалось бы заглавной буквой T, потому что T находится в трех *символах* после знака препинания, а не в трех *буквах* после. Это означает, что если читатель встречает знак препинания внутри предела счета, то он должен начать свой счет сначала.

Таким образом, вот эти правила:

- начинать счет букв сначала с каждым знаком препинания;
- обнулять счет, если встречается знак препинания;
- знаки препинания не могут быть частью открытого сообщения.

Поскольку вы можете не знать, каким должно быть число букв, напишите программный код так, чтобы он проверял все количества вплоть до предела, которое предоставляет пользователь. Псевдокод будет довольно простым:

```
Загрузить текстовый файл и удалить из него пробелы
Получить от пользователя входные данные о том, сколько букв после знаков препинания надо просматривать
и экзаменовать
Перебрать в цикле число букв от 1 до этого значения просмотра вперед
  Создать пустую символьную цепочку, которая будет содержать перевод
  Создать счетчик
  Создать маркер ❶ первый_найден и установить его равным False
  Перебрать в цикле символы текста
    Если символ является знаком препинания
      Счетчик = 0
      первый_найден = True
    В противном случае, если ❷ первый_наден = True
      Счетчик + 1
```

```
    Если Счетчик = значение просмотра вперед
        Добавить символ в цепочку перевода
    Показать перевод для этого значения просмотра вперед
```

Обратите внимание, что переменная `first_found` (первый найден) ❶ будет оставаться `False` до тех пор, пока не будет обнаружен знак препинания, после чего она будет установлена равной `True` ❷. Это не дает программе вести счет до тех пор, пока не будет найден первый знак препинания.

Теперь вы готовы написать код!

Код шифра Треваниона

Программный код этого раздела отыщет нулевой шифр в стиле шифра Треваниона, закодированный с помощью определенного числа букв после каждого знака препинания. Вам также понадобится текстовый файл, содержащий шифр Треваниона. И скрипт, и текстовый файл можно скачать с **https://www.nostarch.com/impracticalpython/** соответственно в виде файлов null_cipher_finder.py и trevanion.txt. Сохраните эти файлы в одной папке.

Загрузка текста

Листинг 5.1 импортирует несколько полезных модулей и загружает текстовый файл, содержащий нулевой шифр.

Листинг 5.1. Импортирует модули и загружает текст нулевого шифра. Файл null_cipher_finder.py, часть 1

```
❶ import sys
  import string

❷ def load_text(file):
      """Загрузить текстовый файл в виде символьной цепочки."""
    ❸ with open(file) as f:
        ❹ return f.read().strip()
```

Сначала импортируйте теперь уже хорошо знакомый модуль `sys` для обработки исключений, которые могут возникнуть во время ввода пользователем данных ❶. Также импортируйте модуль `string` для получения доступа к полезным коллекциям констант, таких как буквы и знаки препинания.

Далее определите функцию загрузки текстового файла, содержащего нулевой шифр ❷. Эта функция аналогична той, которую вы использовали для загрузки файла словаря в *главе 2*. Она будет вызвана функцией `main()` позже для фактической загрузки файла.

Запустите функцию `load_text()` с помощью инструкции `with` для того, чтобы открыть файл ❸. Используя `with`, вы знаете, что файл будет автоматически закрыт

после его загрузки. Примените метод `read()` для загрузки содержимого и метод `strip()` для удаления начальных и замыкающих пробелов. Обратите внимание, что вы можете сделать это в одной строке кода в инструкции `return` ❹.

Отыскание скрытого сообщения

В листинге 5.2 определяется функция, которая отыскивает скрытое сообщение. Она требует двух аргументов. Первый — это сообщение, которое является исходным текстовым файлом в виде цепочки символов, лишенной пробелов, и второй — это число букв, которые нужно проверять после знака препинания. Это проверочное значение берется у пользователя в рамках функции `main()`.

Листинг 5.2. Отыскивает скрытые буквы. Файл null_cipher_finder.py, часть 2

```
def solve_null_cipher(message, lookahead):
    """Решить нулевой шифр, основываясь на числе букв
       после знака препинания.
    message = текст нулевого шифра как цепочка символов без пробелов
    lookahead = конечная точка диапазона букв после знака препинания
    """
❶ for i in range(1, lookahead + 1):
    ❷ plaintext = ''
      count = 0
      found_first = False
    ❸ for char in message:
      ❹ if char in string.punctuation:
            count = 0
            found_first = True
      ❺ elif found_first is True:
            count += 1
      ❻ if count == i:
            plaintext += char
      ❼ print("Используя сдвиг {} после знака препинания = {}".
              format(i, plaintext))
    print()
```

Рассматривайте значение `lookahead` как конечную точку диапазона в цикле `for`, для того чтобы вы могли проверить все промежуточные буквы сообщения на наличие скрытого послания. Установите диапазон в виде `(1, lookahead + 1)` ❶; благодаря этому вы начнете с первой буквы после знака препинания и включите выбранное пользователем число в оценку.

Теперь задайте несколько переменных ❷. Сначала инициализируйте пустую символьную цепочку для хранения переведенного открытого текста. Затем установите счетчик равным 0. Наконец, установите переменную `found_first` равной `False`. Помните, что программа использует эту переменную для того, чтобы отложить подсчет до тех пор, пока не встретится первый знак препинания.

Затем начните перебирать символы в сообщении ❸. Если вы обнаружите знак препинания, то сбросьте значение счетчика в 0 и установите значение параметра `found_first` равным `True` ❹. Если вы уже нашли знак препинания, а текущий символ не является знаком препинания, то переместите счетчик на 1 ❺. Если вы нашли букву, которую ищете, то это означает, что счетчик достиг текущего значения `lookahead` (`i`) — добавьте букву в символьную цепочку с открытым текстом ❻.

После того как проэкзаменуете все символы в сообщении для текущего значения `lookahead`, покажите текущий ключ и перевод ❼.

Определение функции *main()*

В листинге 5.3 определена функция `main()`. Из *главы 3* вы, возможно, помните, что функция `main()` подобна менеджеру проекта вашей программы: она принимает входные данные, отслеживает продвижение работы и сообщает другим функциям, когда нужно приступать к работе.

Листинг 5.3. Определяет функцию `main()`. Файл null_cipher_finder.py, часть 3

```
def main():
    """Загрузить текст, решить нулевой шифр."""
    # загрузить и обработать сообщение:
❶   filename = input("\nВведите полное имя файла для перевода сообщения: ")
❷   try:
        loaded_message = load_text(filename)
    except IOError as e:
        print("{}. Завершение программы.".format(e), file=sys.stderr)
        sys.exit(1)
❸   print("\nПЕРВОНАЧАЛЬНОЕ СООБЩЕНИЕ =")
    print("{}".format(loaded_message), "\n")
    print("\nСписок проверяемых знаков препинания = {}".
          format(string.punctuation), "\n")

    # удалить пробелы:
❹   message = ''.join(loaded_message.split())

    # получить от пользователя диапазон возможных ключей шифра:
❺   while True:
❻       lookahead = input("\nЧисло букв, проверяемых после \
                          знака препинания: ")
❽       if lookahead.isdigit():
            lookahead = int(lookahead)
            break
        else:
❽           print("Пожалуйства, введите число.", file=sys.stderr)
    print()
```

```
    # выполнить функцию декодирования шифра
❾ solve_null_cipher(message, lookahead)
```

Начните с запроса у пользователя имени файла (имя + расширение) ❶, а затем используйте инструкцию `try` для вызова функции `load_text()` ❷. Если файл не найден, то покажите ошибку красным цветом — для тех, кто использует окно интерпретатора IDLE, — и выйдите из программы с использованием команды `sys.exit(1)`, где 1 указывает на завершение программы с ошибкой.

Напечатайте сообщение, за которым следует список знаков препинания из модуля `string` ❸. Программа распознаёт как знаки препинания только те символы, которые находятся в этом списке.

Далее возьмите загруженное сообщение и удалите все пробелы ❹. Вы будете подсчитывать только буквы и знаки препинания, поэтому пробелы будут просто мешать. Начните цикл `while`, который продолжит запрашивать у пользователя входные данные в случае, если он введет неверное значение ❺. Запросите у пользователя число проверяемых букв после знака препинания ❻. Оно будет рассматриваться как диапазон, начиная с 1 и заканчивая выбранным числом плюс 1. Если входное значение является цифрой ❼, то преобразуйте его в целое число, т. к. инструкция `input` возвращает значение в виде цепочки символов. Затем примените инструкцию `break` для выхода из цикла.

Если пользователь вводит недопустимое значение, например "Bob", то примените инструкцию `print` для запроса числа и сделайте шрифт красным для пользователей интерактивной оболочки с помощью конструкции `sys.stderr` ❽. Цикл `while` затем повторит запрос на ввод данных.

Передайте переменную `lookahead` вместе с сообщением в функцию шифрования `solve_null_` ❾. Теперь осталось только вызвать функцию `main()`.

Выполнение функции *main()*

Вернувшись в глобальное пространство, завершите программный код, вызвав `main()`, но только если программа выполняется в автономном режиме, а не импортируется в другую программу (листинг 5.4).

Листинг 5.4. Вызывает функцию main(). Файл null_cipher_finder.py, часть 4

```
if __name__ == '__main__':
    main()
```

Ниже приведен пример вывода из законченной программы, используя на входе шифр Треваниона:

```
Введите полное имя файла для перевода сообщения: trevanion.txt

ПЕРВОНАЧАЛЬНОЕ СООБЩЕНИЕ =
Worthie Sir John: Hope, that is the beste comfort of the afflicted, cannot
much, I fear me, help you now. That I would saye to you, is this only: if ever
```

```
I may be able to requite that I do owe you, stand not upon asking me. 'Tis not
much I can do: but what I can do, bee you verie sure I wille. I knowe that, if
deathe comes, if ordinary men fear it, it frights not you, accounting for it
for a high honour, to have such a rewarde of your loyalty. Pray yet that you
may be spared this soe bitter, cup. I fear not that you will grudge any
sufferings; onlie if bie submission you can turn them away, 'tis the part of
a wise man. Tell me, an if you can, to do for you anythinge that you wolde have
done. The general goes back on Wednesday. Restinge your servant to command.
R.T.

Список проверяемых знаков препинания = !"#$%&'()*+,-./:;<=>?@[\]Ля~}|{'_

Число букв, проверяемых после знака препинания: 4

Используя сдвиг 1 после знака препинания = HtcIhTiisTbbliiiatPcIotTatTRRT

Используя сдвиг 2 после знака препинания = ohafehsftiuekfftcorufnienohe

Используя сдвиг 3 после знака препинания = panelateastendofchapelslides

Используя сдвиг 4 после знака препинания = etnapthvnnwyoerroayaitlfogt
```

Из полученного результата можно заключить, что программа проверяла до четвертой буквы после знака препинания, но, как хорошо видно, она находит решение с использованием трех букв после знака препинания.

Проект 11: написание нулевого шифра

Вот незаконченный пример очень слабого нулевого шифра, основанного на начале каждого слова. Потратьте минуту и попробуйте закончить предложение:

```
H_________ e_________ l_________ p_________ m_________ e_________.
```

Вам, вероятно, будет трудно, потому что независимо от того, используете ли вы буквы или даже целые слова, потребуется тяжелая работа и время на то, чтобы создать нулевой шифр, который не читается с затруднениями и не вызывает подозрений. Суть проблемы заключается в контексте. Если шифр встроен в переписку, то во избежание подозрений эта переписка должна быть связной. А это означает, что она должна касаться соответствующей темы и оставаться верной этой теме в течение разумного числа предложений. Как вы, наверное, убедились, черновой набросок даже одного предложения на любую тему представляет собой непростую задачу!

Ключ состоит в том, чтобы убедительно избегать контекста, и хороший способ сделать это — применить список. Никто не ожидает, что список покупок будет жестко организован или иметь смысл. Списки также могут быть адаптированы к получателю. Например, переписчики могут обсуждать книги или фильмы и обмениваться списками своих самых любимых из них. Заключенный может начать изучать иностранный язык и получать регулярные списки слов от своего наставника. Предприниматель может получать ежемесячные остатки продукции с одного из своих скла-

дов. Со списками контекст соблюдается даже при перетасовке слов, благодаря чему правильная буква отыскивается в правильном месте.

Цель

Написать программный код, который скрывает нулевой шифр в списке слов.

Код спискового шифра

Программный код list_cipher.py в листинге 5.5 встраивает нулевой шифр внутрь списка слов словаря под предлогом изучения слов. Вам также понадобится программа load_dictionary.py, которую вы использовали в *главах 2* и *3*. Указанный файл можно скачать вместе со следующим ниже скриптом с веб-сайта **https://www.nostarch.com/impracticalpython/**. Наконец, вам понадобится один из файлов словаря, который вы использовали в *главах 2* и *3*. Список подходящих онлайновых словарей можно найти в табл. 2.1. Все вышеупомянутые файлы должны храниться в одной папке.

Листинг 5.5. Скрывает нулевой шифр в списке. Файл list_cipher.py

```
❶ from random import randint
  import string
  import load_dictionary

  # написать короткое сообщение, которое не содержит знаки препинания
  # или числа!
  input_message = "Panel at east end of chapel slides"

  message = ''
  for char in input_message:
   ❷ if char in string.ascii_letters:
          message += char
  print(message, "\n")
❸ message = "".join(message.split())

❹ # открыть файл словаря
  word_list = load_dictionary.load('2of4brif.txt')

  # построить список слов словаря со скрытым сообщением
❺ vocab_list = []
❻ for letter in message:
      size = randint(6, 10)
   ❼ for word in word_list:
          if len(word) == size and word[2].lower() == letter.lower()\
          and word not in vocab_list:
              vocab_list.append(word)
              break
```

```
❽ if len(vocab_list) < len(message):
      print("Список слов слишком мал. Попробуйте более крупный \
            словарь либо более короткое сообщение!")
  else:
      print("Слова словаря для блока 1: \n", *vocab_list, sep="\n")
```

Начните с импорта функции `randint()` из модуля `random` ❶. Она позволяет выполнять (псевдо)случайный отбор целого числа. Затем загрузите модуль `string` для доступа к буквам ASCII. Завершите импортом модуля `load_dictionary`.

Затем напишите короткое секретное сообщение. Обратите внимание, что ассоциированный комментарий запрещает знаки препинания или цифры. Попытка использовать их с содержимым файла словаря будет проблематичной. Поэтому отфильтруйте все, кроме букв, проверив принадлежность к буквам `string.ascii_letters`, которые содержат как прописные, так и строчные буквы ❷:

```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Выведите на экран сообщение, а затем удалите пробел ❸. Загрузите файл словаря ❹ и создайте пустой список для хранения слов словаря ❺.

Примените цикл `for` для перебора всех букв сообщения ❻. Объявите переменную `size` и назначьте ей случайное значение от 6 до 10 с помощью функции `randint()`. Эта переменная будет обеспечивать, чтобы слова были достаточно длинными для того, чтобы быть надежными, как слова словаря. Если хотите, вы можете установить максимальное значение выше.

Вложите еще один цикл `for` и используйте его для обхода словаря слов ❼, сравнивая их длину с переменной `size` и (строчную) букву в индексе 2 — третью букву слова — с текущей (строчной) буквой в цикле сообщения. Значение индекса в слове можно изменить, но обеспечьте, чтобы оно не превышало минимально возможное значение в переменной `size` минус 1! Заключительное сравнение не позволяет использовать одно и то же слово дважды. Если слово все проверки проходит, то добавьте его в `vocab_list` и перейдите к следующей букве сообщения.

Типичный файл словаря должен содержать такое количество слов, которое будет достаточным для шифрования короткого сообщения. Но ради подстраховки примените условное выражение для проверки того, что длина списка `vocab_list` не короче длины сообщения ❽. Если она короче, то у вас закончились слова до достижения конца сообщения, и вам нужно напечатать пользователю предупреждение. В противном случае напечатайте список слов.

Вывод спискового шифра

Вот вывод из приведенного выше кода (для удобства чтения я выделил каждую третью букву, хотя сообщение довольно легко обнаруживается без какой-либо помощи):

```
Panelateastendofchapelslides

Слова словаря для блока 1:

alphabets
abandoning
annals
aberration
ablaze
abandoned
acting
abetted
abasement
abseil
activated
adequately
abnormal
abdomen
abolish
affecting
acceding
abhors
abalone
ampersands
acetylene
allegation
absconds
aileron
acidifying
abdicating
adepts
absent
```

Использование шрифта с единообразной шириной символов и печать слов стопкой в действительности ухудшает шифр. Способы решения этих проблем мы рассмотрим в *разд. "Спасение королевы Марии" далее в этой главе.*

Резюме

В этой главе вы написали программу, которая раскрывает сообщение, скрытое в нулевом шифре по типу Треваниона. Затем вы написали вторую программу, которая генерирует нулевой шифр и скрывает его в учебном списке слов. В следующих далее практических проектах вы сможете разведать способы сделать этот списковый шифр еще безопаснее.

Дальнейшее чтение

Более подробную информацию о Марии, королеве Шотландии и сэре Джоне Тревanионе можно найти в книге Гэри Блэквуда "Таинственные сообщения: история кодов и шифров" (Blackwood G. Mysterious messages: a history of codes and ciphers. The Penguin Group, 2009) и книге Саймона Сингха "Книга кодов: наука секретаря от древнего Египта до квантовой криптографии " (Singh S. The code book: the science of secretary from ancient egypt to quantum cryptography. Anchor, 2000).

Практические проекты

Теперь, когда вы являетесь экспертом по нулевому шифру, посмотрите, сможете ли вы изменить судьбу Марии, королевы Шотландии, а затем украдкой взглянуть на самую секретную переписку сэра Джона.

Спасение королевы Марии

Лучшая часть процесса написания программного кода относится к размышлениям о задачах и способах их решения. Давайте вернемся к печальному случаю с Марией, королевы Шотландии. Вот что мы знаем.

- Марии не разрешалось переписываться, поэтому письма приходилось проносить тайно. Это означает, что предатель Гилберт Гиффорд не может быть исключен из уравнения. Гиффорд был единственным человеком, которого Мария знала и который мог доставлять ей почту.
- Мария и те, с кем она переписывалась, слишком доверяли ненадежному шифру и поэтому говорили слишком свободно. Имея меньше уверенности, они могли бы проявить больше сдержанности.
- Тюремщики Марии, имея в своем распоряжении очевидный шифр, посчитали, что он содержит компрометирующие материалы, и продолжали работать над ним до тех пор, пока не нашли его.

Гиффорд, двойной агент, не был посвящен в детали шифров, которыми пользовалась Мария. Теперь допустим, что Мария использовала нулевой шифр. Если переписка была несколько крамольной — хотя и не предательской, — то ее похитители могли бы не заметить послание. В случае беглого осмотра было бы достаточно использовать переменный шаблон, для того чтобы загнать криптоаналитиков в тупик.

Как вы уже видели, нулевой шифр проще скрыть в списке, чем в письме. Список семей, поддерживающих Марию, мог бы послужить этой цели. Это могли бы быть известные сторонники или, в макиавеллиевском стиле, сочетание друзей и врагов! Это послание не было бы откровенно крамольным, но было бы достаточно близким к этому, благодаря чему отсутствие шифрования предполагало бы, что никакая форма шифрования не использовалась вообще.

В этом практическом проекте напишите программу, которая вставляет в список фамилий сообщение "Give your word and we rise" (дайте сигнал, и мы поднимемся). Для того чтобы скрыть буквы в сообщении, начните со второй буквы во втором

имени, перейдите к третьей букве в третьем имени и затем продолжайте чередовать вторую и третью буквы для оставшихся слов.

В дополнение к неиспользуемому имени включите "Stuart" (Стюарт) и "Jacob" (Яков) в качестве нулевых (null) слов в начальной части списка, для того чтобы помочь скрыть присутствие шифра. Не вставляйте в эти нулевые имена буквы из шифра и полностью игнорируйте их при выборе позиции буквы для шифра в следующем слове; если вторая буква использовалась в слове перед нулевым именем, то используйте третью букву в слове после нулевого имени. Нулевой шифр будет занимать следующие ниже буквы, выделенные полужирным шрифтом (расположение нулевых слов зависит от вас, но не давайте им влиять на шаблон):

First S**e**cond Th**i**rd STUART F**o**urth Fi**f**th JACOB Si**x**th Se**v**enth Ei**g**hth

Программа может напечатать список как по вертикали, так и по горизонтали. Список имен должен быть достоверно предварен коротким сообщением, но это сообщение не должно быть частью шифра.

Список имен можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/** как supporters.txt и загрузить как стандартный файл словаря. Решение можно найти в приложении к книге и на веб-сайте в виде файла save_Mary_practice.py.

Колчестерская поимка

Вместо какого-нибудь болвана, накаченного элем, ответственным за заключенного Джона Треваниона остаетесь вы, когда в замок Колчестер приходит следующее письмо:

> Sir John: Odd and too hard, your lot. Still, we will band together and, like you, persevere. Who else could love their enemies, stand firm when all others fail, hate and despair? While we all can, let us feel hope. — R.T.
>
> *Перевод.* Сэр Джон: странна и слишком тяжела Ваша судьба. Тем не менее мы объединимся и, как и Вы, будем упорствовать. Кто еще мог любить своих врагов, твердо стоять на своем, когда все остальные терпят неудачу, ненавидят и отчаиваются? До тех пор, пока мы все способны, давайте ощущать надежду. — Р.Т.

Даже для XVII в. формулировка письма выглядит неуклюжей, и вы решаете изучить его повнимательнее перед тем, как передать его своему заключенному.

Напишите программу Python, которая принимает входные данные, n, и проверяет и показывает нулевой шифр на основе n-й буквы после начала каждого n-го слова. Например, число 2 на входе выделит буквы в следующем ниже послании:

So, t**h**e cold t**e**a didn't p**l**ease the o**l**d finicky w**o**man.

Перевод. Холодный чай не понравился старой привередливой женщине.

Текстовый файл с сообщением можно скачать с **https://nostarch.com/impracticalpython/** в виде файла colchester_message.txt. Решение можно найти в приложении к книге и на веб-сайте в виде файла colchester_practice.py. Сохраните файлы с текстом и программным кодом Python вместе в одной папке.

6

НАПИСАНИЕ ТЕКСТА НЕВИДИМЫМИ ЧЕРНИЛАМИ

Осенью 2012 года в телевизионной сети CBS дебютировала криминальная драма "Элементарно" (Elementary). Она явилась переосмыслением мифов о Шерлоке Холмсе, осуществленном в XXI веке в Нью-Йорке. В указанном сериале в главной роли Шерлока снялся Джонни Ли Миллер (Jonny Lee Miller), и в качестве его напарника, доктора Джоан Ватсон, снялась Люси Лю (Lucy Liu). В эпизоде 2016 г. ("У тебя есть я, у кого есть ты?") Морланд Холмс, ушедший из семьи отец Шерлока, нанимает Джоан для того, чтобы найти в своей организации "крота". Она быстро решает дело, обнаружив в электронном письме шифр Виженера. Однако некоторые поклонники сериала были недовольны: шифр Виженера вряд ли можно считать едва уловимым, и как это такой умный человек, как Морланд Холмс, не смог найти его самостоятельно?

В этом проекте вы разрешите эту дилемму с помощью стеганографии, но не с нулевым шифром, как в *главе 5*. Для того чтобы скрыть это сообщение, вы будете использовать сторонний модуль под названием `python-docx`, который позволит вам скрыть текст, манипулируя документами Microsoft Word непосредственно с помощью Python.

Проект 12: сокрытие шифра Виженера

В первом эпизоде китайские инвесторы нанимают консалтинговую компанию Морланда Холмса для переговоров с колумбийским правительством о лицензиях на добычу нефти и правах на бурение. Прошел год, и в последний момент врывается конкурент и заключает сделку, оставляя китайских инвесторов ни с чем. Морланд

подозревает предательство со стороны одного из своих сотрудников и просит Джоан Ватсон провести расследование в одиночку. Джоан вычисляет "крота", найдя шифр Виженера в одном из его писем.

ВНИМАНИЕ, СПОЙЛЕР!

Расшифрованное содержимое шифра ни разу не упоминается, и "крота" убивают в следующем эпизоде.

Шифр Виженера, также именуемый нераскрываемым шифром, пожалуй, является самым известным шифром всех времен. Изобретенный в XVI в. французским ученым Блезом де Виженером, этот шифр является полиалфавитным подстановочным шифром, который в наиболее распространенной версии использует одно ключевое слово. Это ключевое слово, такое как BAGGINS, многократно печатается поверх открытого текста, как в сообщении, показанном на рис. 6.1.

B	A	G	G	I	N	S	B	A	G	G	I	N	S	B	A	G	G	I
s	p	e	a	k	f	r	i	e	n	d	a	n	d	e	n	t	e	r

Рис. 6.1. Текстовое сообщение с ключевым словом BAGGINS шифра Виженера, печатаемого сверху

Затем для шифрования сообщения используется алфавитная таблица. На рис. 6.2 приведен пример первых пяти строк таблицы Виженера. Обратите внимание, как с каждой новой строкой алфавит сдвигается влево на одну букву.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D

Рис. 6.2. Часть таблицы Виженера

Буква ключевого слова над буквой открытого текста определяет, какую строку следует использовать для шифровки. Например, обратите внимание, что для шифровки буквы *s* в слове *speak* буквой ключевого слова над ней является *B*. Спуститесь в строку *B* и прочитайте поперек до места, где открытый текст *s* находится в верхней части столбца. Для буквы шифротекста используйте букву *T* на пересечении.

На рис. 6.3 показан пример полного сообщения, зашифрованного шифром Виженера. Этот вид текста наверняка привлек бы внимание и стал бы объектом пристального внимания, если бы он был виден в документе!

```
TPKGS SJJETJ IAV FNZKZ
```

Рис. 6.3. Сообщение, зашифрованное шифром Виженера

Шифр Виженера оставался невзломанным вплоть до середины XIX в., когда Чарльз Бэббидж, изобретатель предвестника компьютера, понял, что короткое ключевое слово, используемое с длинным сообщением, приведет к повторяющимся шаблонам, которые способны выявить длину ключевого слова и в конечном счете сам ключ. Взлом шифра был огромным ударом для профессиональной криптографии, и в Викторианскую эпоху оригинального Холмса и Ватсона никаких значительных достижений в этой области сделано не было.

Именно наличие этого шифра как раз и вызывает к эпизоду в сериале "Элементарно" вопросы "с оттенком недоверия". Зачем привлекается внешний консультант для отыскания такого явно подозрительного письма? Давайте посмотрим, сможем ли мы придумать правдоподобное объяснение с помощью языка Python.

Цель

Предположим, что вы являетесь корпоративным "кротом" из эпизода сериала и используете язык Python для сокрытия секретного сообщения, подытоживающего сведения из официального текстового документа о сделке. Начать с незашифрованного сообщения и закончить зашифрованной версией.

Платформа

Ваша программа должна работать с повсеместно распространенным программным обеспечением для обработки текстов, т. к. выход должен совместно использоваться между различными корпорациями. Это подразумевает использование пакета Microsoft Office для Windows или совместимых версий для macOS или Linux. И ограничение вывода в стандартный документ Word делает вопросы, связанные с аппаратным обеспечением, ответственностью компании Microsoft!

Соответственно, этот проект был разработан с помощью Word 2016 для Windows, а результаты проверены с помощью Word для Mac v16.16.2. Если у вас нет лицензии на Word, то вы можете использовать бесплатное приложение Microsoft Office Online, доступное по адресу **https://products.qffice.com/en-us/qffice-online**.

Если в настоящее время вы используете альтернативы Word, такие как LibreOffice Writer или OpenOffice Writer, то вы можете открыть и просмотреть файлы Word (.docx), используемые и созданные в этом проекте; однако скрытое сообщение, скорее всего, будет поставлено под угрозу, как описано в *разд. "Обнаружение скрытого сообщения" далее в этой главе*.

Стратегия

Вы — бухгалтер с начальным знанием языка Python, и вы работаете на очень умного и подозрительного человека. Проект, над которым вы работаете, является очень частным, с элементами управления — такими как фильтры электронной почты — для поддержания конфиденциальности. И если вам удастся умыкнуть сообщение, то непременно последует тщательное расследование. Поэтому вам нужно скрыть явно подозрительное сообщение в электронном письме, либо напрямую, либо

в виде вложения, но избежать первоначального обнаружения и последующего внутреннего аудита.

Вот некоторые ограничения:

- вы не можете отправить сообщение конкурирующей корпорации напрямую, только посреднику;
- вам нужно беспорядочно перемешать (проскремблировать) сообщение так, чтобы избежать фильтров электронной почты, которые будут искать ключевые слова;
- вам нужно скрыть зашифрованное сообщение от посторонних глаз для того, чтобы не вызвать подозрений.

Посредника было бы легко организовать, а бесплатные сайты шифрования легко найти в Интернете, но последний пункт будет проблематичнее.

Стеганография дает ответ, но, как вы видели в предыдущей главе, скрыть даже короткое сообщение в нулевом шифре — задача не из легких. Альтернативные методы включают смещение строк текста по вертикали или слов по горизонтали на небольшие величины, изменение длины букв или использование метаданных файла — но вы является бухгалтером с ограниченными знаниями языка Python и еще меньшим количеством времени. Если бы только был простой способ, подобно невидимым чернилам в былые времена.

Создание невидимых чернил

Работа с невидимыми чернилами в наш век электронных чернил, пожалуй, выглядела бы просто как сумасшествие! Невидимый шрифт легко парировал бы тщательный визуальный осмотр онлайновых документов и даже не существовал бы в бумажных распечатках. Поскольку содержимое будет зашифровано, цифровые фильтры, ищущие ключевые слова, такие как "сделка" (bid) или испанские названия добывающих нефтяных бассейнов, ничего не найдут. Но лучшее состоит в том, что невидимые чернила просты в использовании — нужно просто установить цвет текста в цвет фона.

Форматирование текста и изменение его цвета требует текстового процессора, такого как Microsoft Word. Для того чтобы сделать невидимые электронные чернила в Word, вам просто нужно выбрать символ, слово или строку текста и сделать цвет шрифта белым. Получатель сообщения должен будет выбрать весь документ и использовать инструмент выделения (рис. 6.4) для закрашивания выделенного текста в черный цвет, таким образом скрывая стандартные черные буквы и выводя скрытые белые буквы в поле зрения.

Простое выделение документа в Word не обнаруживает белый текст (рис. 6.5), поэтому кто-то должен быть очень подозрительным, чтобы найти эти скрытые сообщения.

Конечно же, вы можете выполнить все это в текстовом процессоре в одиночку, но есть два случая, когда Python-подход будет предпочтительнее: 1) когда вам нужно зашифровать длинное сообщение и вы не хотите вставлять и скрывать все строки

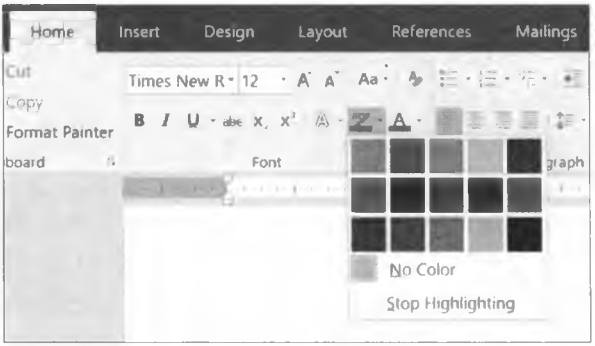


Рис. 6.4. Инструмент выделения цвета текста в Word 2016

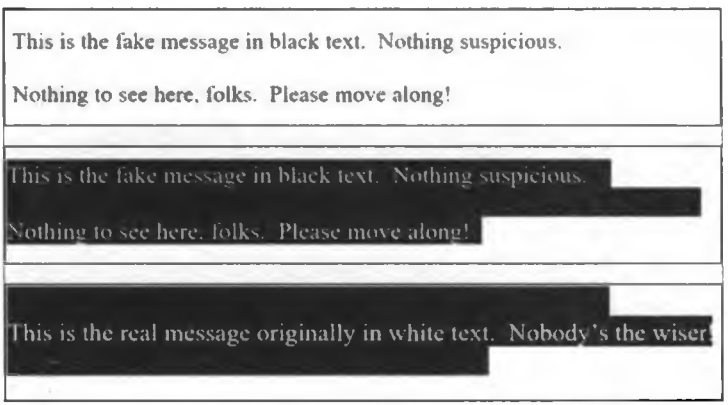


Рис. 6.5. Вверху — часть документа Word с видимым фальшивым сообщением; в середине — документ, выбранный с помощью сочетания клавиш <Ctrl>+<A>; внизу — реальное сообщение, показанное с помощью инструмента выделения и сделанного черным цветом выделения

вручную; 2) когда вам нужно отправить больше чем несколько сообщений. Как вы увидите, короткая программа на Python значительно упростит этот процесс!

Учет типов шрифтов, кернинга и межзнакового интервала

Размещение невидимого текста — ключевое проектное решение. Одним из вариантов является использование пробелов между видимыми словами фальшивого сообщения, но это может вызвать проблемы, связанные с интервалами, которые сделают конечный продукт подозрительным.

Пропорциональные шрифты используют переменную ширину символов для улучшения читаемости. Примером таких шрифтов являются Arial и Times New Roman. *Моноширинные шрифты* характеризуются постоянной шириной символов, что способствует выравниванию текста и распознаванию отдельных символов, в особенности тонких, таких как символы (или {. В результате моноширинные шрифты пользуются популярностью в интерфейсах программирования. Примером таких шрифтов являются `Consolas` и `Courier New`.

Кернинг — это типографская процедура по регулировке расстояния и наложения между отдельными глифами[1] символов с целью улучшения их визуальной привлекательности. Процедура, именуемая *настройкой межзнакового интервала* (или *разрядкой/уплотнением*, или *трекингом*), используется для настройки интервала между символами во всех строках или блоках текста с той же целью. Эти настройки способствуют различимости и удобочитаемости, обеспечивая, чтобы буквы были расположены не настолько близко друг к другу, что они становятся неразличимыми, или не настолько далеко друг от друга, что слова не распознаются. Обратите внимание, что все мы читаем слова, а не буквы. Если сомневаетесь в этом, то прочтите вот это: peopl raed wrds nt lttrs. Corase, contxt hlps[2].

Сначала настраивается кернинг между парами букв, а затем межзнаковый интервал, в ходе которого сохраняется относительный кернинг пар букв. Как упоминалось ранее, эти переменные ширины и автоматические исправления могут вызвать проблемы при попытке скрыть символы между словами, использующими пропорциональные шрифты:

To a great mind nothing is little.	Пропорциональный шрифт без скрытых букв
To a great mind nothing is little.	Пропорциональный шрифт со скрытыми буквами между словами
To3a3great2mind2nothingKis little.	Скрытые буквы обнаружены
To a great mind nothing is little.	Моноширинный шрифт без скрытых букв
To a great mind nothing is little.	Моноширинный шрифт со скрытыми буквами между словами
To3a3great2mind2nothingKis little.	Скрытые буквы обнаружены

Если вы используете моноширинный шрифт, то единообразный интервал обеспечивает удобное укрытие. Но поскольку в профессиональной переписке чаще применяются пропорциональные шрифты, то техника невидимых чернил должна фокусироваться на более легко контролируемых промежутках между строками.

Использование пустых строк между абзацами — самый простой способ программирования и чтения, и он не должен требовать длинного фальшивого сообщения, в результате чего вы можете сжато резюмировать основные моменты сделки. Это важно, т. к. вы не хотите, чтобы в ваше видимое фальшивое сообщение добавлялись пустые страницы. Следовательно, след вашего скрытого сообщения должен быть меньше, чем вашего фальшивого.

Как избежать проблем

Когда вы разрабатываете программное обеспечение, неплохо периодически задаваться вопросом о том, как пользователь может все это испортить. Единственное,

---

[1] В типографике глиф — это характерная особенность шрифта. В вычислительной технике это элемент компьютерного символа, соответствующий графеме или графемоподобной единице текста: это может быть буква, число, знак пунктуации или пиктограмма, декоративный символ, графическая метка. См. **https://ru.wikipedia.org/wiki/Глиф**. — *Прим. перев.*

[2] Перевод: People read words, not letters, because context helps — люди читают слова, а не буквы, потому что помогает контекст. — *Прим. перев.*

что здесь может пойти не так, — это то, что шифровальный процесс изменит буквы в вашем скрытом сообщении, в результате чего настройки кернинга и межзнакового интервала могут подтолкнуть слово выйти за пределы разрыва строки, что приведет к автоматическому переносу строки. В результате в фальшивом сообщении получится неравномерное и подозрительно выглядящее пространство между абзацами. Один из способов избежать этого — нажимать клавишу <Enter> немного раньше, когда вы печатаете в каждой строке реального сообщения. Это оставит некоторое пространство в конце строки для размещения изменений из-за шифрования. Конечно, вам все равно придется проверить результаты. Заранее считать, что программный код работает, так же рискованно, как допустить, что Джеймс Бонд мертв!

Управление документами Word с помощью *python-docx*

Бесплатный сторонний модуль под названием `python-docx` позволяет Python управлять файлами Microsoft Word (.docx). Для скачивания и установки сторонних модулей, упомянутых в этой книге, вы будете использовать программу-инсталлятор pip, т. е. систему управления пакетами, которая упрощает установку программного обеспечения на основе Python. Python 3 в Windows и macOS версии 3.4 и более поздних поставляется с предустановленным инсталлятором pip; для Python 2 предустановка pip начинается с версии 2.7.9. Пользователям Linux может потребоваться установить pip отдельно. Если вы обнаружите, что вам нужно установить либо обновить pip, обратитесь к инструкции по адресу **https://pip.pypa.io/en/stable/installing/** либо поищите в Интернете на предмет установки pip в вашей конкретной операционной системе.

С помощью инструмента pip установить `python-docx` можно, просто выполнив команду **`pip install python-docx`** в интерактивном окне PowerShell либо окне терминала, в зависимости от вашей операционной системы. Онлайновые инструкции для `python-docx` доступны по адресу **https://python-docx.readthedocs.io/en/latest/**.

В этом проекте вам нужно усвоить объекты `paragraph` и `run`. Модуль `python-docx` организует типы данных, используя три объекта в следующей иерархии:

- `document` — весь документ со списком объектов `paragraph`;
- `paragraph` — блок текста, разделенный с использованием клавиши <Enter> в Word; содержит список объектов `run`;
- `run` — связная символьная цепочка с текстом одинакового стиля.

Абзац `paragraph` считается объектом блочного уровня, который `python-docx` определяет следующим образом: "элемент блочного уровня размещает содержащийся в нем текст между его левым и правым краями, добавляя дополнительную строку всякий раз, когда текст выходит за его правую границу. Для объекта `paragraph` границы обычно являются полями страницы, но они также могут быть границами столбцов, если страница выложена в столбцах, либо границами ячеек, если `paragraph` находится внутри ячейки таблицы. Таблица также является объектом блочного уровня".

Объект `paragraph` имеет разнообразные свойства, которые определяют его размещение в контейнере — обычно это страница — и способ разделения его содержимого на отдельные строки. Доступ к свойствам форматирования объекта `paragraph` можно получить с помощью объекта `ParagraphFormat`, доступного через свойство `ParagraphFormat` объекта `paragraph`, и можно задать все свойства `paragraph` с помощью группировки стилей абзаца либо применить их непосредственно к объекту `paragraph`.

Объект `run` — это объект внутристрочного уровня, который появляется в пределах абзацев или других объектов блочного уровня. Объект `run` имеет свойство `font` только для чтения, обеспечивающее доступ к объекту `font`. Объект `font` предоставляет свойства для получения и настройки форматирования символов для конкретного объекта `run`. Этот функционал вам понадобится для установки белого цвета текста вашего скрытого сообщения.

Стиль относится к коллекции атрибутов Word для абзацев и символов (объектов `run`) или их комбинации. Стиль включает в себя такие знакомые всем атрибуты, как шрифт, цвет, отступ, межстрочный интервал и т. д. Возможно, вы заметили, что некоторые из них отображены в Word на группе **Styles** (Стили) на вкладке **Home** (Главная) ленты (рис. 6.6). Любое изменение стиля — даже одной буквы — требует создания нового объекта `run`. В настоящее время доступны только те стили, которые находятся в открытом файле .docx. Это может измениться в будущих версиях `python-docx`.

Рис. 6.6. Панель стилей в Microsoft Word 2016

Полную документацию по использованию стилей в `python-docx` можно найти по адресу **http://python-docx.readthedocs.io/en/latest/user/styles-using.html**.

Вот пример абзацев и фрагментов текста так, как их видит `python-docx`:

> Я — один абзац из одного фрагмента, потому что весь мой текст имеет одинаковый стиль.
>
> Я — один абзац с двумя фрагментами. **Я — второй фрагмента, потому что мой стиль изменен на полужирный.**
>
> Я — один абзац с тремя фрагментами. **Я — второй фрагмент, потому что мой стиль изменен на полужирный. Третий фрагмент является моим последним** словом.

Если что-то из этого кажется неясным, не волнуйтесь. Знать `python-docx` в деталях не потребуется. Как и в любом фрагменте программного кода, вам главным образом нужно знать только то, что вы хотите сделать. Поиск в Интернете обязательно даст много полезных предложений и всеобъемлющих образцов программного кода.

ПРИМЕЧАНИЕ

Для того чтобы все работало гладко, не меняйте стили в реальном (скрытом) сообщении и обеспечьте, чтобы каждая строка заканчивалась жестким переводом строки, вручную нажимая клавишу <Enter>. К сожалению, Word не имеет специального символа для мягких переводов строк, вызываемых автоматическим переносом слов. Так что вы не сможете войти в существующий документ Word, применить автоматические разрывы строк и использовать функционал "Найти и заменить", для того чтобы изменить их все на жесткие переводы строк. Такова жизнь "крота".

Скачивание ресурсных файлов

Внешние файлы, которые вам понадобятся, можно скачать с **https://www.nostarch.com/impracticalpython/** и должны быть сохранены в той же папке, что и код:

- template.docx — пустой документ Word, отформатированный с использованием стилей, шрифтов и полей, официально используемых в корпорации Холмса;
- fakeMessage.docx — фальшивое сообщение без фирменного бланка и даты в документе Word;
- realMessage.docx — реальное сообщение в открытом тексте без фирменного бланка и даты в документе Word;
- realMessage_Vig.docx — реальное сообщение, зашифрованное с помощью шифра Виженера;
- example_template_prep.docx — пример фальшивого сообщения, используемого для создания шаблонного документа (программа не требует запуска этого файла).

ПРИМЕЧАНИЕ

Если вы используете Word 2016, то простой способ сделать пустой файл шаблона — написать фальшивое сообщение (включая фирменный бланк) и сохранить файл. Затем удалить весь текст и сохранить файл снова под другим именем. Когда вы назначаете переменной этот пустой файл, работая с `python-docx`, все существующие стили будут сохранены. Разумеется, вы можете использовать файл шаблона с уже включенным фирменным бланком, но для того, чтобы узнать о `python-docx` побольше, мы построим фирменный бланк тут, используя Python.

Возьмите паузу на то, чтобы просмотреть в Word первые четыре документа. Эти файлы содержат входные данные для программы elementary_ink.py. Фальшивое и реальное сообщения — второй и третий перечисленные выше пункты — также показаны на рис. 6.7 и 6.8.

Обратите внимание, что реальное сообщение содержит несколько цифр и специальные символы. Они не будут зашифрованы с помощью таблицы Виженера, которую мы будем использовать, и я включил их для того, чтобы это подчеркнуть. В идеале ради максимальной секретности они будут прописаны (например, "три" вместо 3 и "процент" вместо %), когда позже мы добавим шифр Виженера.

Dear Mr. Gerard:

I received your CV on Monday. It is very impressive, but I am sorry to inform you that Mr. Holmes is not looking for additional staff at this time.

While we do not normally accept unsolicited applications, I will keep your CV on file for future consideration. If it is convenient, please send me a list of references, especially those pertaining to skills in negotiation, accounting, and data mining (preferably using the Python programming language). A recent photograph is also recommended.

Best of luck to you. Feel free to check back at this time next year in the event a position becomes available. Use this email address, and include your name and the word "check-back" in the subject line.

Sincerely yours,

Emil Kurtz
Associate Director
International Affairs

Уважаемый Г-н Жерар

Я получил ваше резюме в понедельник. Оно весьма впечатляет, но я с сожалением сообщаю вам, что мистер Холмс в настоящее время не ищет дополнительный персонал.

Хотя мы обычно не принимаем не запрошенные заявки, я сохраню ваше резюме в файле для дальнейшего рассмотрения. Если это удобно, пожалуйста, пришлите мне список рекомендаций, в особенности тех, которые касаются навыков ведения переговоров, бухгалтерского учета и глубинного анализа данных (желательно с использованием языка программирования Python). Также рекомендуется сделать свежую фотографию. Желаю Вам удачи.

Не стесняйтесь обратиться к нам снова в это время в следующем году в случае, если позиция станет доступной. Используйте этот адрес электронной почты и укажите свое имя и слово "Повторное обращение" в строке темы.

С искренним уважением,

Эмиль Курц
заместитель директора
по международным делам

Рис. 6.7. "Фальшивый" текст в файле fakeMessage.docx

The Colombian deal will be for 2 new venture wildcat wells, one each in the Llanos & Magdalena Basins. These wells include a carry of thirty percent for the national oil company and will test at least 3 K meters of vertical section. In return, the client will be permitted to drill ten wells in the productive Putumayo province, earning a sixty % interest with a fifty percent royalty rate, increasing to the standard eighty five percent royalty five years after start of production in each well.

Колумбийская сделка будет заключена в отношении 2 новых венчурных поисково-разведочных скважин, по одной в бассейнах Льянос и Магдалена. Эти скважины включают в себя перенос тридцати процентов для национальной нефтяной компании и будет испытывать как минимум 3К метров вертикального участка. В свою очередь, клиенту будет разрешено пробурить десять скважин в продуктивной провинции Путумайо, зарабатывая шестьдесят % прибыли с пятьюдесятью процентной ставкой роялти в начале и стандартными восьмидесяти пятью процентами роялти через пять лет после начала добычи в каждой скважине.

Рис. 6.8. Реальное сообщение в файле realMessage.docx

Псевдокод

В следующем ниже псевдокоде описывается загрузка двух сообщений и шаблонного документа, чередование и сокрытие реального сообщения в пустых строках с помощью белого шрифта, а затем сохранение гибридного сообщения.

```
Создать активы:
В Word создать пустой документ с требуемым форматированием/стилями (шаблон)
В Word создать безобидное фальшивое сообщение, которое будет видно и иметь достаточно число
пустых строк для хранения реального сообщения
В Word создать реальное сообщение, которое будет скрыто
Импортировать docx для того, чтобы получить возможность оперировать документами Word с помощью Python
Использовать docx для загрузки фальшивого и реального сообщений в виде списков
Использовать docx для назначения переменной пустого документа
Использовать docx для добавления заголовка фирменного бланка в пустой документ
Создать переменную-счетчик для строк в реальном сообщении
Определить функцию для форматирования межабзацного интервала с помощью docx
Для строки в фальшивом сообщении:
  Если строка пуста, и в реальном сообщении все еще есть строки:
    Использовать docx и счетчик для заполнения пустой строки из реального сообщения
    Использовать docx для окраски шрифта реального сообщения белым цветом
    Нарастить счетчик для реального сообщения
  В противном случае:
    Использовать docx для написания фальшивой строки
  Выполнить функцию настройки межабзацного интервала
Использовать docx для сохранения конечного документа Word
```

Код

Программа elementary_ink.py в листинге 6.1 загружает реальное сообщение, фальшивое сообщение и пустой шаблонный документ. Она скрывает реальное сообщение в пустых строках фальшивого сообщения с помощью белого шрифта, а затем сохраняет гибридное сообщение как безобидный и профессионально выглядящий фрагмент корреспонденции, который может быть прикреплен к электронной почте. Соответствующий программный код можно скачать с веб-сайта

https://www.nostarch.com/impracticalpython/.

Импорт модуля *python-docx*, создание списков и добавление фирменного бланка

Листинг 6.1 импортирует `python-docx`, превращает строки текста в фальшивом и реальном сообщениях в элементы списка, загружает шаблонный документ, задающий стили, и добавляет фирменный бланк.

Листинг 6.1. Импортирует python-docx, загружает важные файлы .docx и добавляет фирменный бланк. Файл elementary_ink.py, часть 1

```
import docx

❶ from docx.shared import RGBColor, Pt
❺ # получить текст из фальшивого сообщения и сделать
  # каждую строку элементом списка
  fake_text = docx.Document('fakeMessage.docx')
  fake_list = []
  for paragraph in fake_text.paragraphs:
      fake_list.append(paragraph.text)
❷ # получить текст из реального сообщения и сделать
  # каждую строку элементом списка
  real_text = docx.Document('realMessage.docx')
  real_list = []
  for paragraph in real_text.paragraphs:
   ❹ if len(paragraph.text) != 0: # удалить пустые строки
         real_list.append(paragraph.text)

❺ # загрузить шаблон, который устанавливает стиль, шрифт, поля и пр.
  doc = docx.Document('template.docx')

❻ # добавить фирменный бланк
  doc.add_heading('Морланд Холмс', 0)
  subtitle = doc.add_heading('Global Consulting & Negotiations', 1)
  subtitle.alignment = 1
  doc.add_heading('', 1)
❼ doc.add_paragraph('17 декабря 2018')
  doc.add_paragraph('')
```

После импорта модуля `docx` — не как "python-docx" — используйте `docx.shared` для получения доступа к объектам цвета (`RGBColor`) и длины (`Pt`) в модуле `docx` ❶. Это позволит вам изменять цвет шрифта и устанавливать интервал между строками. Следующие два блока кода загружают документы Word с фальшивым ❺ и реальным ❷ сообщениями в виде списков. Место, где в каждом документе Word была нажата клавиша <Enter>, определяет, какие элементы будут в этих списках. Для того чтобы скрыть реальное сообщение, удалите все пустые строки, в результате чего ваше сообщение станет максимально коротким ❹. Теперь можно использовать индексы списков для слияния двух сообщений и отслеживания того, какой является каким.

Затем загрузите шаблонный документ, содержащий предустановленные стили, шрифты и поля ❺. Модуль `docx` будет писать в эту переменную и в конечном итоге сохранит ее в качестве окончательного документа.

После загрузки и подготовки входных данных отформатируйте фирменный бланк окончательного документа в соответствии с официальным стилем, принятым

в корпорации Холмса ❶. Функция `add_heading()` добавляет абзац в стиле заголовка с текстовыми и целочисленными аргументами. Целое число `0` обозначает заголовок самого высокого уровня или заголовочный стиль, унаследованный у шаблонного документа. Подзаголовок отформатирован с помощью `1`, т. е. следующего имеющегося заголовочного стиля, и выровнен по центру, опять-таки с целым числом `1` (`0` — левое выравнивание, `2` — правое выравнивание). Обратите внимание, что при добавлении даты указывать целое число не требуется ❷. Если аргумент не указан, то по умолчанию форматирование наследуется у существующей стилевой иерархии, которая в шаблоне выравнивается по левому краю. Другие инструкции в этом блоке кода просто добавляют пустые строки.

Форматирование и чередование сообщений

Листинг 6.2 выполняет реальную работу, форматируя интервал между строками и чередуя сообщения.

Листинг 6.2. Форматирование абзацев и чередование строк фальшивого и реального сообщений. Файл elementary_ink.py, часть 2

```
❹ def set_spacing(paragraph):
      """Использовать docx для установки строчного интервала
         между абзацами."""
      paragraph_format = paragraph.paragraph_format
      paragraph_format.space_before = Pt(0)
      paragraph_format.space_after = Pt(0)

❸ length_real = len(real_list)
  count_real = 0 # индекс текущей строки в реальном (скрытом) сообщении

  # чередовать строки реального и фальшивого сообщений
  for line in fake_list:
   ❹ if count_real < length_real and line == "":
       ❸ paragraph = doc.add_paragraph(real_list[count_real])
       ❺ paragraph_index = len(doc.paragraphs) - 1

         # сделать цвет реального сообщения белым
         run = doc.paragraphs[paragraph_index].runs[0]
         font = run.font
       ❻ font.color.rgb = RGBColor(255, 255, 255) # для проверки
                                                  # сделать красным
       ❼ count_real += 1
      else:
       ❸ paragraph = doc.add_paragraph(line)

    ❹ set_spacing(paragraph)

❿ doc.save('ciphertext_message_letterhead.docx')

  print("Done")
```

Определите функцию, которая форматирует интервал между абзацами, используя свойство `paragraph_format` модуля `python-docx` ❶. Межстрочный интервал до и после скрытой строки устанавливается равным 0 точкам, обеспечивая, чтобы результат на выходе не имел подозрительно больших промежутков между абзацами, как на рис. 6.9 слева.

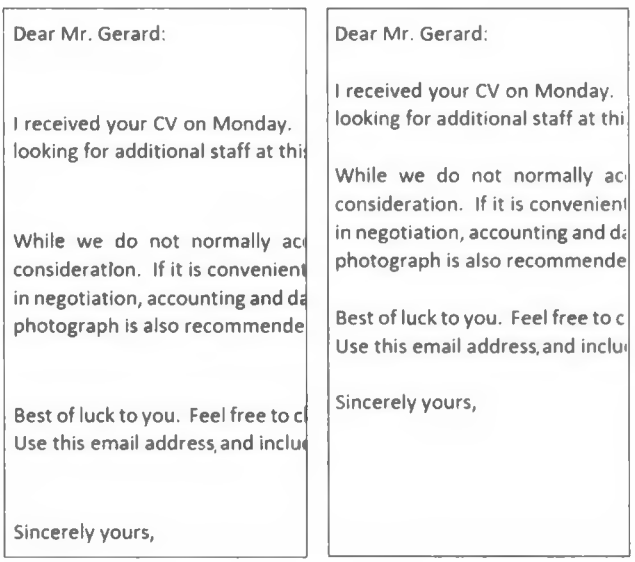


Рис. 6.9. Интервалы между строками фальшивого сообщения без форматирования абзацев с помощью `python-docx` (слева) и с форматированием (справа)

Далее определите рабочее пространство, получив длину списка, содержащего реальное сообщение ❷. Помните, что скрытое реальное сообщение должно быть короче видимого фальшивого сообщения с тем, чтобы имелось достаточно пустых строк для его хранения. Вслед за этим инициируйте счетчик. Программа будет использовать его для отслеживания того, какую строку (элемент списка) она в настоящее время обрабатывает в реальном сообщении.

Поскольку список, сделанный из фальшивого сообщения, является самым длинным и задает размерное пространство для реального сообщения, пройдите по фальшивому сообщению в цикле, используя два условия: 1) достигнут ли конец реального сообщения; 2) является ли строка в фальшивом списке пустой ❸. Если все еще есть строки реального сообщения, а строка фальшивого сообщения пуста, то используйте счетчик `count_real` в качестве индекса для списка `real_list` и модуль `python-docx` для ее добавления в документ ❹.

Возьмите индекс строки, которую вы только что добавили, взяв длину абзацев `doc.paragraphs` и вычтя 1 ❺. Затем используйте этот индекс для установки строки реального сообщения равным объекту `run` (это будет первый `[0]` элемент `run` в списке, т. к. реальное сообщение использует один стиль) и сделайте цвет ее шрифта белым ❻. Поскольку в этом блоке кода программа теперь добавила строку из реального списка, счетчик `count_real` увеличивается на 1 ❼.

Последующий блок `else` обращается к случаю, когда строка, выбранная из фальшивого списка в цикле `for`, не является пустой. В этом случае строка фальшивого сообщения добавляется непосредственно в абзац ❽. Завершите цикл `for`, вызвав функцию межстрочного интервала `set_spacing()` ❾.

После того как длина реального сообщения будет превышена, цикл `for` продолжит добавлять оставшуюся часть фальшивого сообщения — в данном случае информацию с подписью мистера Курца, — и в последней строке кода сохраните документ в виде файла Word с расширением .docx ❿. Конечно, в реальной жизни вы хотели бы использовать менее подозрительное имя файла, чем ciphertext_message_letterhead.docx!

Обратите внимание, что, поскольку вы используете цикл `for`, основанный на фальшивом сообщении, добавление каких-либо скрытых строк после завершения цикла `for`, т. е. после того, как будет достигнут конец элементов в фальшивом списке, станет невозможным. Если вы хотите иметь больше места, то внизу фальшивого сообщения необходимо ввести жесткий перевод строки, но будьте осторожны — не добавьте так много, что это приведет к разрыву страницы и созданию загадочной пустой страницы!

Выполните программу, откройте сохраненный документ Word, нажмите комбинацию клавиш <Ctrl>+<A>, тем самым вы выделите весь текст, а затем установите цвет подсветки (рис. 6.4) в темно-серый, для того чтобы увидеть оба сообщения. Секретное сообщение должно быть раскрыто (рис. 6.10).

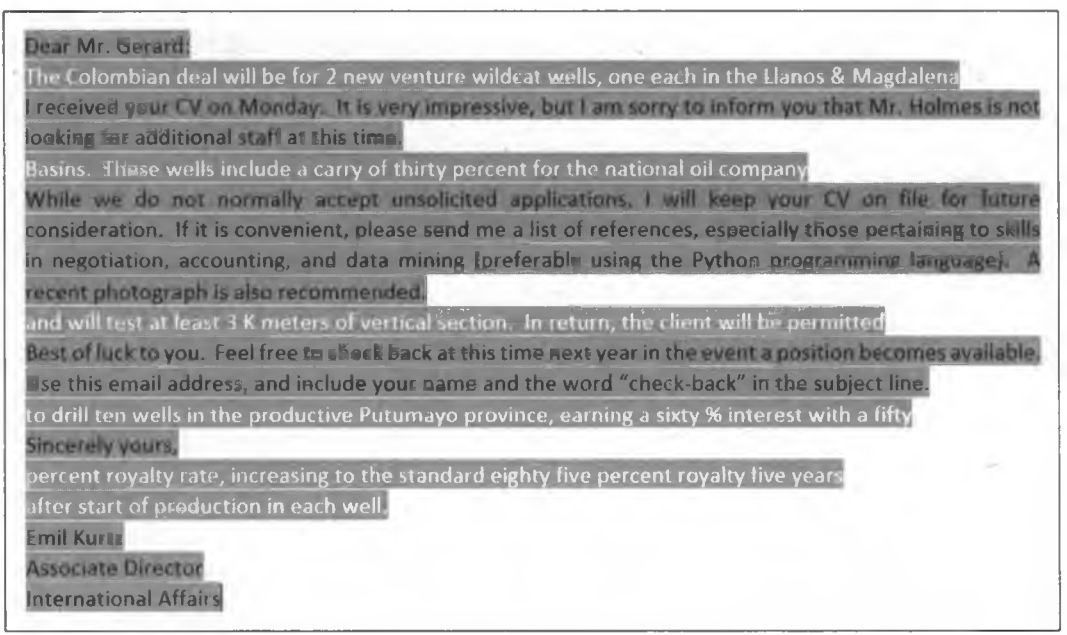

Dear Mr. Gerard:
The Colombian deal will be for 2 new venture wildcat wells, one each in the Llanos & Magdalena
I received your CV on Monday. It is very impressive, but I am sorry to inform you that Mr. Holmes is not
looking for additional staff at this time.
Basins. Those wells include a carry of thirty percent for the national oil company
While we do not normally accept unsolicited applications, I will keep your CV on file for future
consideration. If it is convenient, please send me a list of references, especially those pertaining to skills
in negotiation, accounting, and data mining (preferably using the Python programming language). A
recent photograph is also recommended.
and will test at least 3 K meters of vertical section. In return, the client will be permitted
Best of luck to you. Feel free to check back at this time next year in the event a position becomes available.
Use this email address, and include your name and the word "check-back" in the subject line.
to drill ten wells in the productive Putumayo province, earning a sixty % interest with a fifty
Sincerely yours,
percent royalty rate, increasing to the standard eighty five percent royalty five years
after start of production in each well.
Emil Kurtz
Associate Director
International Affairs

Рис. 6.10. Документ Word выделен темно-серым цветом для показа фальшивого сообщения и незашифрованного реального сообщения

Добавление шифра Виженера

До этого программный код использовал простую текстовую версию реального сообщения, поэтому любой, кто изменит цвет подсветки документа, сможет прочитать и узнать скрытую в нем конфиденциальную информацию. Поскольку вы знаете, что мистер Курц зашифровал ее с помощью шифра Виженера, вернитесь назад и поменяйте код, заменив открытый текст шифротекстом. Для этого найдите следующую строку:

```
real_text = docx.Document('realMessage.docx')
```

Эта строка загружает реальное сообщение в виде открытого текста, поэтому измените имя файла на то, которое показано ниже полужирным шрифтом:

```
real_text = docx.Document('realMessage_Vig.docx')
```

Повторно выполните программу и снова откройте скрытый текст, выделив весь документ и сделав цвет подсветки темно-серым (рис. 6.11).

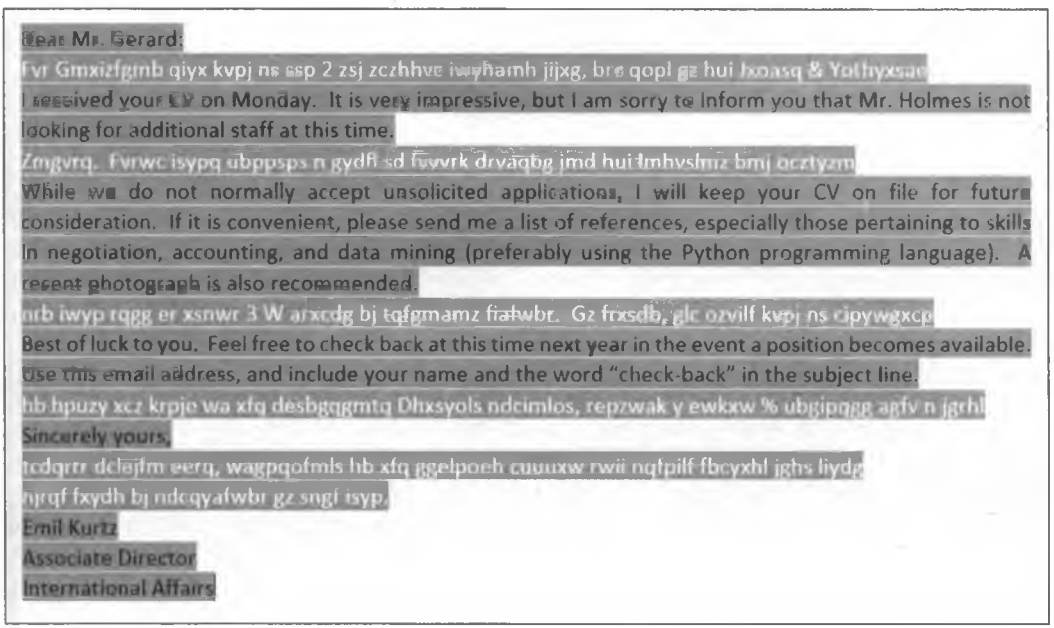


Рис. 6.11. Документ Word, выделенный темно-серым цветом, для показа фальшивого сообщения и зашифрованного реального сообщения

Секретное сообщение должно быть видимым, но нечитаемым для тех, кто не может расшифровать шифр. Сравните зашифрованное сообщение на рис. 6.11 с незашифрованной версией на рис. 6.10. Обратите внимание, что в обеих версиях встречаются числа и знак %. Они были сохранены для демонстрации потенциальных ловушек, связанных с выбором шифрования. Вы хотели бы улучшить шифр Виженера, включив эти символы — либо просто прописав их. Благодаря этому даже если ваше сообщение будет обнаружено, вы оставите минимально возможное количество подсказок относительно его темы.

Если вы хотите закодировать с помощью шифра Виженера свое собственное сообщение, то поищите в Интернете по запросу "online vigenere encoder" (онлайновый кодировщик Виженера). Вы найдете несколько веб-сайтов, таких как **http://www.cs.du.edu/~snarayan/crypt/vigenere.html**, которые позволяют набирать или вставлять открытый текст. И если вы хотите написать собственную Python-программу шифрования с использованием шифра Виженера, то обратитесь к книге Эла Свейгарта "Взламывание кодов с помощью Python" (Sweigart A. Cracking codes with Python. No Cramp Press, 2018).

Если вы поигрываете со своими реальными сообщениями, зашифрованными или незашифрованными, убедитесь, что вы используете тот же шрифт, что и в фальшивом сообщении. Например, типографский шрифт Helvetica Italic и размер шрифта 12 пт. Как вы помните из *разд. "Учет типов шрифтов, кернинга и межзнакового интервала" ранее в этой главе*, при попытке перемешать шрифты, в особенности пропорциональные и моноширинные, скрытые строки сообщения могут быть перенесены на новую строку, что приведет к неравномерному расстоянию между абзацами реального сообщения.

Обнаружение скрытого сообщения

Могла ли Джоан Ватсон или любой другой детектив быстро найти ваше скрытое послание? По правде говоря, скорее всего, нет. На самом деле, когда я пишу это, я смотрю эпизод сериала "Элементарно", где Джоан занята расследованием компании, читая стопку с распечатками электронной почты! Использование шифра Виженера, возможно, было просто вкраплением ленивой режиссуры в общем-то разумно поставленный сериал. Тем не менее мы можем поразмышлять над тем, что могло бы вас выдать.

Во-первых, поскольку окончательная информация о сделке, вероятно, не была отправлена до даты, близкой к дате сделки, поиск может быть ограничен корреспонденцией, отправленной после заключения сделки, тем самым устраняя много шума. Разумеется, детектив не будет точно знать, что конкретно он ищет — или даже о том, существует ли "крот", — что оставляет большое пространство для поиска. И всегда есть вероятность, что информация была передана в телефонном разговоре или во время тайной встречи.

Допуская, что имелся поддающийся обработке объем электронной почты и разведывалась гипотеза скрытого сообщения, следователь мог бы обнаружить ваши невидимые чернила несколькими способами. Например, орфографический корректор Word не будет отмечать белые, бессмысленные зашифрованные слова до тех пор, пока они не будут сделаны видимыми. Если вы в качестве проверки прокручивали документ и сбросили цвет шрифта на нескольких скрытых словах, то они будут окончательно скомпрометированы, даже после того, как их цвет был восстановлен до белого. Корректор сразу же подчеркнет их обличительной красной волнистой линией (рис. 6.12).

Если ведущий расследование детектив использует альтернативу текстовому редактору Word для открытия документа, то орфографический корректор этого продукта,

скорее всего, обнаружит скрытые слова (рис. 6.13). Указанный риск в некоторой степени сглаживается за счет доминирования на рынке текстового редактора Microsoft Word.

I received your CV on Monday. It is very impressive, but I am sorry to inform you that Mr. Holmes is not looking for additional staff at this time.

While we do not normally accept unsolicited applications, I will keep your CV on file for future consideration. If it is convenient, please send me a list of references, especially those pertaining to skills in negotiation, accounting, and data mining (preferably using the Python programming language). A recent photograph is also recommended.

Рис. 6.12. Ранее обнаруженные невидимые зашифрованные слова, подчеркнутые инструментом Word орфографической и грамматической коррекции

Dear Mr. Gerard:

I received your CV on Monday. It is very impressive, but I am sorry to inform you that Mr. Holmes is not looking for additional staff at this time.

While we do not normally accept unsolicited applications, I will keep your CV on file for future consideration. If it is convenient, please send me a list of references, especially those pertaining to skills in negotiation, accounting, and data mining (preferably using the Python programming language). A recent photograph is also recommended.

Best of luck to you. Feel free to check back at this time next year in the event a position becomes available. Use this email address, and include your name and the word "check-back" in the subject line.

Sincerely yours,

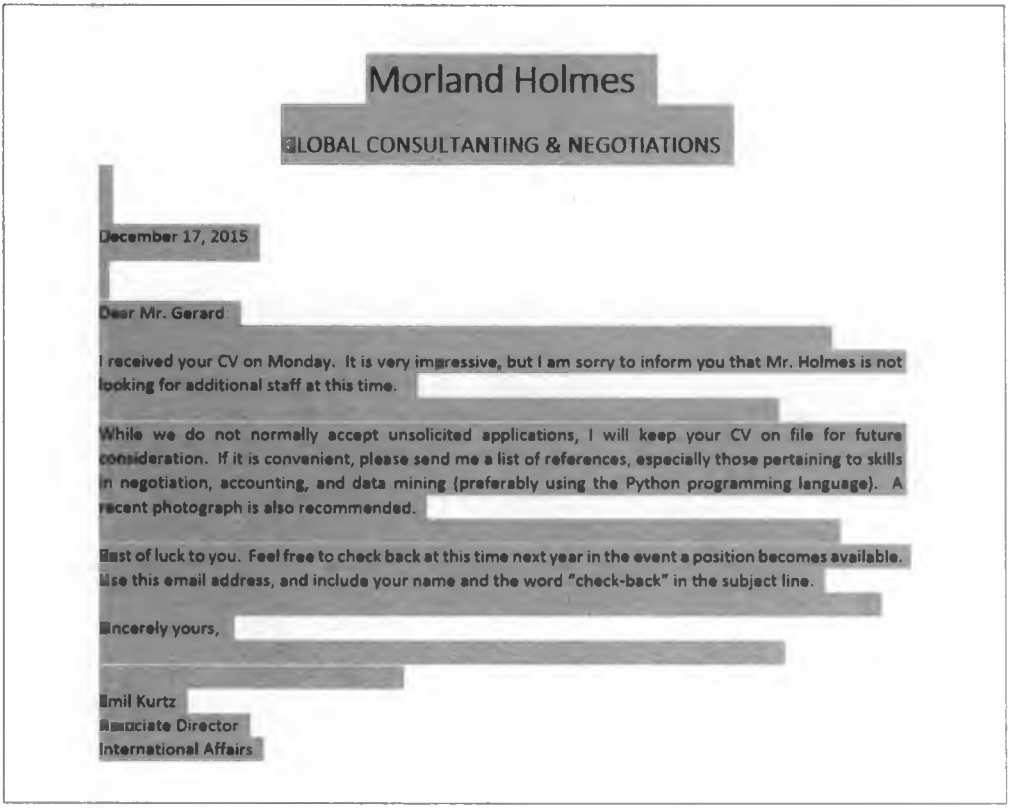
Emil Kurtz
Associate Director
International Affairs

Рис. 6.13. Средство проверки орфографии в текстовом редакторе LibreOffice Writer выделит невидимые слова

Во-вторых, использование сочетания клавиш <Ctrl>+<A> для выделения всего текста в Word не откроет скрытый текст, но будет указывать на то, что некоторые пустые строки длиннее других (рис. 6.14), намекая очень наблюдательным на то, что что-то тут не так.

В-третьих, открытие документа Word с помощью функции предварительного просмотра в некоторых программах электронной почты может выявить скрытый текст, когда содержимое выбирается с помощью прокрутки или с помощью нажатия комбинации клавиш <Ctrl>+<A> (рис. 6.15).

Но в отличие от обнаружения текста при выделении скрытого текста в панели предварительного просмотра почтовой службы Yahoo!Почта то же самое неверно в панели предварительного просмотра Microsoft Outlook на рис. 6.16.



Morland Holmes

GLOBAL CONSULTANTING & NEGOTIATIONS

December 17, 2015

Dear Mr. Gerard:

I received your CV on Monday. It is very impressive, but I am sorry to inform you that Mr. Holmes is not looking for additional staff at this time.

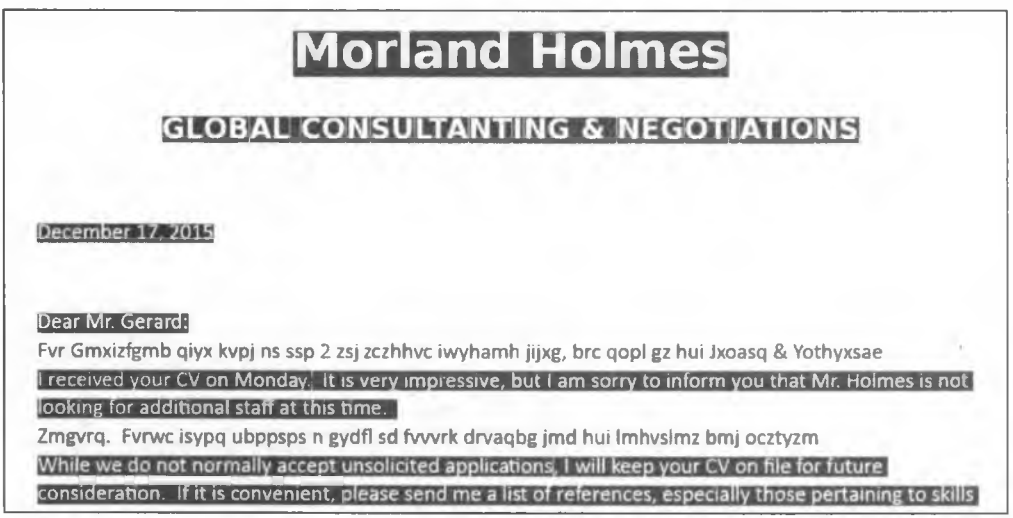
While we do not normally accept unsolicited applications, I will keep your CV on file for future consideration. If it is convenient, please send me a list of references, especially those pertaining to skills in negotiation, accounting, and data mining (preferably using the Python programming language). A recent photograph is also recommended.

Best of luck to you. Feel free to check back at this time next year in the event a position becomes available. Use this email address, and include your name and the word "check-back" in the subject line.

Sincerely yours,

Emil Kurtz
Associate Director
International Affairs

Рис. 6.14. Выделение всего документа Word показывает различия в длине пустых строк.



Morland Holmes

GLOBAL CONSULTANTING & NEGOTIATIONS

December 17, 2015

Dear Mr. Gerard:

Fvr Gmxizfgmb qiyx kvpj ns ssp 2 zsj zczhhvc iwyhamh jijxg, brc qopl gz hui Jxoasq & Yothyxsae

I received your CV on Monday. It is very impressive, but I am sorry to inform you that Mr. Holmes is not looking for additional staff at this time.

Zmgvrq. Fvrwc isypq ubppsps n gydfl sd fvvvrk drvaqbg jmd hui lmhvslmz bmj ocztyzm

While we do not normally accept unsolicited applications, I will keep your CV on file for future consideration. If it is convenient, please send me a list of references, especially those pertaining to skills

Рис. 6.15. Выделение всего документа в панели предварительного просмотра почтовой службы Yahoo!Почта показывает скрытый текст

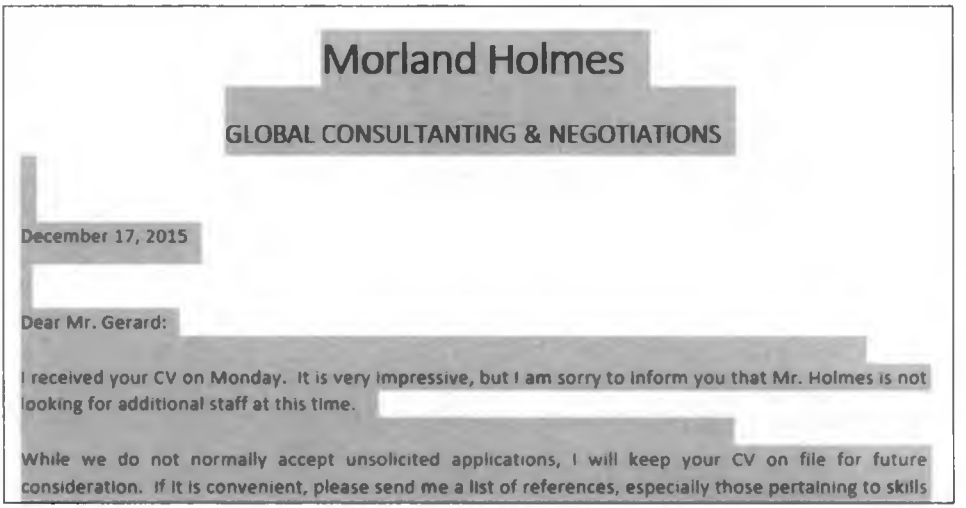


Рис. 6.16. При выделении всего документа в панели предварительного просмотра Microsoft Outlook скрытый текст не показывается

Наконец, сохранение документа Word в виде обычного текстового файла (*.txt) удалит все форматирование и оставит скрытый текст открытым (рис. 6.17).

```
Dear Mr. Gerard:
Fvr Gmxizfgmb qiyx kvpj ns ssp 2 zsj zczhhvc iwyhamh jijxg, brc qopl gz hui Jxoasq & Yothyxsae
I received your CV on Monday.  It is very impressive, but I am sorry to inform you that Mr. Holmes
is not looking for additional staff at this time.
Zmgvrq.  Fvrwc isypq ubppsps n gydfl sd fvvvrk drvaqbg jmd hui lmhvslmz bmj ocztyzm
While we do not normally accept unsolicited applications, I will keep your CV on file for future consideration.
If it is convenient, please send me a list of references, especially those pertaining to skills in negotiation,
```

Рис. 6.17. Сохранение документа Word в виде обычного текстового файла (*.txt) показывает скрытый текст

Для того чтобы скрыть секретное сообщение с помощью стеганографии, вы должны скрыть не только содержимое сообщения, но и тот факт, что сообщение вообще существует. Наши электронные невидимые чернила не всегда могут гарантировать это, но с точки зрения "крота" только что перечисленные недостатки предусматривают, что он либо допустил ошибку, которую теоретически можно было бы контролировать, либо следователь предпринял особые и маловероятные действия, такие как прокрутка текста, сохранение файлов в другом формате или использование менее распространенного текстового процессора. Допуская, что "крот" из сериала "Элементарно" учитывал перечисленные выше приемлемые риски, связанные с применением электронных невидимых чернил, это дает правдоподобное объяснение того, почему внутреннее расследование компании потерпело неудачу.

Резюме

В этой главе вы применили стеганографию для того, чтобы скрыть зашифрованное сообщение в документе Microsoft Word. Вы использовали сторонний модуль `python-docx` для прямого доступа к документу и манипуляции им с помощью языка

Python. Аналогичные сторонние модули доступны для работы с другими популярными типами документов, такими как электронные таблицы Excel.

Дальнейшее чтение

Онлайновую документацию по модулю `python-docx` можно найти по адресам **https://python-docx.readthedocs.io/en/latest/** и **https://pypi.python.org/pypi/python-docx**.

Книга Эла Свейгарта "Автоматизируй скучные вещи с помощью Python" (Sweigart A. Automate the boring stuff with Python. No Cramp Press, 2015) охватывает модули, которые позволяют языку Python оперировать PDF-файлами, файлами Word, электронными таблицами Excel и многим другим. *Глава 13* содержит полезное учебное руководство по модулю `python-docx`, а приложение к книге охватывает установку сторонних модулей с помощью менеджера установки пакетов pip.

Программы Python начального уровня для работы с шифрами можно найти в книге Эла Свейгарта "Взламывание кодов с помощью Python" (Sweigart A. Cracking codes with Python. No Cramp Press, 2018).

Книга Гэри Блэквуда "Таинственные сообщения" (Blackwood G. Mysterious messages. Penguin Group, 2009) является интересным и хорошо иллюстрированным повествованием об истории стеганографии и криптографии.

Практический проект: проверка числа пустых строк

Улучшите программу скрытых сообщений, написав функцию, которая сравнивает число пустых строк в фальшивом сообщении с числом строк в реальном сообщении. Если для скрытия реального сообщения места недостаточно, то указанная функция должна предупредить пользователя и сообщить ему о том, сколько пустых строк нужно добавить в фальшивое сообщение. Вставьте указанную функцию в копию программного кода elementary_ink.py непосредственно перед загрузкой шаблонного документа и вызовите эту функцию. Решение данной задачи можно найти в приложении к книге и на веб-сайте по адресу **https://www.nostarch.com/impracticalpython/** в файле elementary_ink_practice.py. Для тестирования скачайте файл realMessageChallenge.docx с того же веб-сайта и воспользуйтесь этим файлом в качестве реального сообщения.

Сложный проект: использование моноширинного шрифта

Перепишите код программы elementary_ink.py для моноширинных шрифтов и скройте собственное короткое сообщение в промежутках между словами. Описание моноширинных шрифтов *см. в разд. "Учет типов шрифтов, кернинга и межзнакового интервала" ранее в этой главе*. Как обычно, для сложных проектов решения не предоставляются.

7

РАЗВЕДЕНИЕ ГИГАНТСКИХ КРЫС С ПОМОЩЬЮ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

Генетические алгоритмы — это универсальные оптимизаторы, предназначенные для решения многосложных задач. Изобретенные в 1970-х годах, они относятся к классу эволюционных алгоритмов, названных так потому, что они имитируют дарвиновский процесс естественного отбора. Они особенно полезны тогда, когда имеется мало сведений о задаче, когда предстоит решить нелинейную задачу либо при поиске решений грубой силой в большом поисковом пространстве. Что еще лучше, эти алгоритмы легко понять и имплементировать.

В этой главе вы будете использовать генетические алгоритмы для выведения расы суперкрыс, которые способны терроризировать мир. После этого вы переметнетесь на другую сторону и за считаные секунды поможете Джеймсу Бонду взломать высокотехнологичный сейф. Оба проекта должны дать вам хорошее представление о механизме и силе генетических алгоритмов.

Поиск наилучших решений из всех возможных

Генетические алгоритмы *оптимизируют*. Это означает, что они отбирают лучшее решение (с учетом некоторых критериев) из набора имеющихся альтернатив. Например, если вы ищете самый быстрый маршрут для поездки на автомобиле из Нью-Йорка в Лос-Анджелес, то генетический алгоритм никогда не предложит вам лететь. Он может выбирать только из разрешенного множества условий, которые вы предоставляете. Как оптимизаторы, эти алгоритмы работают быстрее, чем традиционные методы, и способны избегать преждевременного схождения к неоптимальному ответу. Другими словами, они эффективно проводят поиск в пространст-

ве решения, причем делают это достаточно тщательно, избегая подбора хорошего ответа, когда есть еще лучше.

В отличие от систем исчерпывающего поиска, которые применяют чистую грубую силу, генетические алгоритмы не апробируют все возможные решения. Вместо этого они постоянно градуируют (оценивают) решения по степени признака, а затем используют их для того, чтобы, двигаясь вперед, делать "информированные догадки". Простой пример — игра "тепло-холодно", где вы ищете скрытый предмет в то время, как кто-то сообщает вам о том, становится ли теплее или холоднее, основываясь на вашей близости к предмету или направлении поиска. Генетические алгоритмы используют функцию приспособленности, аналогичную естественному отбору, с целью отбрасывания "более холодных" решений и концентрации на "более теплых". Базовый процесс заключается в следующем:

1. Случайно сгенерировать популяцию решений.
2. Измерить приспособленность каждого решения.
3. Выбрать наилучшие (самые теплые) решения и отбросить остальные.
4. Скрестить (рекомбинировать) элементы в наилучших решениях для того, чтобы принять новые решения.
5. Мутировать небольшое число элементов в решениях, изменив их значение.
6. Вернуться к шагу 2 и повторить.

Цикл "отобрать-скрестить-мутировать" продолжается до тех пор, пока не достигнет условия останова, например отыскания известного ответа, отыскания "достаточно хорошего" ответа (на основе минимального порога), завершения заданного числа итераций или достижения крайнего срока. Поскольку эти шаги очень похожи на процесс эволюции, дополненный выживанием наиболее приспособленных, применяемая в генетических алгоритмах терминология часто является скорее биологической, чем вычислительной.

Проект 13: разведение армии суперкрыс

Ловите момент стать сумасшедшим ученым с секретной лабораторией, полной кипящих мензурок, пузырящихся пробирок и машин, то и дело издающих свои "БЗЗЗТТТ" на все лады. Так что наденьте черные резиновые перчатки и займитесь превращением проворных мусорщиков в огромных монстров-людоедов.

Цель

Применить генетический алгоритм для симуляции разведения крыс со средним весом[1] 50 кг.

[1] Если быть более точным, с физической точки зрения здесь и далее речь, конечно, идет не о весе, а о массе. Но поскольку в обиходе используют термин "вес", то будем употреблять его, подразумевая массу как физическую величину. — *Прим. ред.*

Стратегия

Ваша мечта — развести расу крыс размером с бульмастифов (мы уже установили, что вы — сумасшедший). Вы начнете с *Rattus norvegicus*, бурой крысы, затем добавите немного искусственных подсластителей, немного атомной радиации 1950-х годов, много терпения и щепотку Python, но никакой генной инженерии — вы являетесь сторонником старой школы! Крысы вырастут от менее полкилограмма до ужасающих 50 кг размером примерно с самку бульмастифа (рис. 7.1).

Рис. 7.1. Сравнение размеров бурой крысы, самки бульмастифа и человека

Прежде чем приступить к такому масштабному начинанию, разумно смоделировать результаты на языке Python. И вы сформулировали нечто лучшее, чем просто план, — вы нарисовали некий графический псевдокод (рис. 7.2).

Процесс, показанный на рис. 7.2, описывает работу генетического алгоритма. Ваша цель состоит в том, чтобы произвести популяцию крыс со средним весом 50 кг из начальной популяции весом гораздо меньше, чем этот. В дальнейшем каждая популяция (или поколение) крыс представляет собой кандидатное решение задачи. Как и любой животновод, вы отбираете нежелательных самцов и самок, которых гуманно отправляете — для вас поклонников Остина Пауэрса — в злой контактный зоопарк. Затем вы спариваете и разводите оставшихся крыс, т. е. выполняете процесс, в генетическом программировании именуемый скрещиванием.

Потомство оставшихся крыс будет, по существу, того же размера, что и их родители, поэтому вам нужно некоторых из них мутировать. В то время как мутация является редким явлением и обычно приводит к нейтральному признаку, вплоть до неблагоприятного (в данном случае к низкому весу), иногда вы успешно выращиваете большую крысу.

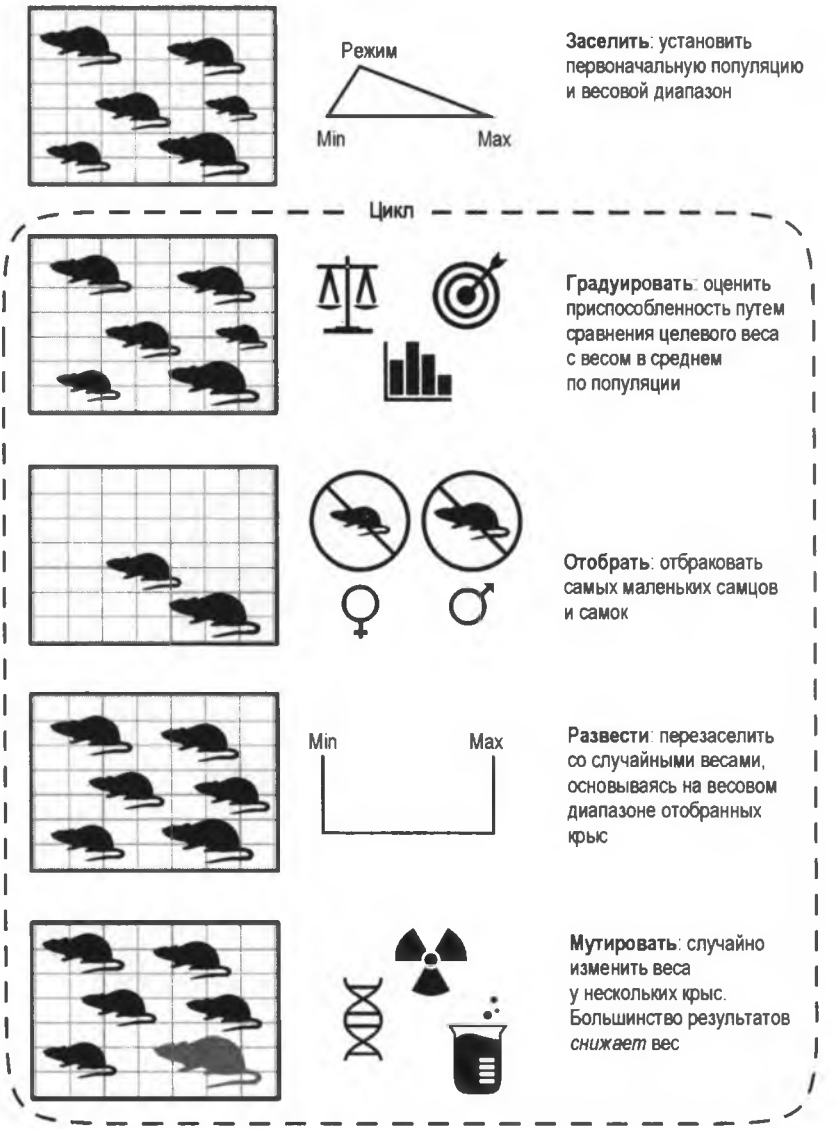


Рис. 7.2. Подход к разведению суперкрыс на основе генетического алгоритма

Затем весь процесс превращается в большой повторяющийся цикл, выполняемый органически или программно, заставляющий меня задуматься, не являемся ли мы в действительности просто виртуальными существами в инопланетной симуляции. В любом случае конец цикла — условие останова — это момент, когда крысы достигают нужного размера либо вы уже просто не можете больше переносить крыс.

Для ввода в симуляцию вам понадобится некоторая статистика. Используйте метрическую систему, т. к. вы являетесь ученым — неважно, сумасшедшим или адекватным. Вы уже знаете, что средний вес самки бульмастифа составляет 50 кг, и полезную статистику по крысам можно найти в табл. 7.1.

Таблица 7.1. *Статистика веса и размножения бурых крыс*

Параметры	Опубликованные значения
Минимальный вес, грамм	200
Средний вес (самки), грамм	250
Средний вес (самца), грамм	300–350
Максимальный вес, грамм*	600
Количество детенышей в помете	8–12
Пометы в год	4–13
Продолжительность жизни (на свободе, в неволе), лет	1–3; 4–6

* Исключительные особи в неволе могут достигать 1000 граммов.

Поскольку существуют как домашние, так и дикие бурые крысы, в некоторых статистических данных может иметься широкий разброс. За крысами в неволе, как правило, лучше ухаживают, чем за дикими крысами, в результате чего они больше весят, чаще размножаются и имеют больше детенышей. Поэтому, когда имеется диапазон, вы можете выбрать из более высокого уровня. В этом проекте начните с принятия допущений, приведенных в табл. 7.2.

Таблица 7.2. *Входные допущения для генетического алгоритма разведения суперкрыс*

Переменная или значение	Комментарии
`GOAL = 50000`	Целевой вес в граммах (самка бульмастифа)
`NUM_RATS = 20`	Общее число взрослых крыс, которых ваша лаборатория может поддерживать
`INITIAL_MIN_WT = 200`	Минимальная масса взрослой крысы (в граммах) в начальной популяции
`INITIAL_MAX_WT = 600`	Максимальная масса взрослой крысы (в граммах) в начальной популяции
`INITIAL_MODE_WT = 300`	Наиболее распространенный вес взрослых крыс (в граммах) в начальной популяции
`MUTATE_ODDS = 0.01`	Вероятность мутации, происходящей у крысы
`MUTATE_MIN = 0.5`	Скаляр на весе крысы с наименее благоприятной мутацией
`MUTATE_MAX = 1.2`	Скаляр по весе крысы с наиболее благоприятной мутацией
`LITTER_SIZE = 8`	Число детенышей на пару спаривающихся крыс
`LITTERS_PER_YEAR = 10`	Число пометов в год на пару спаривающихся крыс
`GENERATION_LIMIT = 500`	Поколенческое отсечение для останова программы размножения

Поскольку крысы размножаются очень часто, вам не придется учитывать продолжительность жизни. Даже если вы сохраните некоторых родителей из предыдущего

поколения, они будут быстро отбракованы, поскольку их потомство от поколения к поколению увеличивается в весе.

Код размножения суперкрыс

Код super_rats.py соблюдает общий рабочий поток, приведенный на рис. 7.2. Указанный код также можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**.

Ввод данных и допущений

Листинг 7.1 в глобальном пространстве в начале программы импортирует модули и назначает константам статистические данные, скаляры и допущения из табл. 7.2. После того как программа будет закончена и работоспособна, не стесняйтесь поэкспериментировать со значениями из этой таблице и посмотреть, как они влияют на ваши результаты.

Листинг 7.1. Импортирует модули и задает константы. Файл super_rats.py, часть 1

```
❶ import time
  import random
  import statistics
❷ # КОНСТАНТЫ (веса в граммах)
❸ GOAL = 50000
  NUM_RATS = 20
  INITIAL_MIN_WT = 200
  INITIAL_MAX_WT = 600
  INITIAL_MODE_WT = 300
  MUTATE_ODDS = 0.01
  MUTATE_MIN = 0.5
  MUTATE_MAX = 1.2
  LITTER_SIZE = 8
  LITTERS_PER_YEAR = 10
  GENERATION_LIMIT = 500
  # обеспечить для племенных пар четное число крыс:
❹ if NUM_RATS % 2 != 0:
      NUM_RATS += 1
```

Начните с импорта модулей `time`, `random` и `statistics` ❶. Модуль `time` будет использоваться для регистрации времени выполнения вашего генетического алгоритма. Засекать время работы генетических алгоритмов интересно хотя бы тем, чтобы они позволяют испытать восторг от того, как быстро они способны находить решение.

Модуль `random` удовлетворит стохастическим потребностям алгоритма, а модуль `statistics` будет использоваться для получения средних значений. Такое примене-

ние указанного модуля является слабоватым, но я хочу, чтобы вы о нем знали, т. к. он бывает довольно удобным.

Далее задайте входные переменные, описанные в табл. 7.2, и обязательно обратите внимание на то, что единицами измерения являются граммы ❷. Для имен используйте прописные буквы, т. к. они представляют константы ❸.

Прямо сейчас мы собираемся исходить из допущения, что используются племенные *пары*, поэтому убедитесь, что пользователь ввел четное число крыс, а если это не так, добавьте крысу ❹. *В разд. "Сложные проекты" далее в этой главе* вы сможете поэкспериментировать с альтернативными гендерными распределениями.

Инициализация популяции

Листинг 7.2 является закупочным представителем программы. Он "идет" в зоомагазин и выбирает крыс для первоначальной племенной популяции. Так как вы хотите иметь племенные пары, он должен выбрать четное число крыс. И поскольку вы не можете позволить себе одно из этих причудливых логовищ для вулкановых хомячков с неограниченным пространством, вам нужно будет в каждом поколении поддерживать число взрослых крыс постоянным, хотя их число может временно увеличиваться с учетом размещения помета. Помните, что крысы будут нуждаться все в большем и большем пространстве по мере того, как они будут вырастать до размера крупных собак!

Листинг 7.2. Определяет функцию, создающую первоначальную популяцию крыс. Файл super_rats.py, часть 2

```
❶ def populate(num_rats, min_wt, max_wt, mode_wt):
      """Инициализировать популяцию треугольным распределением весов."""
   ❷ return [int(random.triangular(min_wt, max_wt, mode_wt))\
             for i in range(num_rats)]
```

Функция `populate()` должна знать количество взрослых крыс, к которому вы стремитесь, минимальный и максимальный вес крыс, а также наиболее часто встречающийся вес ❶. Обратите внимание, что все эти аргументы будут использовать константы, находящиеся в глобальном пространстве. Для того чтобы к ним обратиться, вам не нужно передавать их в функцию в качестве аргументов. Но я делаю это здесь и в последующих функциях для ясности и потому, что доступ к *локальным* переменным выполняется эффективнее.

Вы будете использовать четыре упомянутых выше аргумента с модулем `random`, который включает в себя разные типы распределений. Здесь вы будете применять треугольное распределение, потому что оно обеспечивает четкий контроль над минимальным и максимальным размерами и позволяет статистически моделировать асимметрию.

Поскольку бурые крысы существуют как в дикой природе, так и в неволе — в зоопарках, лабораториях и в качестве домашних животных, их вес смещен в сторону

увеличения. Дикие крысы, как правило, меньше, поскольку их жизнь неприятна, груба и коротка, хотя лабораторные крысы этот момент могут оспорить! Примените операцию включения в список для перебора числа крыс в цикле и назначения веса каждой из них. Свяжите все это вместе инструкцией `return` ❷.

Измерение приспособленности популяции

Измерение приспособленности крыс — это двухэтапный процесс. Сначала проградуируйте всю популяцию, сравнив средний вес всех крыс с целью — бульмастифом. Затем проградуируйте каждую отдельную крысу. Возможность размножаться снова получают только те крысы, чей вес находится в верхних *n* процентах, как определено переменной `NUM_RATS`. Хотя средний вес популяции является допустимым показателем приспособленности, его основная роль здесь заключается в определении момента, когда можно прекратить цикл и объявить об успехе.

В листинге 7.3 определены функции `fitness()` и `select()`, которые вместе образуют измерительную часть вашего генетического алгоритма.

Листинг 7.3. Определяет измерительный шаг генетического алгоритма. Файл super_rats.py, часть 3

```
❶ def fitness(population, goal):
      """Измерить пригодность популяции, основываясь на
         среднем значении атрибута относительно цели."""
      ave = statistics.mean(population)
      return ave / goal

❷ def select(population, to_retain):
      """Отбраковать популяцию, оставив только заданное
         число ее членов."""
   ❸ sorted_population = sorted(population)
   ❹ to_retain_by_sex = to_retain//2
   ❺ members_per_sex = len(sorted_population)//2
   ❻ females = sorted_population[:members_per_sex]
     males = sorted_population[members_per_sex:]
   ❼ selected_females = females[-to_retain_by_sex:]
     selected_males = males[-to_retain_by_sex:]
   ❽ return selected_males, selected_females
```

Определите функцию, которая будет градуировать приспособленность текущего поколения ❶. Используйте модуль `statistics` для получения среднего значения популяции и верните это значение, деленное на целевой вес. Когда это значение будет равно 1 или больше, вы будете знать, что пришло время прекратить размножение.

Затем определите функцию, которая отбраковывает популяцию крыс, основываясь на весе, вплоть до значения `NUM_RATS`, представленного здесь параметром `to_retain` ❷.

Она также примет аргумент `population`, который будет представлять родителей каждого поколения.

Теперь отсортируйте популяцию так, чтобы можно было отличить больших особей от малых ❹. Возьмите число крыс, которое вы хотите сохранить, и разделите его на 2, используя деление с округлением вниз так, чтобы результат был целым числом ❺. Выполните этот шаг для того, чтобы можно было оставить самых больших самцов и самок крыс. Если вы выберете только самых крупных крыс в популяции, то вы теоретически будете выбирать только самцов. Общее число членов текущей популяции можно получить по полу, разделив `sorted_population` на 2, снова используя деление с округлением вниз ❻.

Самцы крыс, как правило, крупнее самок, поэтому сделайте два упрощающих допущения: во-первых, примите за основу, что ровно половина популяции — это самки и, во-вторых, что самая большая самка крысы не тяжелее самого малого самца крысы. Это означает, что первая половина отсортированного популяционного списка представляет самок ❻, а вторая половина — самцов. Затем создайте два новых списка, разделив `sorted_population` пополам, взяв нижнюю половину для самок и верхнюю половину для самцов. Теперь осталось только взять самых больших крыс с конца каждого из этих списков ❼ — используя отрицательный срез — и вернуть их ❽. Оба списка содержат родителей следующего поколения.

При первом выполнении этой функции она лишь отсортирует крыс по полу, т. к. первоначальное число крыс уже равно константе `NUM_RATS`. После этого аргумент входящей популяции будет включать в себя как родителей, так и детенышей, и его значение будет превышать константу `NUM_RATS`.

Разведение нового поколения

Листинг 7.4 определяет в программе шаг "скрещивания", т. е. он порождает следующее поколение. Ключевым допущением является то, что вес каждого детеныша будет больше или равен весу матери и меньше или равен весу отца. Исключения из этого правила будут обрабатываться в "мутационной" функции.

Листинг 7.4. Определяет функцию, которая порождает новое поколение крыс. Файл super_rats.py, часть 4

```
❶ def breed(males, females, litter_size):
      """Скрестить гены среди членов (весов) популяции."""
   ❷ random.shuffle(males)
     random.shuffle(females)
   ❸ children = []
   ❹ for male, female in zip(males, females):
      ❺ for child in range(litter_size):
         ❻ child = random.randint(female, male)
         ❼ children.append(child)
   ❽ return children
```

Функция `breed()` в качестве аргументов принимает списки весов отобранных самцов и самок, возвращенных из функции `select()`, а также размер помета ❶. Затем произвольно перетасуйте два списка ❷, потому что вы отсортировали их в функции `select()` и их обход в цикле без перетасовки приведет к тому, что самый маленький самец будет спарен с самой маленькой самкой и т. д. Вы должны учитывать любовь и романтику; самого крупного самца может притягивать к самой миниатюрной самке!

Создайте пустой список для хранения их детенышей ❸. Теперь что касается шурымуры. С помощью встроенной функции `zip()` пройдите в цикле по перетасованным спискам, спаривая самца с самкой из каждого списка ❹. Каждая пара крыс может иметь несколько детенышей, поэтому начните еще один цикл, который в качестве диапазона использует размер помета ❺. Размер помета — это константа, именуемая `LITTER_SIZE`, которую вы указали во входных параметрах, поэтому если значение равно 8, то вы получите восемь детенышей.

Для каждого детеныша выберите вес наугад между весом матери и отца ❻. Обратите внимание, что вам не нужно указывать `male + 1`, потому что `randint()` использует все числа в предоставленном диапазоне. Обратите также внимание на то, что эти два значения могут быть одинаковыми, но первое значение (вес матери) никогда не может быть больше второго (веса отца). Это еще одна причина для упрощающего допущения о том, что самки должны быть не больше самого маленького самца. Завершите цикл, добавив каждого детеныша в список `children` ❼, а затем верните `children` ❽.

Мутирование популяции

Небольшой процент детенышей должен испытывать мутации, и большинство из них должны приводить к признакам, которые являются неблагоприятными. Это означает более низкие, чем ожидалось, веса, включая "коротышек", которые не выживут. Но время от времени полезная мутация все-таки приведет к более тяжелой крысе.

В листинге 7.5 определена функция `mutate()`, которая применяет допущения о мутации, указанные в списке констант. После вызова `mutate()` настанет время проверить приспособленность новой популяции и начать цикл заново, если целевой вес не был достигнут.

Листинг 7.5. Определяет функцию, которая мутирует небольшую часть популяции. Файл super_rats.py, часть 5

```
❶ def mutate(children, mutate_odds, mutate_min, mutate_max):
      """Случайно изменить веса крыс, используя входные шансы
         и дробные изменения."""
   ❷ for index, rat in enumerate(children):
         if mutate_odds >= random.random():
            ❸ children[index] = round(rat * random.uniform(mutate_min,
                                                           mutate_max))
      return children
```

Указанная функция нуждается в списке детенышей, шансах возникновения мутации, а также минимальном и максимальном влиянии мутации ❶. Влияния — это скаляры, которые вы будете применять к весу крысы. В вашем списке констант в начале программы (и в табл. 7.2) они смещены в минимальную сторону, т. к. большинство мутаций не приводят к благоприятным признакам.

Пройдите в цикле по списку детенышей и примените `enumerate()` — удобную встроенную функцию, которая действует как автоматический счетчик для получения индекса ❷. Затем примените метод `random()` для того, чтобы сгенерировать случайное число между 0 до 1, и сравните его с вероятностью возникновения мутации.

Если переменная `mutate_odds` больше или равна случайно сгенерированному числу, то крыса (вес) в этом индексе мутирует. Выберите значение мутации из равномерного распределения, определяемого минимальными и максимальными значениями мутации; это, в сущности, случайное значение из диапазона между минимумом и максимумом. Поскольку эти значения смещены к минимуму, результатом, скорее всего, будет потеря веса, нежели его увеличение. Умножьте текущий вес на этот мутационный скаляр и округлите его до целого числа ❸. Завершите программу, вернув список мутировавших детенышей.

ПРИМЕЧАНИЕ

Что касается достоверности мутационной статистики, то вы можете найти исследования, которые демонстрируют, что благоприятные мутации являются очень редкими. Но есть и другие, которые сообщают о том, что такие мутации более распространены, чем мы себе представляем. Разведение собак показало, что достижение резких изменений в размерах (например, чихуа-хуа относительно датских догов) не требует миллионов лет эволюции. В знаменитом исследовании XX в. российский генетик Дмитрий Беляев начал с 130 серебристых лисиц и за 40-летний период добился значительных физиологических изменений, просто выбирая в каждом поколении самых прирученных лисиц.

Определение функции *main()*

В листинге 7.6 представлена функция `main()`, которая управляет другими функциями и определяет момент, когда вы выполнили условие останова. Она также будет показывать все важные результаты.

Листинг 7.6. Определяет функцию `main()`. Файл super_rats.py, часть 6

```
def main():
    """ Инициализировать популяцию, отобрать, вывести и мутировать,
        показать результаты."""
❶   generations = 0
❷   parents = populate(NUM_RATS, INITIAL_MIN_WT, INITIAL_MAX_WT,
                       INITIAL_MODE_WT)
    print("первоначальные веса популяции = {}".format(parents))
    popl_fitness = fitness(parents, GOAL)
```

```
    print("первоначальная приспособленность популяции = {}".
          format(popl_fitness))
    print("оставляемое число = {}".format(NUM_RATS))

❸ ave_wt = []

❹ while popl_fitness < 1 and generations < GENERATION_LIMIT:
        selected_males, selected_females = select(parents, NUM_RATS)
        children = breed(selected_males, selected_females, LITTER_SIZE)
        children = mutate(children, MUTATE_ODDS, MUTATE_MIN, MUTATE_MAX)
    ❺ parents = selected_males + selected_females + children
        popl_fitness = fitness(parents, GOAL)
    ❻ print("Приспособленность поколения {} = {:.4f}".
              format(generations, popl_fitness))
    ❼ ave_wt.append(int(statistics.mean(parents)))
        generations += 1
❽ print("средний вес на поколение = {}".format(ave_wt))
    print("\nчисло поколений = {}".format(generations))
    print("число лет = {}".format(int(generations / LITTERS_PER_YEAR)))
```

Начните функцию с инициализации пустого списка для хранения числа поколений. В конечном итоге вы будете использовать его для выяснения количества лет, которое потребовалось для достижения вашей цели ❸.

Далее вызовите функцию `populate()` ❷ и сразу же напечатайте результаты. Затем получите приспособленность вашей первоначальной популяции и напечатайте ее вместе с числом крыс (которое является константой `NUM_RATS`), оставляемым каждым поколением.

Ради любопытства инициализируйте список для хранения среднего веса каждого поколения для его просмотра в конце ❽. Если начертить эти числа на графике относительно числа лет, то вы увидите, что тренд будет экспоненциальным.

Теперь начните большой генетический цикл отбора-спаривания-мутирования. Он имеет вид цикла `while`, в котором условия останова достигают целевого веса либо большого числа поколений без достижения целевого веса ❹. Обратите внимание, что после мутации детенышей вам нужно объединить их с родителями, создав новый список родителей ❺. Детенышам требуется около пяти недель на то, чтобы подрасти и начать размножение, но вы можете учесть это, скорректировав константу `LITTERS_PER_YEAR` вниз от максимально возможного значения (см. табл. 7.1), как мы сделали здесь.

В конце каждого цикла выведите на экран результаты функции `fitness()` до четырех знаков после точки с целью мониторинга алгоритма и обеспечения его продвижения, как того требуется ❻. Получите средний вес поколения, добавьте его в список `ave_wt` ❼, а затем увеличьте число поколений на 1.

Завершите функцию `main()`, показав список средних весов в расчете на поколение, число поколений и число лет, рассчитанных с помощью переменной `LITTERS_PER_YEAR` ❽.

Выполнение функции *main()*

Завершите уже хорошо знакомой условной инструкцией запуска программы в автономном режиме либо в виде модуля. Получите время завершения работы и напечатайте продолжительность выполнения программы. Информация о производительности должна печататься только при запуске модуля в автономном режиме, поэтому обязательно поместите ее под выражением `if` (листинг 7.7).

Листинг 7.7. Выполняет функцию `main()` и модуль `time`, если программа не импортирована. Файл super_rats.py, часть 7

```
if __name__ == '__main__':
    start_time = time.time()
    main()
    end_time = time.time()
    duration = end_time - start_time
    print("\nВремя выполнения этой программы составило {} секунд.".
          format(duration))
```

Резюме

С параметрами табл. 7.2 выполнение программы super_rats.py займет около двух секунд. В среднем на то, чтобы крысы достигли целевого веса 50 кг, им потребуется около 345 поколений, или 34,5 лет. Для сумасшедшего ученого это слишком долго, чтобы оставаться сумасшедшим! Но вооружившись своей программой, вы можете искать способы сократить время до цели.

Процедура анализа чувствительности работает путем внесения многочисленных изменений в единственную переменную и оценивания результатов. Однако следует проявлять осторожность, если некоторые переменные зависят друг от друга. И поскольку результаты являются стохастическими (случайными), необходимо с каждым изменением параметра делать несколько прогонов для того, чтобы уловить диапазон возможных результатов.

В программе разведения крыс можно контролировать две вещи: число размножающихся крыс (`NUM_RATS`) и шансы возникновения мутации (`MUTATE_ODDS`). На шансы мутации влияют такие факторы, как рацион кормления и воздействие радиации. Если изменять эти переменные по одному и повторять скрипт super_rats.py, то можно будет судить о влиянии каждой переменной на временну́ю шкалу проекта.

Прямое наблюдение состоит в том, что если начать с небольших значений каждой переменной и медленно их увеличивать, то получатся впечатляющие первоначальные результаты (рис. 7.3). После этого обе кривые быстро уменьшатся и сгладятся в классическом примере убывающего финансового возврата от инвестиций. Точка, в которой каждая кривая выравнивается, является ключом к оптимальной экономии денег и снижению работы.

Например, очень мало пользы от оставления более чем 300 крыс. Вы просто окажетесь в ситуации, когда вам придется кормить и ухаживать за большим числом лиш-

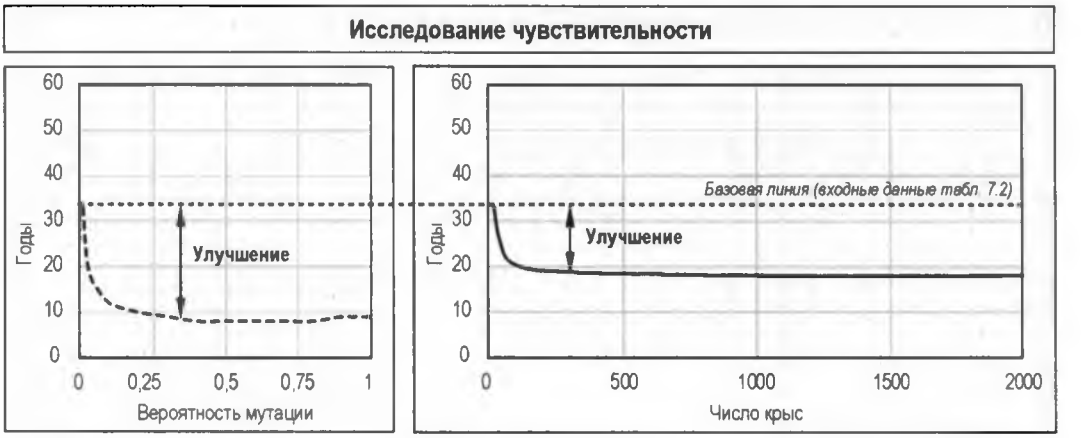


Рис. 7.3. Влияние двух параметров на время, необходимое для достижения целевого веса

них крыс. Точно так же, попытка увеличить шансы мутации свыше 0,3 принесет малую отдачу.

С диаграммами подобного рода легко планировать путь вперед. Горизонтальная пунктирная линия, помеченная как "базовая линия", представляет средний результат использования входных данных табл. 7.2. Это время можно потенциально сократить более чем на 10 лет, просто оставляя вместо 20 крыс 50. Также следует сосредоточиться на увеличении числа благоприятных мутаций. Это будет плодотворнее, но рискованнее и труднее контролировать.

Если повторно выполнить симуляцию с использованием 50 крыс и подтолкнуть шансы мутации до 0,05, то теоретически проект сможет завершиться за 14 лет, улучшив свои шансы на 246% по сравнению с первоначальным уровнем. Теперь это действительно оптимизация!

Разведение суперкрыс было забавным и простым способом разобраться в основах генетических алгоритмов. Но для того чтобы по-настоящему оценить их силу, стоит попробовать нечто посложнее. Вам нужна задача, решаемая грубой силой, т. е. исчерпывающим поиском среди всех вариантов, но которая слишком велика для грубой силы, и следующий проект — именно такая задача.

Проект 14: взламывание высокотехнологичного сейфа

Вы — Q[2], и у Джеймса Бонда есть проблема. Он должен присутствовать на элегантном званом обеде в особняке злодея, ускользнуть в личный кабинет этого человека и взломать его стенной сейф — детская игра для агента 007, за исключе-

[2] Q — это персонаж книг и фильмов о Джеймсе Бонде. Так же как и M, это кодовое имя, в данном случае оно означает первую букву слова Quartermaster (интендант, начальник снабжения). Q — глава Q Branch, исследовательского центра Британской секретной службы. Он постоянно снабжает Джеймса Бонда полезной шпионской техникой. См. **https://ru.wikipedia.org/wiki/Q_(бондиана)**. — *Прим. перев.*

нием одного: этот сейф — цифровой Humperdink BR549, который принимает 10 цифр, давая 10 млрд возможных комбинаций. И колеса замка́ не начинают вращаться до тех пор, пока не будут введены все числа. Приложить стетоскоп к этому сейфу и медленно поворачивать диск не получится!

Помните: вы — Q, и у вас уже есть устройство автодозвона, которое может исчерпывающе перебирать все возможные решения, но у Бонда просто не будет времени его использовать. И вот почему.

Кодовый замо́к в действительности следует называть *перестановочным* замком, потому что он требует *упорядоченных* комбинаций, которые по определению являются перестановками. Более конкретно, замки опираются на перестановки с повторением. Например, допустимой — хотя и небезопасной — комбинацией может быть 999999999.

Вы использовали итератор перестановок `permutations()` модуля `itertools` во время работы с анаграммами в *главе 3* и в *разд. "Практические проекты" главы 4*, но здесь это не поможет, потому что итератор `permutations()` возвращает перестановки *без* повторов. Для того чтобы создать правильный вид перестановки для замка, вам нужно использовать итератор `product()` модуля `itertools`, который вычисляет декартово произведение из многочисленных наборов чисел:

```
>>> from itertools import product
>>> combo = (1, 2)
>>> for perm in product(combo, repeat=2):
    print(perm)
(1, 1)
(1, 2)
(2, 1)
(2, 2)
```

Необязательный именованный аргумент `repeat` позволяет вам взять произведение итерируемого объекта, умножаемого на самого себя, что в данном случае вам как раз и нужно сделать. Обратите внимание, что функция `product()` возвращает все возможные комбинации тогда, как функция `permutations()` возвращает только `(1, 2)` и `(2, 1)`. Дополнительно об итераторе `product()` можно прочитать на вебстранице **https://docs.python.org/3.6/library/itertools.html#itertools.product.Listing**.

Листинг 7.8 представляет собой программу на Python под названием brute_force_cracker.py, в которой итератор `product()` используется для того, чтобы грубой силой добиться правильной комбинации.

Листинг 7.8. Применяет грубую силу для поиска комбинации сейфа. Файл brute_force_cracker.py

```
❶ import time
  from itertools import product

  start_time = time.time()

❷ combo = (9, 9, 7, 6, 5, 4, 3)
```

```
    # применить декартово произведение для генерирования перестановок
    # с повтором
❸ for perm in product([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], repeat=len(combo)):
    ❹ if perm == combo:
            print("Взломано! {} {}".format(combo, perm))

    end_time = time.time()
❺ print("\nВремя выполнения этой программы составило {} секунд.".format
        (end_time - start_time))
```

Импортируйте модуль `time` и из модуля `itertools` итератор `product` ❶. Получите время начала, а затем введите безопасную комбинацию в виде кортежа ❷. Далее примените функцию `product()`, которая для заданной последовательности возвращает кортежи всех перестановок с повтором. Последовательность содержит все допустимые элементы из первых цифр (0–9). В качестве аргумента `repeat` следует задать число цифр в комбинации ❸. Сравните каждый результат с комбинацией и выведите "Взломано!", если они совпадают, вместе с комбинацией и совпадающей перестановкой ❹. Завершите, показав на экране время выполнения ❺.

Это отлично работает для комбинаций длиной до восьми цифр. После этого время ожидания становится все более неудобным. Таблица 7.3 представляет собой регистрацию времени выполнения программы в зависимости от числа цифр в комбинации.

Таблица 7.3. *Времена выполнения относительно цифр в комбинации (процессор с тактовой частотой 2,3 ГГц)*

Число цифр	Время выполнения, секунды
5	0,035
6	0,147
7	1,335
8	12,811
9	133,270
10	1396,955

Обратите внимание, что добавление цифры в комбинацию увеличивает время выполнения на порядок. Это экспоненциальный рост. С 9 цифрами вы бы ожидали ответа более 2 минут. С 10 цифрами — более 20 минут! Для Бонда это долго, для того чтобы "поход в ванную комнату" оставался незамеченным. К счастью, вы — Q, и вы кое-что знаете о генетических алгоритмах. От вас требуется лишь одно — каким-то образом судить о приспособленности каждой кандидатной комбинации. Варианты включают в себя мониторинг колебаний в энергопотреблении, измерение временных задержек в операциях и прослушивание звуков. Будем считать, что используется звукоусилительный инструмент, а также инструмент для предотвраще-

ния блокировки после ввода нескольких неправильных комбинаций. Из-за сторожков в сейфе BR549 звуковой инструмент первоначально может сообщать вам только о том, сколько цифр являются правильными, а не какие именно цифры, но по прошествии очень небольшого времени ваш алгоритм может сойтись на решении.

Цель

Применить генетический алгоритм для быстрого отыскания комбинации сейфа в большом поисковом пространстве.

Стратегия

Стратегия здесь проста. Вы сгенерируете случайную последовательность из 10 чисел и сравните ее с реальной комбинацией, градуируя результат на основе совпадений; в реальном мире вы отыщите число совпадений, используя звуковой детектор, прикрепленный к двери сейфа. Затем вы измените одно значение в своем решении и снова сравните. Если найдено еще одно совпадение, то вы отбросите старую последовательность и двинетесь вперед с новой; в противном случае вы сохраните старую последовательность и повторите попытку.

Поскольку одно решение полностью заменяет другое, это представляет собой 100-процентное скрещивание генетического материала, поэтому вы, по существу, используете только отбор и мутацию. Отбор плюс мутация сами по себе генерируют надежный алгоритм *восхождения к вершине холма*. Восхождение к вершине холма — это оптимизационный технический прием, который начинается с произвольного решения и изменяет (мутирует) в решении одно-единственное значение. Если результат представляет собой улучшение, то новое решение сохраняется и процесс повторяется.

Проблема с восхождением к вершине холма заключается в том, что алгоритм может застрять в локальных минимумах или максимумах и не отыскать оптимальное глобальное значение. Представьте, что вы ищете самое низкое значение в волнообразной функции из рис. 7.4. Текущая наилучшая догадка отмечена большой черной

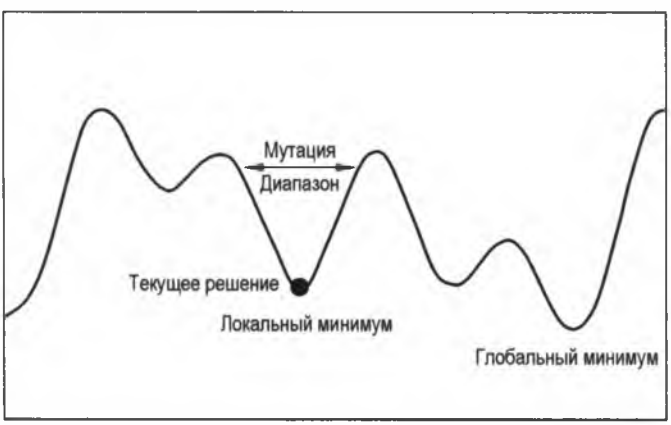


Рис. 7.4. Пример алгоритма восхождения к вершине холма, "застрявшего" в локальном минимуме

точкой. Если величина вносимого вами изменения (мутации) слишком мала, для того чтобы "избежать" локальной котловины, то алгоритм не найдет истинную нижнюю точку. С точки зрения алгоритма, поскольку каждое направление приводит к более худшему ответу, он, должно быть, нашел истинный ответ. Поэтому он преждевременно сходится на решении.

Использование скрещивания в генетических алгоритмах помогает избежать проблем преждевременного схождения, а также допускает относительно большие мутации. Поскольку здесь вы не беспокоитесь о соблюдении биологического реализма, мутационное пространство может охватывать все возможные значения в комбинации. Благодаря этому вы не сможете застрять, и восхождение к вершине холма становится приемлемым подходом.

Код взломщика сейфов

Код программы safe_cracker.py принимает комбинацию из *n* цифр и использует восхождение к вершине холма, для того чтобы достичь комбинации из случайной первоначальной точки. Указанный код можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**.

Настройка и определение функции *fitness()*

Листинг 7.9 импортирует необходимые модули и определяет функцию `fitness()`.

Листинг 7.9. Импортирует модули и определяет функцию `fitness()`. Файл safe_cracker.py, часть 1

```
❶ import time
  from random import randint, randrange

❷ def fitness(combo, attempt):
      """Сравнить элементы в двух списках и подсчитать
         число совпадений."""
      grade = 0

    ❸ for i, j in zip(combo, attempt):
          if i == j:
              grade += 1
      return grade
```

После импорта нескольких хорошо знакомых модулей ❶ определите функцию `fitness()`, которая в качестве аргументов принимает истинную комбинацию и предпринятое решение ❷. Назовите переменную `grade` и установить ее равной 0. Затем примените встроенную функцию `zip()` для перебора каждого элемента в комбинации и вашей попытки ❸. Если они являются одинаковыми, то добавьте 1 в переменную `grade` и верните ее. Обратите внимание, что вы регистрируете не индекс, который совпадает, а просто факт, что функция нашла совпадение. Этим эму-

лируется выход из устройства обнаружения звука. Первоначально оно способно вам сообщить лишь одно — сколько блокировочных колес повернуто, а не их местоположение.

Определение и запуск функции *main()*

Поскольку эта программа является короткой и простой, бóльшая часть алгоритма выполняется не в нескольких функциях, а в функции `main()` (листинг 7.10).

Листинг 7.10. Определяет функцию `main()` и запускает и хронометрирует программу, если она не была импортирована. Файл safe_cracker.py, часть 2

```
def main():
    """ Применить алгоритм восхождения к вершине холма
        для решения комбинации замков."""
❶   combination = '6822858902'
    print("Комбинация = {}".format(combination))
    # конвертировать комбинацию в список:
❷   combo = [int(i) for i in combination]

    # сгенерировать догадку и проградуировать приспособленность:
❸   best_attempt = [0] * len(combo)
    best_attempt_grade = fitness(combo, best_attempt)

❹   count = 0

    # эволюционировать догадку
❺   while best_attempt != combo:
        # скрестить
❻       next_try = best_attempt[:]

        # мутировать
        lock_wheel = randrange(0, len(combo))
❼       next_try[lock_wheel] = randint(0, 9)

        # проградуировать и отобрать
❽       next_try_grade = fitness(combo, next_try)
        if next_try_grade > best_attempt_grade:
            best_attempt = next_try[:]
            best_attempt_grade = next_try_grade
        print(next_try, best_attempt)
        count += 1

    print()
❾   print("Взломано! {}".format(best_attempt), end=' ')
    print("за {} попыток!".format(count))
```

```
if __name__ == '__main__':
    start_time = time.time()
    main()
    end_time = time.time()
    duration = end_time - start_time
⓾ print("\nВремя выполнения этой программы составило {:.5f} секунд.".
          format(duration))
```

Предоставьте истинную комбинацию в качестве переменной ❶ и примените операцию включения в список для ее конвертации в список ради удобства продвижения вперед ❷. Сгенерируйте список нулей, по длине равный комбинации, и назовите его лучшей попыткой `best_attempt` ❸. На данный момент комбинация является такой же хорошей, как и любая другая. Вы должны сохранить это имя — `best_attempt`, потому что по мере подъема к вершине холма вам нужно сохранять только лучшее решение. После того как вы сгенерировали первоначальную попытку, проградуируйте ее с помощью функции `fitness()`, а затем задайте значение переменной `best_attempt_grade`.

Начните переменную `count` с нуля. Программа будет использовать эту переменную для регистрации числа попыток, потребовавшихся на взлом кода ❼.

Теперь начните цикл `while`, который продолжается до тех пор, пока вы не отыщете комбинацию ❹. Назначьте копию переменной `best_attempt` переменной `next_try` ❺. Вы копируете ее для того, чтобы не столкнуться с проблемами зеркальных изменений; при изменении элемента в `next_try` вы не хотите случайно изменить `best_attempt`, потому что вы, возможно, будете продолжать ее использовать в случае, если `next_try` не пройдет тест на приспособленность.

Теперь самое время мутировать копию. Каждая цифра в комбинации поворачивает колесо блокировки в сейфе, поэтому назовите переменную `lock_wheel` и случайно установите ее равной местоположению индекса в комбинации. Он представляет местоположение единственного изменяемого в этой итерации элемента. Далее случайно выберите цифру и используйте ее для замены значения в местоположении, индексируемом переменной `lock_wheel` ❼.

Проградуируйте значение переменной `next_try`, и если оно лучше, чем предыдущая попытка, то сбросьте переменные `best_attempt` и `best_attempt_grade` в новые значения ❼. В противном случае `best_attempt` останется для следующей итерации неизменной. Напечатайте `next_try` и `best_attempt` бок о бок так, чтобы можно было прокрутить попытки, когда программа закончится, и посмотреть, как они эволюционировали. Завершите цикл, нарастив счетчик.

Когда программа отыщет комбинацию, покажите значение переменной `best_attempt` и число попыток, потребовавшихся на то, чтобы ее отыскать ❽. Помните, что аргумент `end=' '` предотвращает возврат каретки в конце напечатанной строки и помещает пробел между концом текущей строки и началом следующей строки.

Завершите программу с помощью условной инструкции для выполнения функции `main()` в автономном режиме и покажите время выполнения с точностью до пяти знаков после точки ⑩. Обратите внимание, что программный код хронометрирования идет после условного и, следовательно, не будет выполняться, если программа импортируется как модуль.

Резюме

Последние несколько строк результата работы программы safe_cracker.py показаны ниже. Для краткости я опустил большинство эволюционирующих сравнений. Прогон был для 10-значной комбинации.

```
[6, 8, 6, 2, 0, 5, 8, 9, 0, 0] [6, 8, 2, 2, 0, 5, 8, 9, 0, 0]
[6, 8, 2, 2, 0, 9, 8, 9, 0, 0] [6, 8, 2, 2, 0, 5, 8, 9, 0, 0]
[6, 8, 2, 2, 8, 5, 8, 9, 0, 0] [6, 8, 2, 2, 8, 5, 8, 9, 0, 0]
[6, 8, 2, 2, 8, 5, 8, 9, 0, 2] [6, 8, 2, 2, 8, 5, 8, 9, 0, 2]

Взломано! [6, 8, 2, 2, 8, 5, 8, 9, 0, 2] за 78 попыток!

Время выполнения этой программы составило 0.69172 секунд.
```

Десять миллиардов возможных комбинаций, и программа нашла решение всего за 78 попыток и менее чем за секунду. Даже Джеймс Бонд был бы этим впечатлен.

Ну, всё. Довольно с нас генетических алгоритмов. Вы использовали пример рабочего потока для размножения гигантских грызунов, а затем подрезали его, для того чтобы в мгновение ока подняться к вершине холма с применением грубой силы. Если вы хотите продолжить играть в цифрового Дарвина и поэкспериментировать с генетическими алгоритмами, то длинный список примеров приложений можно найти в Википедии (**https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications**). Примеры включают:

- моделирование глобальных изменений температуры;
- оптимизацию загрузки контейнеров;
- оптимизацию маршрута движения грузовиков;
- сети мониторинга за уровнем грунтовых вод;
- поведение самообучающегося робота;
- сворачивание белка;
- анализ редких событий;
- взламывание кодов;
- кластеризацию для функций подгонки;
- фильтрацию и обработку сигналов.

Дальнейшее чтение

Книга Клинтона Шеппарда "Генетические алгоритмы с помощью языка Python" (Sheppard C. Genetic algorithms with Python. Amazon Digital Services LLC, 2016) представляет собой введение в генетические алгоритмы начального уровня с использованием языка Python. Она продается в мягкой обложке либо как недорогая электронная книга по адресу **https://leanpub.com/genetic_algorithms_with_python/**.

Сложные проекты

Продолжайте разводить суперкрыс и взламывать суперсейфы с помощью предложенных ниже проектов. Как обычно, в случае сложных проектов вы предоставлены самим себе; никаких решений не предусмотрено.

Формирование крысиного гарема

Поскольку один самец крысы может спариваться с несколькими самками, то нет необходимости иметь равное число самцов и самок крыс. Перепишите программный код super_rats.py для размещения переменного числа самцов и самок. Затем повторите программу с таким же суммарным числом крыс, как и раньше, но используйте 4 самцов и 16 самок. Как это повлияет на количество лет, необходимых для достижения целевого веса 50 000 грамм?

Создание более эффективного взломщика сейфов

В настоящее время код safe_cracker.py написан так, что при отыскании совпадения запирающего колеса это совпадение явно не сохраняется. Пока работает цикл `while`, ничто не мешает тому, чтобы правильное совпадение было стохастически перезаписано. Измените код так, чтобы индексы правильных догадок были исключены из будущих изменений. Сравните время работы двух версий программного кода с целью оценки влияния этого изменения на результат.

8

ПОДСЧЕТ СЛОГОВ В СТИХОТВОРЕНИЯХ ХОККУ

Поэзия, возможно, является высшей формой литературы. Она, как выразился Кольридж, представляет собой "лучшие слова в лучшем порядке". Поэт обязан — с величайшей краткостью — рассказать историю, развить идею, описать сцену или вызвать бурю чувств, при этом подчиняясь строгим правилам рифмы и ритма, стиля и структуры.

Компьютеры любят правила и структуру и даже имеют потенциал к тому, чтобы вызывать эмоции. В своей книге 1996 г. Чарльз Хартман "Виртуальная муза: эксперименты с компьютерной поэзией" (Hartman C. Virtual muse: experiments in computer poetry) отметил ранние попытки написать алгоритмы, которые могли бы имитировать человеческую поэзию. Цитируя Хартмана, можно констатировать, что "сложность поэтического взаимодействия, хитрый танец между поэтом, текстом и читателем, вызывает игру колебаний. В этой игре правильно запрограммированный компьютер имеет шанс в некоторых интересных движениях проскочить мимо".

Ранние программы, описанные Хартманом, могли в лучшем случае порождать поэзию плохих битников. В то время цель состояла в том, чтобы "ввести в язык рассчитанные кусочки механизированной анархии, вернуть результаты обратно в мир, где живет язык, и посмотреть, как оседает пыль". Как мы уже говорили в нескольких главах, контекст является слабым звеном в программировании таких вещей, как анаграммы имен собственных и нулевые шифры. Для того чтобы писать компьютерные стихи, которые проходят "высшую" проверку литературы, игнорировать контекст не получится.

Сделать так, чтобы компьютер симулировал это самое человеческое из человеческих усилий, представляет собой интригующую задачу — и, конечно же, мы не

можем пройти мимо. В этой и следующей главах вы научите свой компьютер генерировать традиционную форму японской поэзии, именуемую *хокку* (или хайку).

Японская поэзия хокку

Хокку состоит из трех строк соответственно по пять, семь и пять слогов. Стихи редко рифмуются, и их предмет прямо или косвенно обычно вращается вокруг естественного мира — главным образом времен года. При надлежащей реализации хокку может погрузить вас в сцену, как будто вызывая воспоминания.

Я привел здесь три примера хокку. Первое стихотворение принадлежит Бусону (1715–1783), второе — Иссе (1763–1828), а третье — вашему покорному слуге на основе воспоминаний о детских дорожных приключениях.

Standing still at dusk
Listen... in far distances
The song of froglings!
— Buson

Застыв, я в сумерках
Прислушался... в дали
Поют лягушки!
— Бусон

Good friend grasshopper
Will you play the caretaker
For my little grave?
— Issa

Добрый друг кузнечик,
Не сыграешь ли ты в смотрителя
Могилки крохотной моей?
— Исса

Faraway cloudbanks
That I let myself pretend
Are distant mountains.
— Vaughan

Небесные гряды холодные,
Которые грезились мне
Как далекие горы.
— Воган

Из-за своего вызывающего воспоминания характера каждое хокку имеет для программиста встроенный "эксплуатируемый разрыв". Его прекрасно подытожил Питер Бейленсон в своей книге 1955 г. "Японская поэзия хокку" (Beilenson P. Japanese Haiku): "Хокку не всегда должно быть законченным или даже ясным утверждением. Читатель должен добавить к словам свои собственные ассоциации и образы и таким образом стать сотворцом своего собственного удовольствия от стихотворения". Хартман добавляет: "Ум читателя наиболее активно работает над редкими материалами. Мы рисуем самые ясные созвездия из наименьшего количества звезд. Таким образом, фактор бессмыслицы низок для крошечного словосочетания, которое может быть проникнуто образным значением". Проще говоря, труднее испортить короткое стихотворение. Читатели всегда исходят из того, что в словах поэта имелся смысл, и придумают его сами, если не могут найти.

Несмотря на это преимущество, натренировать ваш компьютер писать стихи — незаурядное достижение, и вам понадобится аж две главы на то, чтобы это сделать. В этой главе вы напишете программу, которая подсчитывает число слогов в словах и словосочетаниях ради соблюдения слоговой структуры хокку. *В главе 9* вы будете использовать технический прием, именуемый *анализом марковских цепей*, для

улавливания сути хокку — неуловимой вызывающей воспоминания компоненты — и преобразовывать существующие стихи во что-то новое и иногда, возможно, лучшее.

Проект 15: подсчет слогов

Подсчет слогов на английском языке сопряжен с трудностями. Как выразился Чарльз Хартман, проблема лежит в причудливой орфографии и запутанной лингвистической истории английского языка. Например, такое слово, как *aged* (престарелый, состаренный), может произноситься одним слогом либо двумя в зависимости от того, описывает ли оно человека или сыр. Как программе точно подсчитать слоги, не вырождаясь в бесконечный список частных случаев?

Ответ заключается в том, что она не сможет это сделать. По крайней мере, не без "шпаргалки". К счастью, эти шпаргалки существуют благодаря отрасли науки, именуемой *обработкой естественного языка* (ЕЯ), которая занимается взаимодействием между точным и структурированным языком компьютеров и тонким, часто неоднозначным "естественным" языком, используемым людьми. Примеры использования обработки ЕЯ включают машинный перевод с языка на язык, обнаружение спама, понимание запросов к поисковой системе и предсказательное распознавание текста для пользователей мобильных телефонов. Самое большое влияние обработки ЕЯ проявится еще впереди: глубинный анализ огромных объемов ранее непригодных, плохо структурированных данных и участие в безупречных разговорах с нашими компьютерными повелителями.

В этой главе вы будете использовать набор данных для обработки ЕЯ с целью подсчета слогов в словах или словосочетаниях. Вы также напишете код, который отыскивает слова, отсутствующие в этом наборе данных, а затем поможет вам построить вспомогательный словарь. Наконец, вы напишете программу, которая поможет вам проверить ваш код подсчета слогов. *В главе 9* вы будете использовать этот алгоритм подсчета слогов в качестве модуля в программе, которая поможет вам вычислительно произвести самое высокое достижение в литературе: поэзию.

Цель

Написать программу на языке Python, которая подсчитывает число слогов в английском слове или словосочетании.

Стратегия

Для вас и для меня подсчет слогов не представляет труда. Положите тыльную сторону ладони чуть ниже подбородка и начните говорить. Всякий раз, когда ваш подбородок касается руки, вы произносите слог. У компьютеров нет рук или подбородков, но каждый гласный звук представляет слог — и компьютеры могут подсчитывать гласные звуки. Однако это не так элементарно, поскольку для этого нет простого правила. Некоторые гласные в письменном языке являются немыми, на-

пример *e* в слове *like*, а другие объединяются, составляя один звук, например *oo* в слове *moo*. К счастью, число слов в английском языке не бесконечно. Существуют достаточно исчерпывающие списки, которые включают в себя подавляющую часть необходимой информации.

Корпус — это причудливое название, используемое для сборника текстов. В *главе 9* вы будете использовать тренировочный корпус, состоящий из хокку, который инструктирует язык программирования Python о том, как писать новое хокку. В этой главе вы будете использовать этот же корпус для извлечения числа слогов.

Ваш счетчик слогов должен оценивать и словосочетания, и отдельные слова, т. к. в конечном итоге вы будете использовать его для подсчета слогов в целых строках хокку. Программа на входе примет немного текста, подсчитает число слогов в каждом слове и вернет суммарное число слогов. Вам также придется учесть такие вещи, как знаки препинания, пробелы и отсутствующие слова.

Первостепенные шаги, которые необходимо выполнить, таковы:

1. Скачать большой корпус информации о количестве слогов.
2. Сравнить указанный корпус с тренировочным корпусом поэзии хокку и выявить все слова, отсутствующие в корпусе информации о количестве слогов.
3. Составить словарь отсутствующих слов и их количеств слогов.
4. Написать программу, которая использует корпус информации о количестве слогов и словарь отсутствующих слов с целью подсчета слогов в тренировочном корпусе.
5. Написать программу, которая проверяет программу подсчета слогов относительно обновлений тренировочного корпуса.

Использование корпуса

Естественно-языковой инструментарий Natural Language Toolkit (NLTK) — это популярный набор программ и библиотек для работы с данными человеческого языка в Python. Он был создан в 2001 г. в рамках курса компьютерной лингвистики на кафедре компьютерных и информационных наук в Университете Пенсильвании. Развитие и расширение продолжались с помощью десятков участников. Узнать о нем больше можно, посетив официальный сайт NLTK по адресу **http://www.nltk.org/**.

В этом проекте вы будете использовать NLTK для доступа к словарю произношения Университета Карнеги — Меллона (CMUdict). Указанный корпус содержит почти 125 тыс. слов в сопоставлении с их произношением. Он является машиночитаемым и пригоден для таких задач, как распознавание речи.

Установка NLTK

Инструкции по установке NLTK в UNIX, Windows и macOS можно найти по адресу **http://www.nltk.org/install.html**. Если вы используете Windows, то я предлагаю вам

начать с открытия окна командной строки либо интерактивной оболочки PowerShell и попытаться установить с помощью pip:

```
python -m pip install nltk
```

Проверить установку можно, открыв интерактивную оболочку Python и набрав:

```
>>> import nltk
>>>
```

Если вы не получите ошибку, то все готово. В противном случае следуйте инструкциям на только что процитированном веб-сайте.

Скачивание словаря произношения CMUdict

Для получения доступа к корпусу CMUdict (либо любому другому корпусу NLTK) вы должны его скачать. Это можно сделать с помощью удобной программы для скачивания NLTK. После того как вы установили NLTK, введите следующую команду в интерактивной оболочке Python:

```
>>> import nltk
>>> nltk.download()
```

Теперь должно открыться окно скачивания NLTK (рис. 8.1). Перейдите на вкладку **Corpora** (Корпусы) вверху, затем щелкните на **cmudict** в столбце **Identifier** (Идентификатор). Далее прокрутите вниз окна и назначьте каталог для скачиваний; я использовал принятый по умолчанию C:\nltk_data. Наконец, нажмите кнопку **Download** (Скачать), для того чтобы скачать словарь CMUdict.

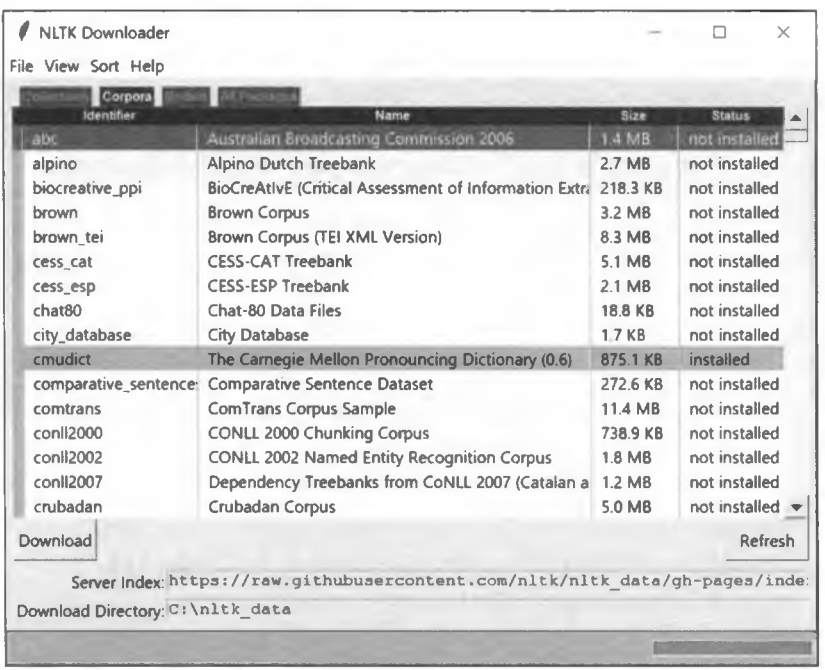


Рис. 8.1. Окно скачивания NLTK с выбранным для скачивания корпусом cmudict

Когда скачивание корпуса CMUdict завершится, выйдите из программы скачивания и введите в интерактивную оболочку Python следующее:

```
>>> from nltk.corpus import cmudict
>>>
```

Если вы не столкнулись с ошибкой, то корпус был успешно скачан.

Подсчет звуков вместо слогов

Корпус CMUdict разбивает слова на множества фонем — чувственно различных единиц звука в определенном языке — и помечает гласные для лексического ударения с помощью чисел (0, 1 и 2). Корпус CMUdict помечает каждую гласную одним и только одним из этих чисел, поэтому эти числа можно использовать для идентификации гласных в слове.

Взгляд на слова как на множество фонем поможет вам обойти несколько проблем. Во-первых, словарь произношения CMUdict не будет включать в письменное слово непроизносимые гласные. Например, вот как он видит слово *scarecrow* (пугало):

```
[['S', 'K', 'AE1', 'R', 'K', 'R', 'OW0']]
```

Каждый элемент с числовым суффиксом представляет произносимую гласную. Обратите внимание, что немое *e* в конце *scare* (пугать) правильно опущено.

Во-вторых, иногда многочисленные и расположенные подряд письменные гласные произносятся как одна фонема. Например, вот как словарь CMUdict представляет слово *house* (дом):

```
[['HH', 'AW1', 'S']]
```

Обратите внимание, как указанный корпус для целей произношения рассматривает письменные двойные гласные *ou* как одну гласную, `'AW1'`.

Обработка слов с несколькими произношениями

Как я уже упоминал в преамбуле к главе, некоторые слова имеют несколько разных произношений; слова *aged* (престарелый, состаренный) и *learned* (усвоенный, ученый) — это лишь два таких примера:

```
[['EY1', 'JH', 'D'], ['EY1', 'JH', 'IH0', 'D']]
[['L', 'ER1', 'N', 'D'], ['L', 'ER1', 'N', 'IH0', 'D']]
```

Обратите внимание на вложенные списки. Корпус признает, что оба слова могут произноситься одним или двумя слогами. Это означает, что для определенных слов он вернет более одного слога, и это вам придется учитывать в своем коде.

Управление отсутствующими словами

Корпус CMUdict очень полезен, но с ним слово либо есть, либо его нет. Потребовалось всего несколько секунд на то, чтобы найти более 50 слов, таких как *dewdrop*, *bathwater*, *dusky*, *ridgeline*, *storks*, *dragonfly*, *beggar* и *archways* (соответственно: ро-

синка, вода в ванне, сумеречный, гребневая линия, аисты, стрекоза, нищий и сводчатый проход), отсутствующих в корпусе CMUdict на тестовом примере из 1500 слов. Поэтому одна из ваших стратегий должна состоять в том, чтобы проверить словарь CMUdict на предмет отсутствующих слов, а затем устранить любые упущения, создав корпус для вашего корпуса!

Тренировочный корпус

В главе 9 вы будете использовать тренировочный корпус из нескольких сотен хокку для того, чтобы "научить" вашу программу писать новые поэтические миниатюры. Но вы не можете рассчитывать на то, что словарь CMUdict будет содержать все слова в этом корпусе, потому что некоторые из них будут японскими словами, такими как *sake* (саке́). И как вы убедились, в словаре CMUdict отсутствуют даже некоторые распространенные английские слова.

Поэтому первым делом следует проверить все слова в тренировочном корпусе на членство в CMUdict. Для этого вам нужно будет скачать тренировочный корпус train.txt с веб-сайта **https://www.nostarch.com/impracticalpython/**. Сохраните его в той же папке, что и все программы Python из этой главы. Указанный файл содержит чуть менее 300 стихотворений хокку, которые были случайно продублированы около 20 раз ради того, чтобы обеспечить надежный тренировочный набор.

После того как вы найдете слова, которые отсутствуют в словаре CMUdict, вы напишете сценарий, который поможет вам подготовить словарь Python, использующий слова в качестве ключей и количества слогов в качестве значений; затем вы сохраните этот словарь в файл, который может поддерживать CMUdict в программе подсчета слогов.

Код отыскания отсутствующих слов

Программный код этого раздела отыщет слова, отсутствующие в корпусе CMUdict, поможет подготовить словарь, состоящий из слов и количеств их слогов, а также сохранить словарь в файл. Указанный код можно скачать с веб-сайта по адресу **https://www.nostarch.com/impracticalpython/** как файл missing_words_finder.py.

Импорт модулей, загрузка корпуса CMUdict и определение функции *main()*

Листинг 8.1 импортирует модули, загружает корпус CMUdict и определяет функцию `main()`, которая выполняет программу.

Листинг 8.1. Импортирует модули, загружает CMUdict и определяет `main()`. Файл missing_words_finder.py, часть 1

```
import sys
from string import punctuation
❶ import pprint
```

```
import json
from nltk.corpus import cmudict

❷ cmudict = cmudict.dict() # Словарь произношения
                           # университета Карнеги - Мелона

❷ def main():
    ❹ haiku = load_haiku('train.txt')
    ❺ exceptions = cmudict_missing(haiku)
    ❻ build_dict = input("\nПостроить словарь исключений вручную (y/n)? \n")
      if build_dict.lower() == 'n':
          sys.exit()
      else:
       ❼ missing_words_dict = make_exceptions_dict(exceptions)
          save_exceptions(missing_words_dict)
```

Вы начинаете с нескольких уже знакомых команд импорта и нескольких новых. Модуль `pprint` позволяет "красиво" напечатать ваш словарь отсутствующих слов в структурном и удобном для чтения формате ❶. Вы запишете этот же словарь как долговременные данные с помощью объектной нотации JavaScript (`json`), т. е. текстового способа обмена данными между компьютерами, который хорошо работает со структурами данных Python; этот модуль является составной частью стандартной библиотеки, стандартно применяемой в многочисленных языках, при этом данные являются безопасными и удобочитаемыми. Завершите импортом корпуса CMUdict.

Далее вызовите метод `dict()` модуля `cmudict`, превратив корпус в словарь со словами в качестве ключей и их фонемами в качестве значений ❷.

Определите функцию `main()`, которая будет вызывать функции для загрузки тренировочного корпуса, отыскания отсутствующих в словаре CMUdict слов, построения словаря со словами и их количествами слогов и сохранения результатов ❸. Вы определите эти функции после определения `main()`.

Вызовите функцию для загрузки тренировочного корпуса хокку и назначьте возвращенное множество переменной `haiku` ❹. Затем вызовите функцию, которая отыщет отсутствующие слова и вернет их в виде множества ❺. Используя множества, вы автоматически удаляете дублирующие слова, которые вам не нужны. Функция `cmudict_missing()` также покажет число отсутствующих слов и некоторую другую статистику.

Теперь спросите пользователя, намерен ли он строить словарь вручную с целью решить вопрос с отсутствующими словами, и назначьте введенное им значение переменной `build_dict` ❻. Если он хочет остановиться, то выйдите из программы; в противном случае вызовите функцию, которая строит словарь ❼, а затем еще одну, которая сохраняет словарь на диск. Обратите внимание, что если пользователь хочет продолжить работу, то он не ограничивается нажатием клавиши <y>, хотя она и указана в подсказке.

Загрузка тренировочного корпуса и отыскание отсутствующих слов

Листинг 8.2 загружает и подготавливает тренировочный корпус, сравнивает его содержимое со словарем CMUdict и отслеживает различия. Эти задачи разделены между двумя функциями.

Листинг 8.2. Определяет функции загрузки корпуса и отыскивает слова, отсутствующие в словаре CMUdict. Файл missing_words_finder.py, часть 2

```
❶ def load_haiku(filename):
      """Открыть и вернуть тренировочный корпус хокку в виде множества."""
      with open(filename) as in_file:
       ❷ haiku = set(in_file.read().replace('-', ' ').split())
       ❸ return haiku

  def cmudict_missing(word_set):
      """Отыскать и вернуть слова в множестве слов,
         отсутствующих в cmudict."""
    ❹ exceptions = set()
      for word in word_set:
          word = word.lower().strip(punctuation)
          if word.endswith("'s") or word.endswith("’s"):
              word = word[:-2]
       ❺ if word not in cmudict:
              exceptions.add(word)
  print("\nИсключения:")
  print(*exceptions, sep='\n')
❻ print("\nЧисло уникальных слов в корпусе хокку = {}"
        .format(len(word_set)))
  print("Число слов в корпусе слов не из cmudict = {}"
        .format(len(exceptions)))
  membership = (1 - (len(exceptions) / len(word_set))) * 100
❼ print("членство в cmudict = {:.1f}{}".format(membership, '%'))
  return exceptions
```

Определите функцию, которая читает слова из тренировочного корпуса хокку ❶. Хокку в train.txt были продублированы много раз, плюс к тому оригинальные хокку содержат дубликаты слов, таких как *moon* (луна), *mountain* (гора) и *the*. Нет никакого смысла оценивать слово более одного раза, поэтому загрузите слова в виде множества, удалив повторы ❷. Также необходимо заменить дефисы пробелами. Дефисы весьма популярны в хокку, но для того чтобы выполнить проверку на их наличие в корпусе CMUdict, вам нужно отделить слова с обеих сторон. Завершите функцию, вернув множество `haiku` ❸.

Теперь самое время отыскать пропущенные слова. Определите функцию `cmudict_missing()`, которая в качестве аргумента принимает последовательность — в дан-

ном случае множество слов, возвращаемых функцией `load_haiku()`. Создайте пустое множество `exceptions`, в котором будут храниться все отсутствующие слова ❼. Переберите каждое слово в множестве `haiku`, конвертируя его в нижний регистр и удаляя любые начальные или замыкающие знаки препинания. Обратите внимание, что вы не удаляете их внутреннюю пунктуацию, кроме дефисов, потому что словарь CMUdict распознает слова, такие как *wouldn't*. Притяжательные слова, как правило, не находятся в корпусе, поэтому удалите замыкающие *'s*, т. к. они не повлияют на количество слогов.

ПРИМЕЧАНИЕ

Будьте осторожны с фигурными апострофами (’), создаваемыми программным обеспечением для обработки текстов. Они отличаются от прямых апострофов ('), используемых в простых текстовых редакторах и интерактивных оболочках и не могут быть распознаны словарем CMUdict. Если вы добавляете новые слова в тренировочные или JSON-файлы, то в сокращениях или притяжательных существительных непременно используйте прямой апостроф.

Если слово в CMUdict не найдено, то добавьте его в множество исключений `exceptions` ❺. Напечатайте эти слова в качестве проверки вместе с некоторой базовой информацией ❻, например числом уникальных слов, числом отсутствующих слов и процентом слов тренировочного корпуса, являющихся членами CMUdict. Установите процентное значение точности равным одному десятичному знаку ❼. Завершите работу функции, вернув множество исключений `exceptions`.

Построение словаря отсутствующих слов

Листинг 8.3 продолжает код скрипта missing_words_finder.py, теперь дополняя корпус CMUdict посредством назначения количеств слогов отсутствующим словам в качестве значений словаря Python. Поскольку число отсутствующих слов должно быть относительно небольшим, пользователь может задать количества вручную; поэтому напишите код, помогающий ему взаимодействовать с программой.

Листинг 8.3. Позволяет пользователю вручную подсчитывать слоги и создавать словарь. Файл missing_words_finder.py, часть 3

```
❶ def make_exceptions_dict(exceptions_set):
      """Вернуть словарь слов и количеств слогов из множества слов."""
   ❷ missing_words = {}
      print("Введите число слогов в слове. Ошибки можно исправить в конце. \n")
      for word in exceptions_set:
          while True:
           ❸ num_sylls = input("Введите число слогов в {}: ".
                               format(word))
           ❹ if num_sylls.isdigit():
                 break
             else:
                 print("                    Недопустимый ответ!",
                       file=sys.stderr)
```

```
        ❺ missing_words[word] = int(num_sylls)
    print()
❻ pprint.pprint(missing_words, width=1)

❼ print("\nВнести изменения в словарь перед сохранением?")
    print("""
    0 - Выйти и сохранить
    1 - Добавить слово либо изменить количество слогов
    2 - Удалить слово
    """)

❽ while True:
        choice = input("\nСделайте свой выбор: ")
        if choice == '0':
            break
        elif choice == '1':
            word = input("\nДобавляемое или изменяемое слово: ")
            missing_words[word]=int(input("Введите число слогов в {}: "
                                          .format(word)))
        elif choice == '2':
            word = input("\nВведите удаляемое слово: ")
         ❾ missing_words.pop(word, None)

    print("\nИзменения в новых словах и слогах:")
❿ pprint.pprint(missing_words, width=1)

    return missing_words
```

Начните с определения функции, которая в качестве аргумента принимает множество исключений `exceptions`, возвращаемых функцией `cmudict_missing()` ❶. Сразу же назначьте переменной с именем `missing_words` пустой словарь ❷. Сообщите пользователю, что если он допустит ошибку, то у него будет шанс исправить ее позже; затем примените циклы `for` и `while` для того, чтобы перебрать множество отсутствующих слов и показать каждое слово пользователю, на входе запрашивая число слогов. Слово будет ключом словаря, а переменная `num_sylls` станет его значением ❸. Если введенное значение является цифрой ❹, выйдите из цикла. В противном случае предупредите пользователя и снова в цикле `while` попросите ввести данные. Если входные данные пройдут проверку, добавьте значение в словарь в виде целого числа ❺.

Примените `pprint` для вывода каждой пары "ключ — значение" на экран в отдельной строке для проверки. Параметр `width` действует как аргумент новой строки ❻.

Дайте пользователю возможность внести последние изменения в словарь `missing_words` перед сохранением его в виде файла ❼. Примените тройные кавычки для показа пользователю меню вариантов, а затем цикл `while` для поддержки вариантов активными до тех пор, пока пользователь не будет готов сохранить сло-

варь ❷. Вот эти три варианта: **Выйти** — вариант вызывает команду `break`; **Добавить слово либо изменить количество слогов** существующего слова — требует на входе указать слово и количество слов; **Удалить слово** — удаляет запись, используя метод `pop()` из словарного типа ❾. Добавление аргумента `None` в `pop()` означает, что программа не вызовет системное исключение `KeyError`, если пользователь введет слово, которого в словаре нет.

Завершите программу, позволив пользователю последний раз взглянуть на словарь, в случае если были внесены изменения ❿, а затем верните его.

Сохранение словаря отсутствующих слов

Долговременные данные — это данные, которые сохраняются после завершения программы. Для того чтобы сделать словарь `missing_words` доступным для использования в программе count_syllables.py, которую вы напишете в этой главе позже, его нужно сохранить в файл. Листинг 8.4 как раз это и делает.

Листинг 8.4. Сохраняет словарь отсутствующих слов в файл и вызывает `main()`. Файл missing_words_finder.py, часть 4

```
❶ def save_exceptions(missing_words):
      """Сохранить словарь исключений exceptions как файл json."""
   ❷ json_string = json.dumps(missing_words)
   ❸ f = open('missing_words.json', 'w')
     f.write(json_string)
     f.close()
   ❹ print("\nФайл сохранен как missing_words.json")

❺ if __name__ == '__main__':
     main()
```

Для сохранения словаря используйте модуль `json`. Определите новую функцию, которая в качестве аргумента принимает множество отсутствующих слов ❷. Назначьте словарь `missing_words` новой переменной с именем `json_string` ❷; затем откройте файл с расширением .json ❸, запишите на диск переменную `json_string` и закройте файл. Покажите имя файла пользователю в качестве напоминания ❹. Завершите фрагментом кода, который позволяет программе работать как модуль либо в автономном режиме ❺.

Метод `json.dumps()` сериализует словарь `missing_words` в символьную цепочку. *Сериализация* — это процесс конвертации данных в более удобный для передачи либо хранения формат. Например:

```
>>> import json
>>> d = {'scarecrow': 2, 'moon': 1, 'sake': 2}
>>> json.dumps(d)
'{"sake": 2, "scarecrow": 2, "moon": 1}'
```

Обратите внимание, что сериализованный словарь с обеих сторон ограничен одинарными кавычками, что и делает его цепочкой символов.

Здесь показан частичный вывод из скрипта missing_words_finder.py. Список отсутствующих слов вверху и ручное количество слогов внизу для краткости были сокращены.

```
--обрезано--
froglings
scatters
paperweights
hibiscus
cumulus
nightingales

Число уникальных слов в корпусе хокку = 1523
Число слов в корпусе не из cmudict = 58
членство в cmudict = 96.2%

Построить словарь исключений вручную (у/п)?
y
Введите число слогов в woodcutter: 3
Введите число слогов в morningglory: 4
Введите число слогов в cumulus: 3
--обрезано--
```

Не волнуйтесь — вам не придется самим назначать все количества слогов. Файл missing_words.json завершен и готов к скачиванию в любой момент, когда он вам потребуется.

> **ПРИМЕЧАНИЕ**
>
> В случае слов, которые имеют несколько произношений, таких как *jagged* (зазубренный) или *our* (наш), вы можете вынудить программу использовать то, которое вы предпочитаете, вручную открыв файл missing_words.json и добавив пару "ключ — значение" (в любом месте, т. к. словари не упорядочены). Я сделал это со словом *sake* (саке́), для того чтобы программа использовала двухсложное японское произношение. Поскольку в этом файле в первую очередь проверяется членство слова, программа переопределит значение словаря CMUdict.

Теперь, когда вы уладили уязвимости в корпусе CMUdict, вы готовы написать программный код, который подсчитывает слоги. *В главе 9* вы будете использовать указанный код в качестве модуля программы markov_haiku.py.

Код подсчета слогов

Этот раздел содержит код программы count_syllables.py. Вам также понадобится файл missing_words.json, созданный в предыдущем разделе. Оба файла можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**. Сохраните их вместе в одной папке.

Подготовка, загрузка и подсчет

Листинг 8.5 импортирует необходимые модули, загружает словарь CMUdict и словарь отсутствующих слов и определяет функцию, которая будет подсчитывать слоги в данном слове или словосочетании.

Листинг 8.5. Импортирует модули, загружает словари и подсчитывает слоги. Файл count_syllables.py, часть 1

```
import sys
from string import punctuation
import json
from nltk.corpus import cmudict

# Загрузить словарь слов из корпуса хокку, но не из cmudict
with open('missing_words.json') as f:
    missing_words = json.load(f)

❶ cmudict = cmudict.dict()

❷ def count_syllables(words):
    """Использовать корпусы для подсчета слогов в английском слове
       или словосочетании."""
    # подготовить слова для корпуса cmudict
    words = words.replace('-', ' ')
    words = words.lower().split()
  ❸ num_sylls = 0
  ❹ for word in words:
        word = word.strip(punctuation)
        if word.endswith("'s") or word.endswith("’s"):
            word = word[:-2]
      ❺ if word in missing_words:
            num_sylls += missing_words[word]
        else:
          ❻ for phonemes in cmudict[word][0]:
                for phoneme in phonemes:
              ❼ if phoneme[-1].isdigit():
                    num_sylls += 1
  ❽ return num_sylls
```

После нескольких уже знакомых импортов загрузите файл missing_words.json, содержащий все слова и их количества слогов, отсутствующие в корпусе CMUdict. Использование `json.load()` восстанавливает словарь, сохраненный в виде символьной цепочки. Затем превратите корпус CMUdict в словарь, используя метод `dict()` ❶.

Определите функцию `count_syllables()`, которая будет подсчитывать слоги. Она должна принимать как слова, так и словосочетания, потому что в конечном итоге вы захотите передавать ей строки из хокку. Подготовьте слова, как вы делали ранее в программе missing_words_finder.py ❷.

Задайте переменную `num_sylls` для хранения количества слогов и установите ее равной `0` ❸. Теперь начните перебирать входные слова в цикле, удаляя знаки препинания и *'s* с обоих концов. Обратите внимание, что вы можете запутаться в формате апострофа, поэтому предоставлены две версии: одна с прямым апострофом и одна с фигурным апострофом ❹. Затем проверьте, является ли слово членом небольшого словаря отсутствующих слов. Если слово найдено, то добавьте значение из словаря для этого слова в `num_sylls` ❺. В противном случае начните просматривать фонемы, которые представляют значение в CMUdict; по каждой фонеме просмотрите цепочки символов, которые его составляют ❻. Если в конце цепочки вы найдете цифру, это указывает, что данная фонема является гласной. Если проиллюстрировать использование слова *aged*, то только первая цепочка (здесь выделенная жирным шрифтом) заканчивается цифрой, поэтому указанное слово содержит одну гласную:

```
[['EY1', 'JH', 'D'], ['EY1', 'JH', 'IH0', 'D']]
```

Обратите внимание, что вы используете первое значение (`[0]`) в случае, если имеется несколько произношений; помните, что корпус CMUdict представляет каждое произношение во вложенном списке. Это может привести к случайной ошибке, т. к. правильный выбор будет зависеть от контекста.

Проверьте, имеет ли конец фонемы цифру, и если имеет, то добавьте 1 в `num_sylls` ❼. Наконец, верните общее количество слогов в слове или словосочетании ❽.

Определение функции *main()*

Завершая программу, листинг 8.6 определяет и запускает функцию `main()`. Программа вызовет эту функцию при запуске программы в автономном режиме, например для выборочной проверки слова или фразы, но она не будет вызвана, если вы импортируете `syllable_counter` в качестве модуля.

Листинг 8.6. Определяет и вызывает функцию `main()`. Файл count_syllables.py, часть 2

```
def main():
❶ while True:
      print("Счетчик слогов")
  ❷ word = input("Введите слово или словосочетание либо нажмите Enter,
                                                    чтобы выйти: ")
  ❸ if word == '':
        sys.exit()
  ❹ try:
        num_syllables = count_syllables(word)
```

```
            print("число слогов в {} равно: {}"
                  .format(word, num_syllables))
            print()
        except KeyError:
            print("Слово не найдено. Попробуйте еще раз.\n",
                  file=sys.stderr)

❺ if __name__ == '__main__':
      main()
```

Определите функцию `main()`, а затем начните цикл `while` ❶. Попросите пользователя ввести слово или словосочетание ❷. Если пользователь нажимает клавишу <Enter> без ввода, программа завершает работу ❸. В противном случае начните блок `try-except`, благодаря которому программа не зависнет, если пользователь введет слово, не найденное ни в одном словаре ❹. Исключение должно быть вызвано только в автономном режиме, т. к. вы уже подготовили программу, которая будет выполняться на тренировочном корпусе хокку без исключений. В этом блоке вызывается функция `count_syllables()`, ей передаются входные данные, а затем результаты показываются в интерактивной оболочке. Завершите работу стандартным кодом, который позволяет программе работать автономно либо как модуль в другой программе ❺.

Программа проверки вашей программы

Вы тщательно приспособили программу подсчета слогов, обеспечив ее работу с тренировочным корпусом. По мере продолжения работы с программой хокку вы можете добавить в этот корпус одно или два стихотворения, но добавление нового хокку может внести новое слово, которого нет ни в словаре CMUdict, ни в вашем словаре исключений. Прежде чем вернуться и перестроить словарь исключений, проверьте, действительно ли это необходимо.

Листинг 8.7 автоматически подсчитает слоги в каждом слове вашего тренировочного корпуса и покажет любое слово (слова), на котором проверка оказалась безуспешной. Эту программу можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/** как файл test_count_syllables_w_full_corpus.py. Сохраните его в той же папке, что и файлы count_syllables.py, train.txt и missing_words.json.

Листинг 8.7. Пытается подсчитать слоги в словах в тренировочном корпусе и перечисляет все безуспешные попытки. Файл test_count_syllables_w_full_corpus.py

```
import sys
import count_syllables

with open('train.txt.') as in_file:
  ❶ words = set(in_file.read().split())
```

```
❷ missing = []

❸ for word in words:
      try:
          num_syllables = count_syllables.count_syllables(word)
          ##print(word, num_syllables, end='\n') # раскомментируйте для
                                                 # показа количеств слов
    ❹ except KeyError:
          missing.append(word)

❺ print("Отсутствующие слова:", missing, file=sys.stderr)
```

Откройте обновленный тренировочный корпус train.txt и загрузите его в виде множества, удалив дубликаты ❶. Создайте пустой список с именем `missing`, который будет содержать любые новые слова, для которых невозможно подсчитать слоги ❷. Слова в `missing` не будут ни в словаре CMUdict, ни в вашем словаре `missing_words`.

Переберите в цикле слова из нового тренировочного корпуса ❸ и примените блок `try-except` для обработки системного исключения `KeyError`, который будет вызван, если count_syllables.py не сможет отыскать слово ❹. Добавьте это слово в отсутствующий список, а затем покажите список ❺.

Если программа показывает пустой список, то все слова в новом хокку уже присутствуют либо в CMUdict, либо в missing_words.json, поэтому вносить никаких изменений не требуется. В противном случае у вас есть возможность вручную добавить слова в файл missing_words.json либо повторно выполнить скрипт missing_words_finder.py для того, чтобы перестроить файл missing_words.json.

Резюме

В этой главе вы узнали, как скачать модуль NLTK и использовать один из его наборов данных, словарь произношения университета Карнеги — Меллона (CMUdict). Вы сверили набор данных CMUdict с тренировочным корпусом хокку и построили вспомогательный словарь Python с любыми отсутствующими словами. Вы сохранили этот словарь Python как долговременные данные с помощью объектной нотации JavaScript (JSON). Наконец, вы написали программу, которая умеет подсчитывать слоги. В *главе 9* вы воспользуетесь своей программой подсчета слогов в качестве вспомогательного средства при генерировании новаторских стихотворений хокку.

Дальнейшее чтение

Книга Чарльза О. Хартмана "Виртуальная муза: эксперименты с компьютерной поэзией" (Hartman C. O. Virtual muse: experiments in computer poetry. Wesleyan University Press, 1996) представляет собой увлекательный взгляд на раннее сотрудничество людей и компьютеров при написании стихотворений.

Книга Стивена Берда, Эвана Клейна и Эдварда Лопера "Обработка естественного языка с помощью Python: анализ текста с помощью естественно-языкового инструментария NLTK" (Bird S., Klein E., Loper E. Natural language processing with Python: analyzing text with the natural language toolkit. O'Reilly, 2009) представляет собой доступное введение в обработку ЕЯ с использованием языка Python с большим количеством упражнений и полезной интеграцией с веб-сайтом NLTK. Новая версия книги, обновленная до Python 3 и NLTK 3, доступна в Интернете по адресу **http://www.nltk.org/book/**.

Статья Стивена Ф. Деанджелиса "Растущая важность обработки естественного языка" (DeAngelis S. F. The growing importance of natural language processing) из журнала Wired посвящена расширяющейся роли обработки ЕЯ в Больших данных. Ее онлайновая версия доступна по адресу **https://www.wired.com/insights/2014/02/growing-importance-natural-language-processing/**.

Практический проект: счетчик слогов против файла словаря

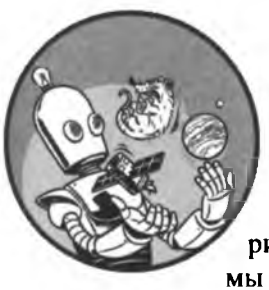
Напишите программу на языке Python, которая позволит вам протестировать скрипт count_syllables.py (либо любой другой программный код Python для подсчета слогов) относительно файла словаря. Разрешив пользователю указать число слов для проверки, выберите слова наугад и покажите список, состоящий из слов и их количеств слогов на отдельных строках. Результат должен выглядеть аналогично приведенному далее выводу:

```
ululation 4
intimated 4
sand 1
worms 1
leatherneck 3
contenting 3
scandals 2
livelihoods 3
intertwining 4
beaming 2
untruthful 3
advice 2
accompanying 5
deathly 2
hallos 2
```

Скачиваемые файлы словарей перечислены в табл. 2.1. Решение указанной задачи можно найти в приложении к книге либо скачать с веб-сайта **https://www.nostarch.com/impracticalpython/** как файл test_count_syllables_w_dict.py.

9

НАПИСАНИЕ СТИХОТВОРЕНИЙ ХОККУ С ПОМОЩЬЮ АНАЛИЗА МАРКОВСКИХ ЦЕПЕЙ



Компьютеры способны писать стихи путем перекомпоновки существующих стихотворений. Это, в сущности, то, что делают люди. Мы с вами не придумали язык, на котором говорим, — мы его выучили. Для того чтобы говорить или писать, мы просто рекомбинируем существующие слова — и редко по-настоящему в оригинальной манере. Как выразился Стинг о написании музыки, "не думаю, что в поп-музыке есть такая вещь, как композиция. Мне кажется, то, что мы делаем в поп-музыке, называется подборкой... Я — хороший подборщик".

В этой главе вы собираетесь написать программу, которая ставит "наилучшие слова в наилучшем порядке" в виде хокку. Но для этого Python нуждается в хороших примерах, поэтому вам нужно будет предоставить тренировочный корпус хокку японских поэтов.

В целях более или менее осмысленной перекомпоновки этих слов вы будете использовать *марковские цепи*, названные так в честь русского математика Андрея Маркова. *Анализ марковских цепей*, являющийся важной частью теории вероятностей, — это процесс, который пытается предсказать последующее состояние на основе свойств текущего состояния. Современные приложения указанного метода включают распознавание речи и рукописного текста, оценивание производительности компьютера, фильтрацию спама и алгоритм PageRank компании Google для поиска в Интернете.

С помощью анализа марковских цепей, тренировочного корпуса и программы подсчета слогов из *главы 8* вы сможете создавать новые хокку, которые следуют слоговым правилам этого жанра и в значительной степени остаются "в теме". Вы также узнаете, как использовать модуль журналирования Python `logging`, помогающий

осуществлять мониторинг поведения вашей программы с легкой нерегулярной обратной связью. И в *разд. "Сложные проекты" далее в этой главе* вы сможете заинтриговать своих друзей из социальных сетей тем, чтобы испытать их литературные навыки и попробовать отличить ваше симулированное хокку от реального.

Проект 16: анализ марковских цепей

Как и генетические алгоритмы в *главе 7*, анализ марковских цепей звучит внушительно, но его легко реализовать. Вы делаете это каждый день. Если вы слышите, как кто-то говорит: "Это элементарно, любезный...", то автоматически думаете: "Ватсон". Всякий раз, когда ваш мозг слышал эту фразу, он брал образец. Основываясь на серии образцов, он может предсказывать ответ. С другой стороны, если вы слышали, как кто-то сказал: "Хочу сходить...", то вы можете подумать: "в ванную комнату" либо "в кино", но, вероятно, не "в Хоума, Луизиана". Тут есть много возможных решений, но некоторые из них более вероятны, чем другие.

Еще в 1940-х годах Клод Шеннон (Claude Shannon) впервые использовал марковские цепи для статистического моделирования последовательности букв в тексте. Например, для каждого появления диграммы *th* в англоязычной книге следующей наиболее вероятной буквой является буква *e*.

Но вы не просто хотите знать, какая буква является наиболее вероятной; вам требуется знать фактическую вероятность получения этой буквы, а также шансы получения любой другой буквы, что представляет собой задачу, идеально подходящую для компьютера.

Для того чтобы решить эту задачу, вам нужно сопоставить каждую двухбуквенную диграмму во фрагменте текста с буквой, которая следует сразу за ней. Эта задача является классическим словарным приложением, где диграммы выступают в качестве ключей, а буквы в качестве значений.

Применительно к буквам в словах, *марковская модель* — это математическая модель, которая вычисляет вероятность появления буквы на основе предыдущих *k* букв подряд, где *k* — это целое число. *Модель порядка 2* означает, что вероятность появления буквы зависит от двух букв, которые ей предшествуют. *Модель порядка 0* означает, что каждая буква является независимой. И эта же логика применима к словам. Рассмотрим два примера хокку.

A break in the clouds	В разрыве облаков
The moon a bright mountaintop	Луна сияет, как вершина гор
Distant and aloof	Далекая и чуждая
Glorious the moon	Луна — великолепна
Therefore our thanks dark clouds come	И наша благодарность грядам облаков
To rest our tired necks	За успокоенье шеям нашим утомленным

Словарь Python, который увязывает каждое слово хокку с каждым последующим словом, выглядит следующим образом:

```
'a': ['break', 'bright'],
'aloof': ['glorious'],
'and': ['aloof'],
'break': ['in'],
'bright': ['mountaintop'],
'clouds': ['the', 'come'],
'come': ['to'],
'dark': ['clouds'],
'distant': ['and'],
'glorious': ['the'],
'in': ['the'],
'moon': ['a', 'therefore'],
'mountaintop': ['distant'],
'our': ['thanks', 'tired'],
'rest': ['our'],
'thanks': ['dark'],
'the': ['clouds', 'moon', 'moon'],
'therefore': ['our'],
'tired': ['necks'],
'to': ['rest']
```

Поскольку существует только два хокку, то большинство ключей словаря имеют только одно значение. Но посмотрите на нижнюю часть списка: слово *moon* (луна) появляется дважды. Это происходит потому, что марковская модель хранит каждое вхождение слова как отдельное, повторное значение. И поэтому если для ключа *the* вы выбираете значение наугад, то шансы выбора слова *moon* вместо слова *clouds* равны 2:1. И наоборот, модель автоматически отсеивает крайне редкие или невозможные комбинации. Например, за одним словом *the* потенциально могут следовать многие слова, но только не за еще одним словом *the*!

Приведенный далее словарь увязывает каждую *пару* слов с последующим словом, а это означает, что мы имеем модель порядка 2.

```
'a break': ['in'],
'a bright': ['mountaintop'],
'aloof glorious': ['the'],
'and aloof': ['glorious'],
'break in': ['the'],
'bright mountaintop': ['distant'],
'clouds come': ['to'],
'clouds the': ['moon'],
'come to': ['rest'],
'dark clouds': ['come'],
'distant and': ['aloof'],
'glorious the': ['moon'],
'in the': ['clouds'],
'moon a': ['bright'],
'moon therefore': ['our'],
```

```
'mountaintop distant': ['and'],
'our thanks': ['dark'],
'our tired': ['necks'],
'rest our': ['tired'],
'thanks dark': ['clouds'],
'the clouds': ['the'],
'the moon': ['a', 'therefore'],
'therefore our': ['thanks'],
'to rest': ['our']
```

Обратите внимание, что увязывание переходит от первого хокку ко второму, поэтому словарь содержит элементы `'and aloof': ['glorious']` и `'aloof glorious': ['the']`. Такое поведение означает, что ваша программа может перепрыгивать от одного хокку к другому и не ограничивается только парами слов в одном хокку. Этот процесс может свободно образовывать новые пары слов, которые поэты, возможно, никогда не задумывали.

Ввиду очень короткого тренировочного корпуса словарная пара *the moon* является единственной, имеющей несколько ключей. Для всех остальных вы "заперты" в одном результате. В этом примере размер тренировочного корпуса в большой степени определяет число значений в расчете на ключ, но при наличии у нас более крупного корпуса значение *k* в марковской модели будет иметь более сильное влияние.

Размер *k* определяет, несете ли вы чушь, демонстрируете плагиат или же создаете вразумительный образец оригинальности. Если *k* равно 0, то вы будете выбирать слова случайно на основе совокупной частоты этого слова в корпусе и, вероятно, произведете много тарабарщины. Если *k* является большим, то результаты будут жестко ограничены, и вы начнете воспроизводить тренировочный текст дословно. Таким образом, малые значения *k* способствуют творчеству, а большие значения ведут к дублированию. Задача состоит в том, чтобы найти баланс между ними.

Для иллюстрации, если применить марковскую модель порядка 3 к предыдущим хокку, то все полученные ключи будут иметь одно значение. Два значения, связанные со словарной парой *the moon*, теряются, потому что эта пара слов становится двумя ключами, каждый с уникальным значением:

```
'the moon a': ['bright'],
'the moon therefore': ['our']
```

Поскольку хокку являются короткими — всего 17 слогов длиной и имеющиеся тренировочные корпуса являются относительно малыми, использование *k*, равного 2, должно быть достаточным для обеспечения некоторого порядка, при этом по-прежнему допуская в вашей программе творческие замены слов.

Цель

Написать программу, которая генерирует хокку с помощью анализа марковских цепей. Дать пользователю возможность изменять хокку, независимо регенерируя вторую и третью строки.

Стратегия

Ваша общая стратегия симулирования хокку будет заключаться в построении марковских моделей порядков 1 и 2 с помощью тренировочного корпуса стихотворений хокку, написанных людьми. Затем вы примените указанные модели и программу count_syllables.py из *главы 8* для создания новых хокку, которые отвечают требуемой слоговой структуры 5–7–5 трех строк хокку.

Программа должна строить хокку по одному слову за раз, инициируя хокку случайным словом, взятым из корпуса, выбирая второе слово хокку с использованием марковской модели порядка 1, а затем выбирая каждое последующее слово с использованием модели порядка 2.

Каждое слово выводится из префикса, т. е. слова или пары слов, которые определяют, какое слово будет выбрано дальше в хокку; ключ в словарях сочетания слов представляет этот *префикс*. И следовательно, слово, которое префикс определяет, является *суффиксом*.

Выбор и отбрасывание слов

Когда программа отбирает слово, она сначала подсчитывает слоги в слове, и если слово не подходит, то она выбирает новое слово. Если в стихотворении нет возможных слов, основанных на префиксе, то программа прибегает к тому, что я называю *призрачным префиксом*, т. е. префиксу, который в хокку не встречается. Например, если словарной парой в хокку является *temple gong* (храмовый гонг) и все последующие слова в марковской модели имеют слишком много слогов для того, чтобы завершить строку по правилам, то программа случайно отбирает новую пару слов и использует ее для отбора следующего слова в хокку. Новый префикс словарной пары *не должен включаться в строку*, т. е. словарная пара *temple gong* заменена не будет. Хотя новое подходящее слово можно выбрать несколькими способами, я предпочитаю именно этот технический прием, потому что он позволяет упростить, поддерживая единообразный процесс на протяжении всей программы.

Эти шаги можно выполнить с помощью функций, показанных на рис. 9.1 и 9.2. Если исходить из того, что вы работаете на пятисложной строке, то рис. 9.1 является примером того, что произойдет на высоком уровне, если все выбранные слова соответствуют слоговой цели.

Программа случайно отбирает начальное слово *the* из корпуса, а затем подсчитывает его слоги. Далее оно выбирает слово *bright* (яркий) из модели порядка 1, основываясь на префиксе *the*. Затем она подсчитывает число слогов в строке и добавляет это число в число слогов в строке. Поскольку сумма слогов не превышает пяти, программа добавляет *bright* в строку, переходит к отбору слова *autumn* (осень) из модели порядка 2 на основе префикса *the bright*, а затем повторяет процесс подсчета слогов. Наконец, программа отбирает слово *moon* на основе префикса *bright autumn*, подсчитывает слоги и — поскольку суммарное число слогов в строке равно пяти — добавляет слово *moon* в строку, завершая ее.

Рис. 9.1. Высокоуровневый графический псевдокод для пятисложной строки хокку

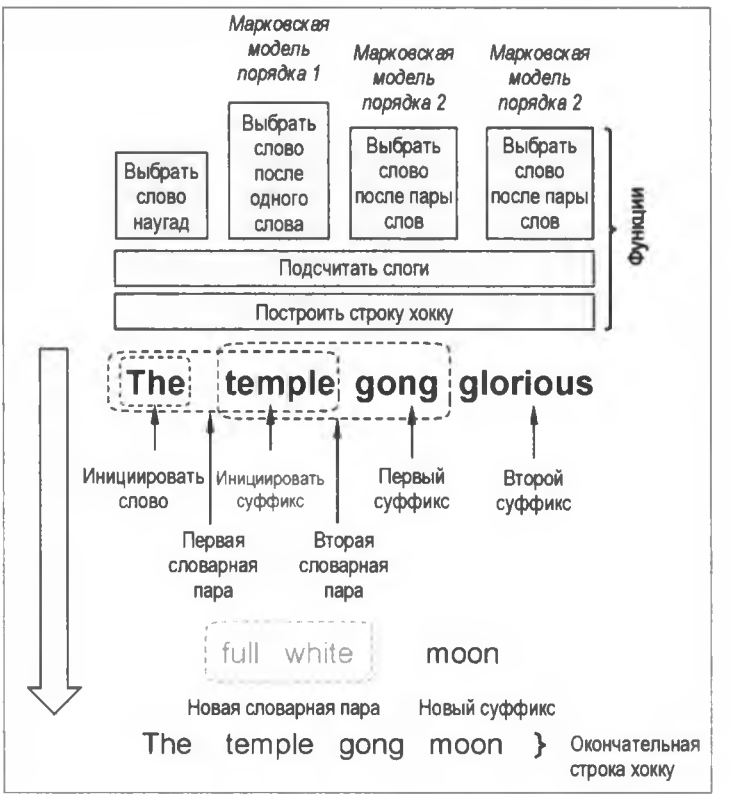


Рис. 9.2. Выбор нового суффикса со случайно отобранным призрачным префиксом ("full white" — иссиня-белый)

На рис. 9.2 представлен случай, когда для успешного завершения пятисложной строки программа должна задействовать призрачный префикс.

Будем считать, что единственным словом, которое следует за префиксом *temple gong* в марковской модели, является *glorious* (великолепный, славный). Это слово имеет слишком много слогов для того, чтобы уместиться в строке, поэтому программа наугад отбирает призрачный префикс — *full white*. Слово *moon* следует за призрачным префиксом и удовлетворяет оставшемуся числу слогов в строке, поэтому программа добавляет его в строку. Затем программа отбрасывает префикс *full white*, и на этом строка завершается. С помощью этого приема префиксации вы не можете гарантировать, что контекстуально новый суффикс будет иметь смысл, но в то же время он является одним из способов включить в процесс творчество.

Переход от одной строки к другой

Марковская модель — это "особый соус", который позволяет наполнить хокку контекстом и смыслом, продолжающимися от одной строки к другой. Японские поэты *обычно* писали стихотворения хокку, в котором каждая строка является отдельной фразой, но контекстуальная нить перетекает от строки к строке, как в этом хокку Бон Чо:

In silent midnight	В тихую полночь
Our old scarecrow topples down	Старое пугало падает вниз
Weird hollow echo	Странное эхо глухое
— Bon Cho	*— Бон Чо*

Хотя поэты предпочитали, чтобы каждая строка хокку представляла собой законченную мысль, они не следовали этому правилу строго. Вот пример из хокку Бусона:

My two plum trees are	Мои два сливовых дерева
So gracious see, they flower	Так милостивы, смотри, они цветут
One now, one later	Одно сейчас, другое позже
— Buson	*— Бусон*

Первая строка хокку Бусона не является грамматической сама по себе, поэтому читатель должен перейти к следующей строке, не прерываясь. Когда в стихосложении поэтическая фраза движется от одной строки к другой без паузы или синтаксического разрыва, это называется *анжамбеманом* (enjambment; от фр. *enjamber* — перешагивать, перепрыгивать), или перетеканием. По словам Чарльза Хартмана, автора книги "Виртуальная муза", анжамбеман — это то, что придает метрическим строкам значительную часть их гибкой подвижности. И это хорошо, т. к. очень трудно получить алгоритм написания связного стихотворения без некоторого грамматического перетекания от строки к строке. Для того чтобы побудить вашу программу продолжать "мысль" по нескольким строкам, вам нужно использовать словарную пару с конца предыдущей строки в качестве начального префикса для текущей строки.

Наконец, вы должны дать пользователю возможность не только строить стихотворение, но и редактировать его в интерактивном режиме, регенерируя вторую и третью строки. Написание стихотворений в большинстве своем состоит из переписывания, и было бы бессовестно оставлять пользователя в зависшем положении с двумя идеальными строками и не давать ему возможности снова бросить кости на несговорчивую строку.

Псевдокод

Если вы следуете стратегии, которую я только что изложил, то ваш высокоуровневый псевдокод должен выглядеть следующим образом:

```
Импортировать модуль count_syllables
Загрузить текстовый файл тренировочного корпуса
Обработать тренировочный корпус на предмет пробелов, разрывов новой строки и т. д.
Сопоставить каждое слово в корпусе со следующим словом (марковская модель порядка 1)
Сопоставить каждую словарную пару в корпусе со следующим словом (марковская модель порядка 2)
Дать пользователю возможность выбрать: генерировать полное хокку, переделать строки 2 или 3, либо выйти
из программы
Если это первая строка:
  Целевое количество слогов = 5
  Получить случайное слово из корпуса <= 4 слогов (однословные строки запрещены)
  Добавить слово в строку
  Задать префиксную переменную равной случайному слову
  Получить увязанные слова после префикса
  Если увязанные слова имеют слишком много слогов
    Случайно выбрать новое префиксное слово и повторить
  Случайно выбрать новое слово из увязанных слов
  Добавить новое слово в строку
  Подсчитать слоги в слове и вычислить их суммарное количество в строке
  Если слоги в строке равны целевому количеству слогов
    Вернуть строку и последнюю словарную пару в строке
В противном случае если это строка 2 или 3:
  Цель = 7 или 5 слогов
  Строка равна последней словарной паре из предыдущей строки
  Пока слоговая цель не достигнута:
    Префикс = последняя словарная пара в строке
    Получить увязанные слова после префикса, состоящего из словарной пары
    Если увязанные слова имеют слишком много слогов
      Случайно выбрать новый префикс, состоящий из словарной пары, и повторить
    Случайно выбрать новое слово из увязанных слов
    Добавить новое слово в строку
    Подсчитать слоги в слове и вычислить суммарное количество в строке
    Если сумма больше целого количества слогов
      Отбросить слово, сбросить сумму и повторить еще раз
    Если сумма меньше целого количества слогов
      Добавить слово в строку, сохранить сумму и повторить еще раз
```

Если сумма равна целому количеству слогов
Добавить слово в строку
Вернуть строку и последнюю словарную пару в строке
Показать результаты и меню с вариантами выбора

Тренировочный корпус

Марковские модели строятся из корпуса, поэтому они являются уникальными для этого корпуса. Модель, построенная из полного собрания сочинений Эдгара Райса Берроуза, будет другой и отличаться от модели, построенной из произведений Энн Райс. У всех нас есть фирменный стиль, или *голос*, и с учетом достаточно большой выборки марковский подход может порождать статистическую модель вашего стиля. Как и отпечатки пальцев, эта модель может увязать вас с документом или рукописью.

При построении марковских моделей используемый вами корпус будет представлять текстовый файл, состоящий почти из 300 древних и современных хокку, более 200 из которых были написаны признанными мастерами слова. В идеале ваш тренировочный корпус должен состоять из тысяч хокку, которые принадлежать перу одного и того же автора (ради единообразия голоса); но их трудно найти, в особенности потому, что многие из древних японских хокку не подчиняются слоговым правилам, либо намеренно, либо в результате перевода на другой, в частности английский, язык.

Для того чтобы увеличить число значений в расчете на ключ в марковской модели, хокку в первоначальном корпусе были продублированы 18 раз и случайно распределены по всему файлу. Этот факт не влияет на словесные ассоциации *внутри* хокку, но увеличивает взаимодействие *между* хокку.

Для иллюстрации допустим, что словарная пара в конце следующего ниже хокку является уникальной, увязывая только с начальным словом второго хокку; это приводит к довольно бесполезной паре "ключ — значение" `'hollow frog': ['mirror-pond']` (дутая лягушка, зеркальный пруд):

Open mouth reveals	Откроешь рот, и все пустое
Your whole wet interior	Влажное нутро наружу
Silly **hollow frog**!	**Дутая лягушка**!
Mirror-pond of stars	**Зеркальный пруд** из звезд
Suddenly a summer shower	Вдруг летний ливень
Dimples the water	Ямки на воде

Если хокку продублировать и перетасовать, то в это сочетание можно внести предлог, значительно увеличив шансы связать странную дутую лягушку с чем-то разумным:

Open mouth reveals	Откроешь рот, и все пустое
Your whole wet interior	Влажное нутро наружу
Silly **hollow frog**!	**Дутая лягушка**!

In the city fields	**На** городских лугах
Contemplating cherry trees	Вишню созерцая
Strangers are like friends	Чужаки похожи на друзей

Марковская модель теперь назначает "дутой лягушке" два значения: `'hollow frog': 'mirror-pond'` и `'in'`. И всякий раз, когда вы дублируете хокку, вы увидите увеличение числа значений в расчете на ключ для заканчивающих слов или словарных пар хокку. Но это полезно только до определенного момента; через некоторое время устанавливается убывающая возвратность от инвестиций, и вы начинаете снова и снова добавлять одни и те же значения, ничего не получая.

Отладка

Отладка — это процесс поиска и исправления ошибок (дефектов, багов) в аппаратном и программном обеспечении компьютера. Когда вы пытаетесь запрограммировать решение сложной задачи, вам нужно держать свою программу в ежовых рукавицах с целью отыскания источника проблемы, когда возникает что-то неожиданное. Например, если первая строка вашего хокку заканчивается с семью слогами вместо пяти, то вы захотите узнать, успешно ли завершилась функция подсчета слогов, возникли ли трудности в сопоставлении слов со словами, или программа подумала, что она находится на второй строке. Для того чтобы выяснить, что же пошло не так, вам нужно следить за тем, что конкретно ваша программа возвращает на каждом ключевом шаге, и это требует либо сооружения строительных лесов, либо ведения журнала. Оба технических решения будут рассмотрены в следующих двух разделах.

Сооружение строительных лесов

Сооружение строительных лесов, как определено здесь, — это временный код, который пишется для помощи в разработке программ, а затем удаляется, когда работа завершена. Название термина ссылается на леса, используемые в строительстве, — они необходимы, но никто не собирается их держать вечно.

Одним из распространенных элементов строительных лесов является инструкция `print()`, которая проверяет, что конкретно возвращает функция или вычисление. Пользователю не нужно видеть ее результат, поэтому вы удаляете ее после подтверждения правильности работы программы.

Полезные результаты строительных лесов включают такие вещи, как тип значения или переменной, длина набора данных и результаты инкрементных вычислений. Процитирую Аллена Дауни из книги "Думай по-питоновски" (Downey A. Think Python. O'Reilly, 2015): "Время, которое вы тратите на сооружение лесов, может уменьшить время, которое вы тратите на отладку".

Недостатком использования инструкций `print()` для отладки является то, что вам позже придется вернуться и удалить (или закомментировать) все эти инструкции, и вы рискуете случайно удалить инструкцию `print()`, которая полезна конечному

пользователю. К счастью, строительным лесам есть альтернатива, которая позволяет избежать этих проблем. Она называется модулем журналирования `logging`.

Использование модуля журналирования

Модуль журналирования `logging` является частью стандартной библиотеки Python (**https://docs.python.org/3/library/logging.html**). С помощью модуля `logging` можно получать кастомизированный отчет о том, что конкретно ваша программа делает в любом выбранном месте. Отчеты можно даже записать в долговременный журнальный файл. Следующий далее модуль журналирования используется в примере с интерактивной оболочкой для проверки правильности работы программы подсчета гласных:

```
❶ >>> import logging
❷ >>> logging.basicConfig(level=logging.DEBUG,
                          format='%(levelname)s - %(message)s')
>>> word = 'scarecrow'
>>> VOWELS = 'aeiouy'
>>> num_vowels = 0
>>> for letter in word:
        if letter in VOWELS:
            num_vowels += 1
    ❸ logging.debug('буква и количество = %s-%s', letter, num_vowels)

DEBUG - буква и количество = s-0
DEBUG - буква и количество = c-0
DEBUG - буква и количество = a-1
DEBUG - буква и количество = r-1
DEBUG - буква и количество = e-2
DEBUG - буква и количество = c-2
DEBUG - буква и количество = r-2
DEBUG - буква и количество = o-3
DEBUG - буква и количество = w-3
```

Для использования модуля `logging` сначала его импортируйте ❶. Затем задайте, какую отладочную информацию вы хотите видеть и в каком формате ❷. Уровень отладки `DEBUG` является самым низким уровнем информации и используется для диагностики деталей.

Обратите внимание, что в распечатке используется символьное форматирование со спецификатором `%s`. Существует возможность включить дополнительную информацию, например дата и время выводятся на экран с помощью `format='%(asctime)s'`, но в данном фрагменте кода вам лишь нужно проверить, правильно ли программа подсчитывает гласные.

Для каждой оцениваемой буквы введите собственное текстовое сообщение, которое будет печататься вместе со значениями переменных. Обратите внимание, что необходимо конвертировать несимвольные объекты, такие как целые числа и списки,

в символьные цепочки ❸. Далее следует результат работы модуля `logging`. Вы увидите совокупное количество наряду с тем, какие буквы фактически изменяют количество.

Как и строительные леса, модуль `logging` предназначен не для пользователя, а для разработчика. И подобно функции `print()`, журналирование может замедлить работу вашей программы. Для отключения сообщений журналирования просто вставьте вызов `logging.disable(logging.CRITICAL)` после импорта модуля, как показано ниже:

```
>>> import logging
>>> logging.disable(logging.CRITICAL)
```

Инструкция с отключающим вызовом, размещенная вверху программы, позволяет легко ее найти и активировать/деактивировать сообщения. Функция `logging.disable()` будет подавлять все сообщения на указанном уровне или ниже. Поскольку уровень `CRITICAL` является самым высоким, его передача в функцию `logging.disable()` отключает все сообщения. Такое техническое решение гораздо лучше, чем вручную отыскивать и отключать инструкции `print()` комментариями!

Код

Программный код markov_haiku.py этого раздела будет принимать тренировочный корпус train.txt, готовить марковские модели в виде словарей и генерировать хокку по одному слову за раз. Программа count_syllables.py и файл missing_words.json из *главы 8* обеспечат, чтобы скрипт markov_haiku.py использовал правильное число слогов в каждой строке. Все эти файлы можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/** (папка *главы 9*). Обязательно сохраните их вместе в одном каталоге.

Настройка

Листинг 9.1 импортирует необходимые модули, затем загружает и подготавливает внешние файлы.

Листинг 9.1. Импортирует, загружает и готовит тренировочный корпус. Файл markov_haiku.py, часть 1

```
❶ import sys
  import logging
  import random
  from collections import defaultdict
  from count_syllables import count_syllables

❷ logging.disable(logging.CRITICAL) # закомментировать для активации
                                    # отладочных сообщений
  logging.basicConfig(level=logging.DEBUG, format='%(message)s')
```

```
❸ def load_training_file(file):
      """Вернуть текстовый файл в виде символьной цепочки."""
      with open(file) as f:
       ❹ raw_haiku = f.read()
         return raw_haiku

❺ def prep_training(raw_haiku):
      """Загрузить символьную цепочку, удалить символы новой строки,
         разбить на слова по пробелам и вернуть список."""
      corpus = raw_haiku.replace('\n', ' ').split()
      return corpus
```

Начните с импортов, перечисленных в отдельных строках ❶. Модуль `logging` вам понадобится для получения отладочных сообщений, а контейнерный тип `defaultdict` поможет построить словарь из списка, автоматически создавая новый ключ без выдачи ошибки. Вы также импортируете функцию `count_syllables` из программы count_syllables.py, которую вы написали в *главе 8*. С остальными импортами вы должны быть уже знакомы.

Поместите инструкцию, отключающую журналирование, сразу после импорта, благодаря чему вы сможете ее легко найти. Для того чтобы увидеть журнальные сообщения, необходимо закомментировать эту инструкцию ❷. Следующая инструкция конфигурирует то, что вы увидите, как это было описано в предыдущем разделе. Я решил убрать с экрана обозначение уровня.

Затем определите функцию загрузки текстового файла тренировочного корпуса ❸. Используйте встроенную функцию `read()`, которая будет читать данные в виде символьной цепочки, подготовленную программой перед конвертацией в список ❹. Верните эту цепочку для использования в следующей функции.

Функция `prep_training()` ❺ в качестве аргумента принимает выход функции `load_training_file()`. Затем она заменяет символы новой строки пробелами, разбивает текст на слова по пробелам и конвертирует слова в элементы списка. В конце указанная функция возвращает корпус в виде списка.

Построение марковских моделей

Марковские модели — это просто словари Python, которые в качестве ключа используют слово или пару слов и в качестве значения слово, которое следует сразу за ними. Статистическая частота замыкающих слов улавливается повторением замыкающего слова в списке значений — подобно множествам, словари не могут иметь повторяющихся ключей, но они могут иметь повторяющиеся значения.

В листинге 9.2 определены две функции. Обе функции в качестве аргумента принимают корпус и возвращают марковскую модель.

Листинг 9.2. Определяет функции, которые строят марковские модели порядков 1 и 2. Файл markov_haiku.py, часть 2

```
❶ def map_word_to_word(corpus):
      """Загрузить список и применить словарь для увязки слова
         с последующим словом."""
   ❷ limit = len(corpus) - 1
   ❸ dict1_to_1 = defaultdict(list)
   ❹ for index, word in enumerate(corpus):
         if index < limit:
            ❺ suffix = corpus[index + 1]
              dict1_to_1[word].append(suffix)
   ❻ logging.debug("результаты увязки слова со словом для \"sake\" = %s\n",
dict1_to_1['sake'])
   ❼ return dict1_to_1

❽ def map_2_words_to_word(corpus):
      """ Загрузить список и использовать словарь для увязки
          словарной пары с замыкающим словом."""
   ❾ limit = len(corpus) - 2
      dict2_to_1 = defaultdict(list)
      for index, word in enumerate(corpus):
          if index < limit:
            ❿ key = word + ' ' + corpus[index + 1]
              suffix = corpus[index + 2]
              dict2_to_1[key].append(suffix)
      logging.debug("результаты увязки 2 слов со словосочетанием \"sake jug\"
= %s\n", dict2_to_1['sake jug'])
      return dict2_to_1
```

Сначала определите функцию для увязки каждого индивидуального слова с его следующим словом ❶. Программа будет использовать эту функцию только для того, чтобы отобрать второе слово хокку из инициирующего слова. Его единственным параметром является корпусный список, который возвращается функцией `prep_training()`.

Задайте предел `limit` для того, чтобы вы не могли выбирать последнее слово в корпусе ❷, потому что иначе это приведет к ошибке индекса. Теперь инициализируйте словарь с помощью контейнерного типа `defaultdict` ❸. Вы хотите, чтобы значения словаря были списками, которые содержат все найденные суффиксы, поэтому в качестве аргумента примените `list`.

Начните перебирать каждое слово из корпуса в цикле с использованием встроенной функции `enumerate`, превращая индекс каждого слова в объект ❹. Используйте условие и переменную `limit`, которые предотвратят выбор последнего слова в качестве ключа. Задайте переменную с именем `suffix`, которая будет представлять последующее слово ❺. Ее значением будет местоположение индекса текущего

слова плюс 1, т. е. следующее слово в списке. Добавьте эту переменную в словарь как значение текущего слова.

Для проверки того, что все работает по плану, примените журналирование, показав результаты для одного ключа ❻. В корпусе имеются тысячи слов, поэтому вы не захотите печатать их все. Выберите слово, которое, как вы знаете, находится в корпусе, например *sake* (саке́). Обратите внимание, что вы используете старое форматирование символьной цепочки с применением спецификатора `%`, поскольку оно соответствует текущей конструкции модуля журналирования. Завершите программу, вернув словарь ❼.

Следующая функция, `map_2_words_to_word()`, в основном является копией предыдущей, за исключением того, что в ней в качестве ключа используются два идущих подряд слова и увязываются последующие одиночные слова ❽. Важные изменения в ней состоят в том, чтобы установить предел в два слова назад от конца корпуса ❾, сделать ключ состоящим из двух слов с пробелом между ними ❿ и добавить 2 в индекс для переменной `suffix`.

Выбор случайного слова

Программа не сможет использовать марковскую модель без ключа, поэтому либо пользователь, либо программа должны предоставить первое слово в симулируемом хокку. Листинг 9.3 определяет функцию, которая случайно подбирает первое слово, способствуя автоматической инициации.

Листинг 9.3. Случайно выбирает инициирующее слово для порождения хокку. Файл markov_haiku.py, часть 3

```
❶ def random_word(corpus):
      """Вернуть случайное слово и количество слогов
         из тренировочного корпуса."""
   ❷ word = random.choice(corpus)
   ❸ num_syls = count_syllables(word)
   ❹ if num_syls > 4:
         random_word(corpus)
      else:
       ❺ logging.debug("случайное слово и слоги = %s %s\n", word,
                       num_syls)
         return (word, num_syls)
```

Определите функцию и передайте ей список `corpus` ❷. Затем задайте переменную `word` и используйте метод `choice()` модуля `random` для выбора слова из `corpus` ❷.

Примените функцию `count_syllables()` из модуля `count_syllables`, которая будет подсчитывать слоги в слове; сохраните их количество в переменной `num_syls` ❷. Я не поклонник однословных строк в хокку, поэтому не даю функции выбирать слово из более четырех слогов (напомним, что самые короткие строки хокку имеют

пять слогов). Если это происходит, то вызывайте функцию `random_word()` рекурсивно до тех пор, пока не получите приемлемое слово ❹. Обратите внимание, что по умолчанию Python имеет максимальную глубину рекурсии, равную 1000 вызовам, но коль скоро вы используете правильный тренировочный корпус хокку, существует мало шансов того, что вы ее превысите до того, как найдете подходящее слово. Если бы это было не так, то вы могли бы обратиться к решению этого условия позже, вызывая функцию с помощью цикла `while`.

Если слово содержит менее пяти слогов, то примените модуль `logging` для вывода на экран слова и его количества слогов ❺; затем верните слово и количество слогов в виде кортежа.

Применение марковских моделей

Для того чтобы выбрать одно-единственное слове, которое следует за инициирующим словом, используйте марковскую модель порядка 1. После этого программа должна отбирать все последующие слова с помощью модели порядка 2, которая в качестве ключей использует словарные пары. Листинг 9.4 определяет отдельную функцию для каждого из этих действий.

Листинг 9.4. Две функции для отбора слова с учетом префикса, марковской модели и количества слогов. Файл markov_haiku.py, часть 4

```
❶ def word_after_single(prefix, suffix_map_1, current_syls, target_syls):
      """ Вернуть все приемлемые слова в корпусе,
          которые следуют за одним словом."""
   ❷ accepted_words = []
   ❸ suffixes = suffix_map_1.get(prefix)
   ❹ if suffixes != None:
       ❸ for candidate in suffixes:
             num_syls = count_syllables(candidate)
             if current_syls + num_syls <= target_syls:
               ❻ accepted_words.append(candidate)
   ❺ logging.debug("приемлемые слова после \"%s\" = %s\n",
                   prefix, set(accepted_words))
     return accepted_words

❶ def word_after_double(prefix, suffix_map_2, current_syls, target_syls):
     """Вернуть все приемлемые слова в корпусе, которые следуют
        за словарной парой."""
     accepted_words = []
   ❷ suffixes = suffix_map_2.get(prefix)
     if suffixes != None:
         for candidate in suffixes:
             num_syls = count_syllables(candidate)
             if current_syls + num_syls <= target_syls:
                 accepted_words.append(candidate)
```

```
        logging.debug("приемлемые слова после \"%s\" = %s\n",
                      prefix, set(accepted_words))
    ⓾ return accepted_words
```

Определите функцию с именем `word_after_single()`, которая будет отбирать следующее слово в хокку, основываясь на предыдущем единственном инициирующем слове. Указанная функция в качестве аргументов принимает предыдущее слово, марковскую модель порядка 1, текущее количество слогов и целевое количество слогов ❶.

Создайте пустой список, в котором будут храниться приемлемые слова, т. е. слова, которые следуют за префиксом и в которых количество слогов не превышает целевое количество слогов ❷. Назовите эти замыкающие слова суффиксами `suffixes` и примените метод словаря `get()`, который возвращает заданное ключом значение словаря, для того чтобы назначить их переменной ❸. Если вы запросите ключ, которого в словаре нет, то вместо вызова ошибки `KeyError` метод `get()` вернет `None`.

Существует крайне редкий шанс, что префикс будет последним словом в корпусе и что он будет уникальным. В этом случае суффиксов не будет. Используйте инструкцию `if` для того, чтобы предвосхитить этот случай ❹. Если суффиксов нет, то функция, вызывающая функцию `word_after_single()`, которую вы определите в следующем разделе, выбирает новый префикс.

Каждый суффикс представляет собой *кандидатное* слово для хокку, но программа еще не определила, "впишется" ли этот кандидат или нет. Поэтому используйте цикл `for`, модуль `count_syllables` и инструкцию `if` для выявления, нарушает ли добавление слова в строку целевое количество слогов в строке ❺. Если цель не превышена, то добавьте слово в список принятых слов ❺. Покажите допустимые слова в сообщении модуля `logging`, а затем верните его ❺.

Следующая функция, `word_after_double()`, похожа на предыдущую функцию, за исключением того, что вы передаете ей словарные пары и марковскую модель порядка 2 (`suffix_map_2`) ❽ и получаете суффиксы из этого словаря ❾. Но как и функция `word_after_single()`, функция `word_after_double()` возвращает список допустимых слов ❿.

Генерирование строк хокку

Когда все вспомогательные функции будут готовы, можно определить функцию, которая фактически пишет строки хокку. Указанная функция будет либо строить все стихотворение хокку целиком, либо обновлять только вторую или третью строку. Можно выбрать один из двух путей: один — применить функцию, когда у программы для работы имеется лишь однословный суффикс, и другой — в любой другой ситуации.

Сборка первой строки

Листинг 9.5 определяет функцию, которая пишет строки хокку и инициирует первую строку хокку.

Листинг 9.5. Определяет функцию, которая пишет строки хокку и инициирует первую строку. Файл markov_haiku.py, часть 5

```
❸ def haiku_line(suffix_map_1, suffix_map_2, corpus,
                 end_prev_line, target_syls):
    """Собрать строку хокку из тренировочного корпуса и вернуть ее."""
  ❷ line = '2/3'
    line_syls = 0
    current_line = []
  ❸ if len(end_prev_line) == 0: # собрать первую строку
      ❷ line = '1'
      ❺ word, num_syls = random_word(corpus)
        current_line.append(word)
        line_syls += num_syls
      ❻ word_choices = word_after_single(word, suffix_map_1,
                                         line_syls, target_syls)
      ❼ while len(word_choices) == 0:
            prefix = random.choice(corpus)
            logging.debug("новый случайный префикс = %s", prefix)
            word_choices = word_after_single(prefix, suffix_map_1,
                                             line_syls, target_syls)
      ❽ word = random.choice(word_choices)
        num_syls = count_syllables(word)
        logging.debug("слово и слоги = %s %s", word, num_syls)
      ❾ line_syls += num_syls
        current_line.append(word)
      ❿ if line_syls == target_syls:
            end_prev_line.extend(current_line[-2:])
            return current_line, end_prev_line
```

Определите функцию, которая в качестве аргументов принимает обе марковские модели, тренировочный корпус, последнюю словарную пару из конца предыдущей строки и целевое количество слогов для текущей строки ❶. Сразу же задействуйте переменную для указания того, какие строки хокку симулируются ❷. Подавляющая часть обработки будет связана со второй и третьей строками (и, возможно, последней частью первой строки), где вы будете работать с существующим префиксом, состоящим из словарной пары, поэтому пусть они представляют базовый случай. После этого создайте счетчик для скользящего суммарного количества слогов в строке и создайте пустой список для хранения слов текущей строки.

Используйте инструкцию `if`, которая равна `True` при условии, что длина параметра `end_prev_line` — количество слогов в двух последних словах предыдущей строки — равна 0, имея в виду, что предыдущей строки не было и вы находитесь в первой строке ❸. Первая инструкция в том блоке `if` меняет переменную `line` на 1 ❹.

Выберите начальное инициирующее слово и получите его количество слогов, вызвав функцию `random_word()` ⑱. Задавая переменные `word` и `num_syls` вместе, вы "распаковываете" кортеж `(word, num_sylls)`, который возвращается функцией `random_word()`. Функции заканчиваются на инструкциях `return`, поэтому возврат кортежей является отличным способом вернуть несколько переменных. В более продвинутой версии этой программы можно применить функции-генераторы с ключевым словом `yield`, поскольку `yield` возвращает значение, не передавая контроль исполнения назад.

Далее добавьте слово `word` в `current_line` и добавьте `num_syls` в скользящий итог. Теперь, когда у вас есть начальное значение, соберите все его возможные суффиксы с помощью функции `word_after_single()` ⑰.

Если приемлемых слов нет, то начните цикл `while` для обработки этой ситуации. Этот цикл будет продолжаться до тех пор, пока не будет возвращен непустой список допустимых вариантов слов ⑰. Программа выберет новый префикс — призрачный префикс, используя метод `choice` модуля `random`. (Напомним, что этот префикс не станет частью хокку, а используется только для повторного обращения к марковской модели.) Внутри цикла `while` журнальное сообщение даст вам знать о том, какой призрачный префикс был выбран. Затем программа еще раз вызовет функцию `word_after_single()`.

После того как список допустимых слов будет построен, примените метод `choice` снова для того, чтобы выбрать слово из списка `word_choices` ⑱. Поскольку указанный список может включать дублирующие слова, именно здесь вы наблюдаете статистическое воздействие марковской модели. После этого подсчитайте слоги в слове и зарегистрируйте результаты модулем `logging`.

Добавьте количество слогов в скользящий итог строки и добавьте слово в список `current_line` ⑱.

Если количество слогов в первых двух словах равно 5 ⑯, то объявите переменную `end_prev_line` и назначьте ей последние два слова предыдущей строки; эта переменная является префиксом второй строки. Наконец, верните всю строку и переменную `end_prev_line`.

Если целевое количество слогов в первой строке не было достигнуто, то программа перепрыгнет к циклу `while` в следующем разделе с целью завершить строку.

Сборка остальных строк

В листинге 9.6 последняя часть функции `haiku_line()` обращается к случаю, когда хокку уже содержит префикс, состоящий из словарной пары, который программа может использовать в марковской модели порядка 2. Программа применяет ее для завершения первой строки — исходя из того, что первые два слова еще не составляют пять слогов — и для сборки второй и третьей строк. После того как полное хокку будет написано, пользователь также сможет регенерировать вторую и третью строки.

Листинг 9.6. Использует марковскую модель порядка 2 для завершения функции, пишущей строки хокку. Файл markov_haiku.py, часть 6

```
❹ else: # собрать строки 2 и 3
    ❷ current_line.extend(end_prev_line)

❸ while True:
      logging.debug("строка = %s\n", line)
    ❹ prefix = current_line[-2] + ' ' + current_line[-1]
    ❺ word_choices = word_after_double(prefix, suffix_map_2,
                                       line_syls, target_syls)
    ❻ while len(word_choices) == 0:
          index = random.randint(0, len(corpus) - 2)
          prefix = corpus[index] + ' ' + corpus[index + 1]
          logging.debug("новый случайный префикс = %s", prefix)
          word_choices = word_after_double(prefix, suffix_map_2,
                                           line_syls, target_syls)
      word = random.choice(word_choices)
      num_syls = count_syllables(word)
      logging.debug("слово и слоги = %s %s", word, num_syls)

    ❼ if line_syls + num_syls > target_syls:
          continue
      elif line_syls + num_syls < target_syls:
          current_line.append(word)
          line_syls += num_syls
      elif line_syls + num_syls == target_syls:
          current_line.append(word)
          break
❽ end_prev_line = []
  end_prev_line.extend(current_line[-2:])

❾ if line == '1':
      final_line = current_line[:]
  else:
      final_line = current_line[2:]

  return final_line, end_prev_line
```

Начните с инструкции `else`, которая исполняется, если есть суффикс ❶. Поскольку последняя часть функции `haiku_line()` должна обрабатывать первую строку, а также вторую и третью строки, примените трюк, в котором вы добавляете список `end_prev_line` (построенный вне условия на шаге ❽) в список `current_line` ❷. Позже, когда в хокку будет добавлена завершенная строка, эта начальная словарная пара будет отброшена.

Начните цикл `while`, который продолжается до тех пор, пока в строке не будет достигнуто целевое количество слогов ❸. Каждая итерация начинается с отладочного сообщения, которое информирует о вычисляемом в цикле пути: `'1'` или `'2/3'`.

При добавлении последних двух слов предыдущей строки в начало текущей строки последние два слова текущей строки всегда будут префиксом ❹.

Используя марковскую модель порядка 2, создайте список допустимых слов ❺. Если указанный список является пустым, то программа использует процедуру призрачного префикса ❺.

Оцените, что делать дальше, с помощью количества слогов ❺. Если слогов слишком много, то примените инструкцию `continue` для перезапуска цикла `while`. Если слогов недостаточно, то добавьте слово и количество его слогов к количеству слогов в строке. В противном случае добавьте слово и завершите цикл.

Назначьте последние два слова в строке переменной `end_prev_line` для того, чтобы программа могла использовать ее в качестве префикса для следующей строки ❾. Если текущий путь равен `'1'`, скопируйте текущую строку в переменную с именем `final_line`; если этот путь равен `'2/3'`, используйте индексный срез с целью исключить первые два слова перед передачей значения переменной `final_line` ❾. Таким образом, вы удалите первоначальную словарную пару `end_prev_line` из второй или третьей строк.

Написание пользовательского интерфейса

Листинг 9.7 определяет функцию `main()` программы markov_haiku.py, которая выполняет настроечные функции и организует пользовательский интерфейс. Этот интерфейс представляет пользователю меню с вариантами выбора и показывает результирующее хокку.

Листинг 9.7. Запускает программу и показывает пользовательский интерфейс. Файл markov_haiku.py, часть 7

```
def main():
    """Предоставить пользователю варианты: собрать новое хокку либо
       модифицировать существующее."""
    intro = """\n
    Тысяча мартышек за тысячью печатными машинками...
    или же один компьютер... иногда способны порождать хокку.\n"""
    print("{}".format(intro))

❶ raw_haiku = load_training_file("train.txt")
    corpus = prep_training(raw_haiku)
    suffix_map_1 = map_word_to_word(corpus)
    suffix_map_2 = map_2_words_to_word(corpus)
    final = []
```

```
    choice = None
❷ while choice != "0":

    ❸ print(
          """
          Генератор японского хокку
          0 - Выйти
          1 - Сгенерировать хокку
          2 - Регенерировать строку 2
          3 - Регенерировать строку 3
          """
          )

    ❹ choice = input("Вариант: ")
      print()

      # выйти из программы
    ❺ if choice == "0":
          print("Сайенара.")
          sys.exit()

      # сгенерировать полное хокку
    ❻ elif choice == "1":
          final = []
          end_prev_line = []
          first_line, end_prev_line1 = haiku_line(suffix_map_1,
                                                  suffix_map_2,
                                                  corpus, end_prev_line, 5)
          final.append(first_line)
          line, end_prev_line2 = haiku_line(suffix_map_1, suffix_map_2,
                                            corpus, end_prev_line1, 7)
          final.append(line)
          line, end_prev_line3 = haiku_line(suffix_map_1, suffix_map_2,
                                            corpus, end_prev_line2, 5)
          final.append(line)

      # регенерировать строку 2
    ❼ elif choice == "2":
          if not final:
              print("Сначала сгенерируйте все хокку (Вариант 1).")
              continue
          else:
              line, end_prev_line2 = haiku_line(suffix_map_1,
                                                suffix_map_2,
                                            corpus, end_prev_line1, 7)
              final[1] = line
```

```
        # регенерировать строку 3
    ❽ elif choice == "3":
            if not final:
                print("Сначала сгенерируйте все хокку (Вариант 1).")
                continue
            else:
                line, end_prev_line3 = haiku_line(suffix_map_1,
                                                  suffix_map_2,
                                                corpus, end_prev_line2, 5)
                final[2] = line

        # некий неизвестный вариант
    ❾ else:
            print("\nИзвините, но это недопустимый вариант.",
                  file=sys.stderr)
            continue

    ❿ # показать результаты
        print()
        print("Первая строка = ", end="")
        print(' '.join(final[0]), file=sys.stderr)
        print("Вторая строка = ", end="")
        print(" ".join(final[1]), file=sys.stderr)
        print("Третья строка = ", end="")
        print(" ".join(final[2]), file=sys.stderr)
        print()

    input("\n\nНажмите клавишу Enter для выхода из программы.")

if __name__ == '__main__':
    main()
```

После вводного сообщения загрузите и подготовьте тренировочный корпус, а потом постройте две марковские модели. Затем создайте пустой список для хранения окончательного хокку ❶. Далее объявите переменную `choice` и установите для нее значение `None`. Начните цикл `while`, который продолжается до тех пор, пока пользователь не выберет вариант `0` ❷. Введя `0`, пользователь решает выйти из программы.

Примените инструкцию `print()` с тройными кавычками для вывода на экран меню ❸, а затем получите от пользователя выбранный им вариант ❹. Если пользователь выбирает `0`, то попрощайтесь и выйдите из программы ❺. Если пользователь выбирает `1`, то он хочет, чтобы программа сгенерировала новое хокку, поэтому повторно инициализируйте список `final` и переменную `end_prev_line` ❻. Затем вызовите функцию `haiku_line()` для всех трех строк и передайте ей правильные аргументы, включая целевое количество слогов в каждой строке. Обратите внимание, что имя переменной `end_prev_line` изменяется с каждой строкой; например, пере-

менная `end_prev_line2` содержит последние два слова второй строки. Последняя переменная, `end_prev_line3`, является просто заполнителем, для того чтобы можно было использовать функцию повторно; другими словами, она никогда не используется на практике. Каждый раз, когда вызывается функция `haiku_line()`, она возвращает строку, которую необходимо добавить в список `final`.

Если пользователь выбирает 2, то программа регенерирует вторую строку ❼. Прежде чем программа будет регенерировать строку, должно существовать полное хокку, поэтому используйте инструкцию `if` для обработки ситуации, когда пользователь побежит впереди паровоза. Затем вызовите функцию `haiku_line()`, передав ей переменную `end_prev_line1`, связав ее с предыдущей строкой, и установите целевое количество слогов равным семи. Вставьте перестроенную строку в список `final` в индексе 1.

Повторите процесс, если пользователь выбирает 3, только сделайте целевое количество слогов равным 5 и передайте переменную `end_prev_line2` функции `haiku_line()` ❽. Вставьте строку в индексе 2 в список `final`.

Если пользователь вводит что-либо, что в меню отсутствует, то сообщите ему об этом и затем продолжите цикл ❾. Закончите обработку, показав хокку. Примените метод `join()` и `file=sys.stderr` для структурированной распечатки в интерактивной оболочке ❿.

Завершите программу стандартным фрагментом кода для запуска программы в виде модуля либо в автономном режиме.

Результаты

Для того чтобы оценить программу написания стихотворений, вам нужен способ измерить нечто субъективное — "хороши" или неудачны стихотворения, используя объективные критерии. Для программы markov_haiku.py я предлагаю следующие категории, основанные на двух критериях — оригинальности и человекоподобии.

- **Дубликат** — дословное дублирование хокку из тренировочного корпуса.
- **Хорошее** — такое хокку, которое (по крайней мере для некоторых людей) не отличимо от хокку, написанного поэтом-человеком. Оно должно представлять собой начальный результат либо результат неоднократной регенерации второй или третьей строки.
- **Инициированное** — такое хокку, которое имеет достоинства, но которое многие заподозрили бы в том, что оно было написано компьютером, либо такое хокку, которое вы могли бы превратить в хорошее хокку, изменив или переставив не более двух слов (подробнее описано ниже). Оно может потребовать многократной регенерации второй или третьей строки.
- **Мусор** — такое хокку, которое явно является случайной амальгамой слов и не имеет никакого достоинства как стихотворение.

Если вы примените программу для генерирования большого количества хокку и поместите результаты в эти категории, то в итоге вы, вероятно, закончите распре-

делением, показанным на рис. 9.3. Примерно в 5% случаев вы будете дублировать хокку, существующее в тренировочном корпусе; в 10% случаев вы создадите хорошее хокку; около 25% из них будут проходимыми либо исправимыми, а остальные будут мусором.

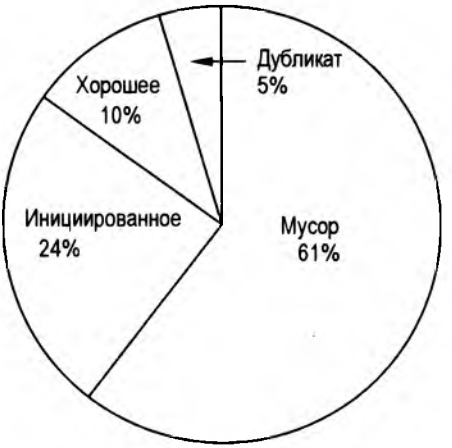


Рис. 9.3. Субъективные результаты генерирования 500 хокку с помощью программы markov_haiku.py

Учитывая то, насколько простым является марковский процесс, результаты на рис. 9.3 впечатляют. Еще раз процитирую Чарльза Хартмана: "Здесь язык самовоспроизводит себя из ничего, из простого статистического шума... Мы наблюдаем, как чувство эволюционирует и смысл, шатаясь, поднимается на свои собственные чудодейственные ноги".

Хорошее стихотворение хокку

Далее приведено несколько примеров симулированных хокку, классифицированных как "хорошие". В первом примере программа едва уловимо — можно было бы сказать "умело", если бы вы не знали, что за это в ответе алгоритм, — изменила мое хокку из *главы 8*, произведя новое хокку с тем же значением.

Cloudbanks that I let	Небесные гряды, о которых
Myself pretend are distant	Грезил я, горами
Mountains faraway	Были вдалеке

В следующем примере программе удалось продублировать общую тему в традиционном хокку: сопоставление образов или идей.

The mirror I stare	В зеркало смотрю
Into shows my father's face	Оно явит отца лицо
An old silent pond	Старый тихий пруд

В этом случае вы обнаружите, что зеркало на самом деле является поверхностью неподвижного пруда, хотя вы можете интерпретировать само лицо как пруд.

Выполнение этой программы немного смахивает на промывку золотого песка в лотке: иногда вы обнаруживаете самородок. Верхнее хокку было написано Рингаем более 300 лет назад. В нижнем хокку программа внесла тонкие изменения, в результате чего это стихотворение теперь вызывает образы поздних весенних заморозков — отката назад от неумолимого движения времени года.

In these dark waters Drawn up from my frozen well Glittering of Spring *—Ringai*	В этих темных водах Зачерпнутых из замерзшего колодца Весны сверкает мерцание *— Рингай*
Waters drawn up from My frozen well glittering Of Spring standing still *—Python*	Вода, зачерпнута из Замерзшего колодца — сверкает Весной, застывшей на миг *— Python*

Ниже приведены примеры еще нескольких "хороших" хокку. Первый примечателен тем, что он был построен из трех отдельных хокку из тренировочного корпуса, но повсюду сохраняет четкую контекстуальную нить.

As I walk the path Eleven brave knights canter Through the stormy woods	Пока иду я по тропинке Одиннадцать рыцарей храбрых Скачут галопом по бурному лесу
Cool stars enter the Window this hot evening all Heaven and earth ache	Прохладные звезды Входят в окно в эту жаркую мглу И небо с землей их жаждут до боли
Such a thing alive Rusted gate screeches open Even things feel pain	Такая вещь живая Ржавые ворота — с визгом открываются Даже вещи чувствуют боль
The stone bridge! Sitting Quietly doing nothing Yet Spring comes grass grows	Каменный мост! Сидит Ленивый тихо Но вот весна, и он зарастает травой
Dark sky oh! Autumn Snowflakes! A rotting pumpkin Collapsed and covered	О, темное небо! Осенние Снежинки! Гниющая тыква Распалась и ими укрылась
Desolate moors fray Black cloudbank, broken, scatters In the pines, the graves	Заброшенные пустоши потрепали Черную тучу, разбили и рассеяли В соснах и в могилах

Инициирующее хокку

Идея о компьютерах, помогающих людям писать стихи, существует уже немало лет. Поэты часто "заправляют насос" путем имитации более ранних стихов, и нет причин, почему компьютер не может предоставлять первые черновики в рамках киберпартнера. Даже довольно плохое компьютерное творение имеет потенциал "инициировать" творческий процесс и помочь человеку преодолеть творческий кризис.

Ниже приведены три инициирующих примера хокку из программы markov_haiku.py. Слева находится не совсем правильное компьютерное хокку. Справа — версия, которую я исправил. В каждом из них я изменил только одно слово, выделенное жирным шрифтом.

My life must end like
Another flower what a
Hungry wind **it** is

Моя жизнь должна закончиться, как
Еще один цветок. Какой
Голодный ветер

My life must end like
Another flower what a
Hungry wind is **death**

Моя жизнь должна закончиться как
Еще один цветок. Какой
Голодный **ветер-смерть**

The dock floating in
The hot caressing night just
Before the dawn **old**

Док плывет в
Жаркой ласковой ночи
Перед самым рассветом **старый**

The dock floating in
The hot caressing night just
Before the dawn **rain**

Док плывет в
Жаркой ласковой ночи
Перед самым рассветным **дождем**

Moonrise on the grave
And my old sadness a sharp
Shovel thrust **the** stars

Восход Луны на могиле
И моя старая печаль острая
Лопата вонзилась **в** звезды

Moonrise on the grave
And my old sadness a sharp
Shovel thrust **of** stars

Восход Луны на могиле
И моя старая печаль. Вонзились
Точно острая лопата **в небо** звезды

Последнее имеет загадочный смысл, но, кажется, работает, поскольку оно наполнено естественными ассоциациями (луна и звезды, могила и лопата, могила и печаль). Во всяком случае, вам не следует слишком беспокоиться о смысле. Перефразируя Т. С. Элиота: смысл подобен мясу, которое грабитель бросает собаке, для того чтобы отвлечь ум, пока стихотворение делает свою работу!

Резюме

Нам потребовались две главы, но теперь у вас есть программа, которая может симулировать японское хокку, созданное мастерами слова — или, по крайней мере, обеспечить полезную отправную точку для поэта-человека. Кроме того, вы приме-

нили модуль журналирования `logging` для мониторинга того, что конкретно программа делает в ключевых точках.

Дальнейшее чтение

Книга Чарльза О. Хартмана "Виртуальная муза: эксперименты с компьютерной поэзией" (Hartman C. O. Virtual muse: experiments in computer poetry. Wesleyan University Press, 1996) представляет собой увлекательный взгляд на раннее сотрудничество между людьми и компьютерами в стихосложении.

Если вы хотите узнать о Клоде Шенноне больше, то ознакомьтесь с книгой Джимми Сони и Рода Гудмана "Игра ума: как Клод Шеннон изобрел информационный век (Soni J., Goodman R. A Mind at play: how Claude Shannon invented the information age. Simon & Schuster, 2017).

Цифровую версию книги "Японское хокку: двести двадцать примеров семнадцатисложных стихотворений (Beilenson P. Japanese haiku: two hundred twenty examples of seventeen-syllable poems. The Peter Pauper Press, 1955), переведенную на английский язык Питером Бейленсоном, можно найти на веб-сайте Global Grey (**https://www.globalgreyebooks.com/**).

В статье "Гайку: генерирование хайку с помощью норм словарной ассоциации" (Gaiku: generating haiku with word association norms. Association for Computational Linguistics, 2009) Яэль Нетцер (Yael Netzer) и соавторы разведывают использование норм ассоциации слов (word association norm, WAN) для генерирования хокку. Можно создавать корпусы WAN, отправляя людям триггерные слова и записывая их немедленные ответы (например, house для fly, arrest, keeper и т. д.). Это приводит к неким тесно связанным, интуитивным отношениям, характерным для человеческого хокку. Эту работу можно найти в Интернете по адресу

http://www.cs.brandeis.edu/~marc/misc/proceedings/naacl-hlt-2009/CALC-09/pdf/CALC-0905.pdf.

Книга Эла Свейгарта "Автоматизируй скучные вещи с помощью Python" (Sweigart A. Automate the boring stuff with Python. No Cramp Press, 2015) имеет полезную обзорную главу о методах отладки, включая журналирование.

Сложные проекты

В этом разделе я описал несколько предложений для побочных проектов. Как и во всех сложных проектах, вы предоставлены сами себе — никаких решений не предусмотрено.

Генератор новых слов

В своем удостоенном наград научно-фантастическом романе 1961 года "Незнакомец в чужой стране" (Stranger in a Strange Land) автор Роберт А. Хайнлайн (Robert A. Heinlein) изобрел слово "грок" (grok) для обозначения глубокого, интуитивного понимания. Это слово вошло в популярную культуру — в особенности культуру

компьютерного программирования — и сейчас находится в Оксфордском словаре английского языка.

Придумать новое слово, которое звучит легитимно, совсем нелегко, отчасти потому, что люди очень привязаны к словам, которые мы уже знаем. Но компьютеры от этого недуга не страдают. В "Виртуальной музе" Чарльз Хартман заметил, что его программа генерирования стихотворений иногда создает интригующие комбинации букв, такие как *runkin* или *avatheformitor*, которые могут легко представлять новые слова.

Напишите программу, которая рекомбинирует буквы, используя марковские модели порядков 2, 3 и 4, и примените эту программу для генерирования новых интересных слов. Дайте им определение и начните их применять. Кто знает — может быть, вы еще раз услышите *frickin*, *frabjous*, *chortle* или *trill*!

Тест Тьюринга

По словам Алана Тьюринга, "компьютер заслужил бы того, чтобы его называли разумным, если бы он смог обхитрить человека, заставив его поверить, что он человек". Пригласите своих друзей проверить хокку, сгенерированные программой markov_haiku.py. Перемешайте компьютерное хокку с несколькими хокку, написанными поэтами или вами самим. Так как компьютерные хокку часто характерны тем, что имеют анжамбеман, т. е. перетекание со строки на строку, внимательно выберите человеческие хокку, которые тоже имеют анжамбеман, чтобы не дать своим умным друзьям прокатиться бесплатно. Также помогает использование строчных букв и минимальной пунктуации во всех хокку. На рис. 9.4 я привел пример с использованием Facebook.

Рис. 9.4. Пример эксперимента с тестом Тьюринга в Facebook

Потрясающе! Просто потрясающе!

Президент Трамп известен тем, что говорит короткими, простыми предложениями, в которых присутствуют "лучшие слова", а короткие, простые предложения отлично подходят для хокку. Газета Washington Post на самом деле опубликовала несколько непреднамеренных хокку, обнаруженных в некоторых его предвыборных речах. Среди них были:

He's a great, great guy.	Он великолепный, великолепный парень.
I saw him the other day.	Видел его на днях.
On television.	По телевизору.
They want to go out.	Они хотят путешествовать.
They want to lead a good life.	Они хотят вести хорошую жизнь.
They want to work hard.	Они хотят много работать.
We have to do it.	Мы должны это сделать.
And we need the right people.	И нам нужны правильные люди.
So Ford will come back.	И значит, Форд вернется.

Используйте онлайновые стенограммы выступлений Дональда Трампа, для того чтобы построить новый тренировочный корпус для программы markov_haiku.py. Помните, что вам нужно будет вернуться к *главе 8* и построить новый словарь "отсутствующих слов" с любыми словами, не входящими в словарь произношения Университета Карнеги — Меллона. Затем снова выполните программу и сгенерируйте хокку, которые улавливают этот момент в истории. Сохраните лучшие из них и вернитесь к тесту Тьюринга для того, чтобы узнать, смогут ли ваши друзья отделить ваше хокку от истинных цитат Трампа.

Писать хокку или не писать

Уильям Шекспир написал много известных фраз, которые укладываются в слоговую структуру хокку, такие как "all our yesterdays" (все наши вчерашние дни), "dagger of the mind" (кинжал ума) и "parting is such sweet sorrow" (расставанья столь сладкая печаль). Используйте одну или несколько пьес знаменитого драматурга в качестве тренировочного корпуса для программы markov_haiku.py. Большой трудностью здесь будет подсчет слогов для всего этого старого *аглицкаго* языка.

Марковская музыка

Если вы отмечены музыкальными талантами, то выполните онлайновый поиск по терминам "composing music with Markov chains" (сочинение музыки с помощью марковских цепей). Вы должны найти обширный материал по применению анализа марковских цепей для сочинения музыки, используя ноты существующих песен в качестве тренировочного корпуса. Получившаяся "марковская музыка" используется подобно нашим инициирующим хокку — как вдохновение для песенников-людей.

10

МЫ ОДНИ? РАЗВЕДЫВАНИЕ ПАРАДОКСА ФЕРМИ

Ученые используют уравнение Дрейка для получения оценочного количества возможных цивилизаций в галактике, в настоящее время испускающих электромагнитные излучения, такие как радиоволны. В 2017 г. указанное уравнение было обновлено с учетом открытий новых экзопланет с помощью космической обсерватории NASA "Кеплер". Результат, опубликованный в научном журнале Astrobiology, был ошеломляющим.

Для того чтобы человечество стало первым и единственным технологически развитым видом, вероятность того, что на обитаемой чужой планете разовьется продвинутая цивилизация, должна составить меньше 1 к 10 миллиардам триллионов! И все же, как заметил лауреат Нобелевской премии физик Энрико Ферми, "Где все остальные?".

Ферми относился к межзвездным путешествиям более скептически, чем к существованию инопланетян, но его вопрос стал известен как *парадокс Ферми*, и он превратился в гипотезу "если бы они были там, то они были бы и тут". По данным Института SETI, даже при скромных ракетных технологиях усердная цивилизация в состоянии разведать всю галактику, если вообще ее не колонизировать, в течение 10 млн лет. Может показаться, что этот срок является очень долгим, но это только 1/1000 возраста Млечного Пути! В результате некоторые пришли к пониманию парадокса Ферми как доказательству того, что мы в космосе одни. Другие находят уязвимости в аргументации.

В этой главе вы расследуете отсутствие инопланетных радиопередач, вычислив вероятность того, что одна цивилизация обнаружит другую, основываясь на объеме их передач и результате уравнения Дрейка. Вы также примените стандартный пакет

`tkinter` графического интерфейса Python с целью быстрого и легкого создания графической модели Млечного Пути.

Проект 17: моделирование Млечного Пути

Наша галактика, Млечный Путь, является довольно распространенной спиральной галактикой, как та, которая показана на рис. 10.1.

Рис. 10.1. Спиральная галактика NGC 6744, "Старший брат" Млечного Пути

В поперечном разрезе Млечный Путь представляет собой сплющенный диск с центральной выпуклостью, которая в своем ядре, скорее всего, содержит сверхмассивную черную дыру. Четыре "спиральных рукава" — в состав которых входят относительно плотно упакованные газ, пыль и звезды — излучаются из этой центральной массы. Размеры Млечного Пути показаны на рис. 10.2.

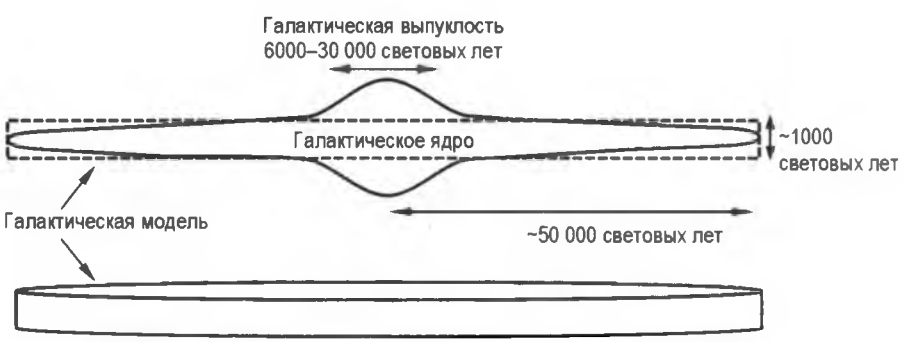


Рис. 10.2. Схематический профиль (вид по краю) галактики Млечный Путь и упрощенная модель

Центр галактики считается довольно негостеприимным для жизни местом из-за высокого уровня излучения, связанного с более плотно упакованными звездами. Поэтому в данном проекте вы можете рассматривать галактику как простой диск,

сбросив часть объема, связанного с выпуклостью, но все же оставив место для нескольких развитых цивилизаций вблизи ядра (см. галактическую модель на рис. 10.2).

Цель

Для заданного числа развитых галактических цивилизаций и среднего размера радиопузыря оценить вероятность обнаружения одной цивилизацией радиопередач любой другой цивилизации. Для перспективы разместить на двухмерном графическом представлении Млечного Пути размер текущего радиопузыря Земли.

Стратегия

Вот шаги, необходимые для выполнения этого проекта:

1. Оценить число передающих цивилизаций, используя уравнение Дрейка.
2. Выбрать диапазон размеров их радиопузырей.
3. Создать формулу для оценивания вероятности обнаружения одной цивилизацией другой.
4. Построить графическую модель галактики и разместить земной пузырь радиоизлучений.

Для того чтобы программный код оставался близким к описанию, каждая из этих задач будет подробно описана в собственном разделе. Обратите внимание, что первые два шага применения языка Python не требуют.

Оценивание числа цивилизаций

Число развитых цивилизаций можно оценить вручную, используя уравнение Дрейка:

$$N = R^* \cdot f_p \cdot n_e \cdot f_l \cdot f_i \cdot f_c \cdot L,$$

где:

- N — число цивилизаций в нашей галактике, электромагнитные излучения которых можно обнаружить;
- R^* — средняя скорость звездообразования в галактике (новые звезды в год);
- f_p — доля звезд с планетами;
- n_e — для звезд с планетами среднее число планет с окружающей средой, пригодной для жизни;
- f_l — доля планет, которые развивают жизнь;
- f_i — доля планет-носителей жизни с разумной, цивилизованной жизнью;
- f_c — доля цивилизаций, которые испускают в космос обнаруживаемые признаки своего существования;

- L — период времени (в годах), в течение которого цивилизации испускают обнаруживаемые сигналы.

Благодаря недавним достижениям в области обнаружения экзопланет первые три компоненты (R^*, f_p, n_e) становятся все более ограниченными. Например, недавние исследования показывают, что пригодными для той или иной формы жизни могут быть от 10 до 40% всех планет.

Что касается остальных компонент, то Земля является единственным примером. В 4,5-миллиардной истории Земли *Homo sapiens* существовал всего 200 тыс. лет, цивилизация — всего 6 тыс. лет, а радиопередачи — всего 112 лет. Что касается L, то способность цивилизации передавать радиосигналы могут нарушить войны, эпидемии, ледниковые периоды, столкновения с астероидами, сверхвулканы, сверхновые звезды и выброс корональной массы. И чем короче время передачи, тем меньше вероятность сосуществования цивилизаций.

Согласно статье в Википедии об уравнении Дрейка (**https://erLwikipedia.org/wiki/Drake_equation**), в 1961 г. Дрейк и его коллеги оценили число общающихся в галактике цивилизаций в пределах от 1000 до 100 млн. Более поздние обновления установили диапазон от 1 (только мы) до 15,6 млн (табл. 10.1).

Таблица 10.1. *Некоторые входные данные и результаты уравнения Дрейка*

Параметр	Дрейк, 1961** г.	Дрейк, 2017 г.	Ваши варианты; срединная точка показанных диапазонов
R^*	1	3	
f_p	0,35	1	
n_e	3	0.2	
f_1	1	0,13	
f_i	1	1	
f_c	0,15	0,2	
L	50×10^6	1×10^9	
N	$7,9 \times 10^6$	$15,6 \times 10^6$	

** Средняя точка представленных диапазонов.

Для ввода в программу можно использовать оценочные значения из указанной таблицы, либо те, которые вы найдете в Интернете, либо те, которые вы вычислите сами (в последнем столбце таблицы).

Выбор размеров радиопузыря

Радиоволны, которые не фокусируются в луч для нацеленной передачи, являются случайными. Думайте об этом как о "планетной утечке". Поскольку мы решили не сообщать о нашем присутствии инопланетянам, которые могут прийти и нас

съесть, почти все наши радиопередачи являются случайными. Эти передачи образуют расширяющуюся сферу вокруг Земли, в настоящее время диаметром около 225 световых лет.

Пузырь в 225 световых лет звучит впечатляюще, но на самом деле имеет совершенно конкретный размер. Фронт радиоволн подчиняется закону обратных квадратов, т. е. при расширении он непрерывно теряет плотность мощности. Дополнительные потери мощности могут быть вызваны поглощением или рассеянием. В какой-то момент сигнал ослабляется настолько, что становится неотделимым от фонового шума. Даже с помощью нашей лучшей технологии — радиотелескопов программы прорывного прослушивания Breakthrough Listen[1] — мы смогли обнаружить, что наш собственный радиопузырь простирается только примерно на 16 световых лет.

Поскольку мы по-настоящему расследуем причину, почему мы не обнаружили инопланетян, то в целях этого проекта вы должны принять допущение о том, что у других цивилизаций есть технология, подобная нашей. Еще одно допущение должно заключаться в том, что, как и мы, все инопланетяне имеют параноидальное планетарное сознание и не передают сигналы типа "а вот и мы", которые объявили бы об их присутствии. Разведывание размеров случайных пузырей, варьирующихся от немного меньших, чем те, которые в настоящее время обнаруживаются, до тех, которые не намного больше, чем наши собственные радиопередачи, должно быть разумным местом для начала. Это предполагает диаметрический диапазон от 30 до 250 световых лет. Хотя мы не можем обнаружить пузырь в 250 световых лет, будет интересно посмотреть, каковы были бы шансы, если бы мы смогли.

Генерирование формулы вероятности обнаружения

По мере увеличения числа развитых цивилизаций в галактике возрастает и вероятность того, что одна из них обнаружит другую. Это интуитивно понятно, но как назначить фактические вероятности?

Самое приятное в компьютерах то, что они позволяют нам пробиваться к решениям, которые бывают интуитивными, а бывают и осознанными. Одним из подходов здесь было бы сделать трехмерную модель диска Млечного Пути, случайно распределить цивилизации повсюду и измерить расстояния между ними, используя один из многих инструментов Python для вычисления евклидова расстояния. Но с учетом потенциально сотен миллионов анализируемых цивилизаций этот метод был бы вычислительно дорогостоящим.

---

[1] Breakthrough Listen — проект Юрия Мильнера по поиску разумной внеземной жизни во Вселенной, рассчитанный на 10 лет и с проектным бюджетом 100 млн долларов. Является составной частью проекта Мильнера под названием "Прорывные инициативы" (Breakthrough Initiatives) и был анонсирован одновременно с проектом Breakthrough Message.
См. **https://ru.wikipedia.org/wiki/Breakthrough_Listen**. — *Прим. перев.*

Поскольку мы имеем дело с огромными неизвестными, нет необходимости в сверхточности или прецизионности. Мы просто хотим держаться в поле зрения, т. е. быть более или менее точными, поэтому простое упрощение состоит в том, чтобы структурировать галактику на ряд "эквивалентных" по радиопузырям объемов, разделив объем галактического диска на объем радиопузыря (рис. 10.3).

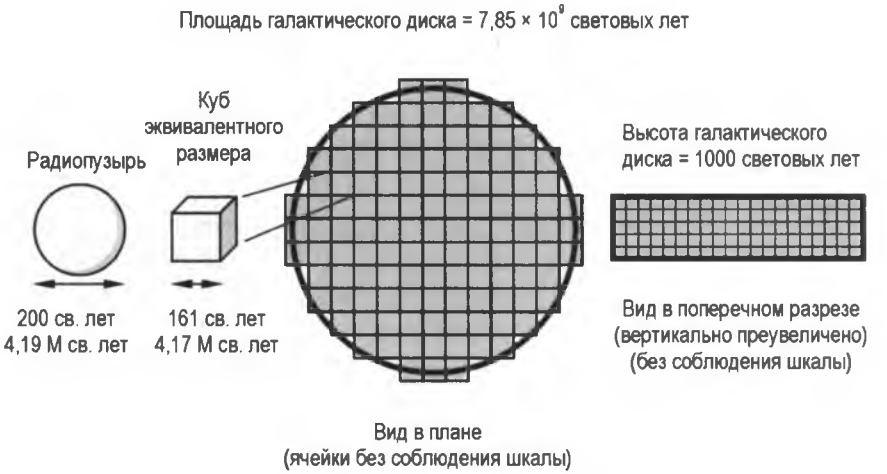


Рис. 10.3. Моделирование галактики с использованием кубов с объемами, эквивалентными 200-светогодичному радиопузырю

Объемы можно найти, используя следующие уравнения:

$$\text{объем диска} = n \times R^2 \times \text{высота диска};$$

$$\text{объем радипузыря} = \frac{4}{3} \times \pi \times r^3;$$

$$\text{шкалированный объем диска} = \frac{\text{объем диска}}{\text{объем радиопузыря}},$$

где R — радиус галактического диска; r — радиус радиопузыря.

Шкалированный объем диска — это число эквивалентных объемов, которые "вписываются" в галактику. Думайте о них как о рамках, пронумерованных от 1 до максимального числа объемов.

Размещая цивилизации, вы просто выбираете номер рамки наугад. Дублирующие выборки указывают на несколько цивилизаций в одной рамке. Допустим, что цивилизации в одной рамке способны друг друга обнаружить. Это не совсем верно (рис. 10.4), но поскольку вы будете использовать большое число цивилизаций, то расхождения будут тяготеть к обнулению друг друга, как при суммировании длинного ряда округленных чисел.

Во избежание необходимости повторять это упражнение всякий раз, когда вы изменяете число цивилизаций и/или размеры радиопузырей, результаты можно за-

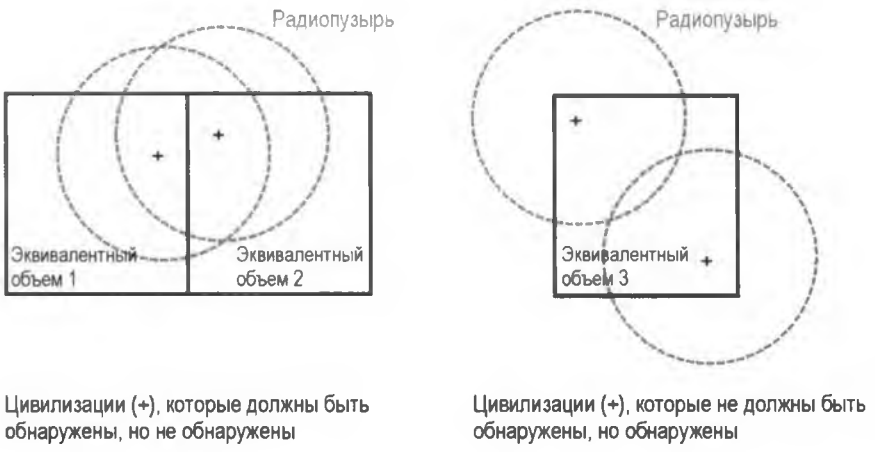


Рис. 10.4. Проблемы обнаружения на уровне индивидуального эквивалентного объема

фиксировать в виде формулы — *многочленного уравнения*, которое можно использовать для генерирования всех будущих оценок вероятности. *Многочлен* — это сумма или разность множества алгебраических членов. Всем известное *квадратное уравнение*, которое мы все изучали в школе, является многочленным уравнением второй степени (т. е. показатели степени переменных не превышают 2):

$$ax^2 + bx + c = 0.$$

Многочлены порождают хорошие кривые, поэтому они идеально подходят к этой задаче. Но для того чтобы указанная формула работала с переменным числом цивилизаций и размерами пузырей, вам нужно будет использовать *отношение* числа цивилизаций к суммарному объему. Суммарный объем представлен шкалированным объемом диска, который совпадает с суммарным числом эквивалентных объемов.

На рис. 10.5 каждая точка представляет вероятность обнаружения для отношения ниже ее. Показанное на рисунке уравнение является многочленным выражением,

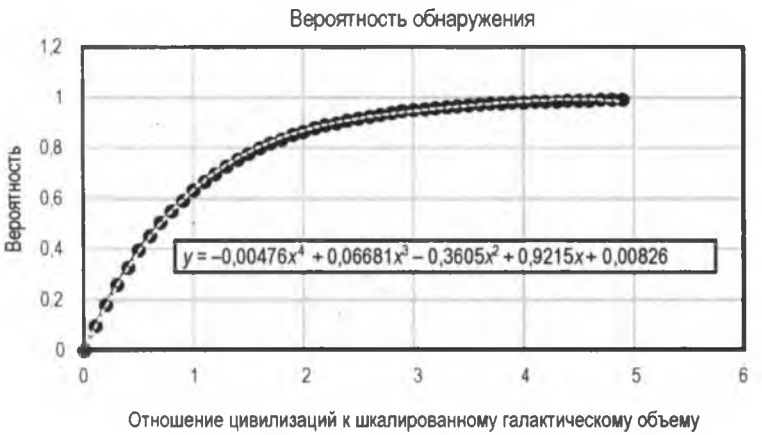


Рис. 10.5. Вероятность обнаружения в зависимости от отношения числа цивилизаций к шкалированному объему галактики

которое генерирует линию, соединяющую точки. С помощью этой формулы можно предсказать вероятность любого отношения цивилизаций в расчете на объем, вплоть до значения 5 (выше этого мы просто будем считать, что вероятность равна 1,0).

На рис. 10.5 отношения цивилизация к объему разнесены по оси *x*. Отношение 0,5, например, означает, что существует в два раза меньше цивилизаций, чем имеется эквивалентных объемов радиопузырей, отношение 2 означает, что существует в два раза больше цивилизаций, чем объемов, и т. д. Ось *y* — это вероятность того, что эквивалентный объем содержит более одной цивилизации.

На рис. 10.5 следует отметить еще одну вещь — для обеспечения того, чтобы у всех цивилизаций был сосед по комнате, требуется много цивилизаций. Представьте, что 999 999 из 1 000 000 эквивалентных объемов содержат по крайней мере две цивилизации и вы задействуете свои богоподобные силы, случайно размещая одну новую цивилизацию. Существует шанс один на миллион, что эта новая цивилизация окажется в оставшемся объеме с одним обитателем. Этот последний эквивалентный объем является пресловутой иголкой в стоге сена!

ПРИМЕЧАНИЕ

Аксиома компьютерного моделирования состоит в том, чтобы начинать с простого и постепенно добавлять сложность. Самое простое "базовое" допущение состоит в том, что развитые цивилизации случайно распределены по всей галактике. В *разд. "Сложные проекты" далее в этой главе* вы получите возможность оспорить это допущение, используя концепцию галактических обитаемых зон.

Код вероятности обнаружения

Программный код вероятности обнаружения случайно выбирает местоположения (эквивалентные объемы радиопузырей) для заданного числа местоположений и цивилизаций, подсчитывает, сколько местоположений встречается только один раз (т. е. содержит только одну цивилизацию), и повторяет эксперимент несколько раз, сходясь на оценке вероятности. Затем этот процесс повторяется для нового числа цивилизаций. Результат представлен в виде вероятности против отношения цивилизаций на объем вместо фактического числа цивилизаций и превратился в многочленное выражение, вследствие чего результаты легко переносятся. А это означает, что указанную программу требуется выполнить всего один раз.

Для генерирования многочленного уравнения и проверки того, что оно вписывается в данные, вы будете использовать библиотеки `NumPy` и `matplotlib`. Библиотека `NumPy` привносит поддержку больших многоразмерных массивов и матриц и включает больше число математических функций, которые с ними работают. Библиотека `matplotlib` поддерживает двухмерное и рудиментарное трехмерное графопостроение, а `NumPy` обеспечивает ей численное математическое расширение.

Существует несколько способов установки этих научных дистрибутивов Python. Один из способов — использовать `SciPy`, библиотеку Python с открытым исходным кодом, применяемую для научных и технических вычислений (см. **https://scipy.org/**

index.html). Если вы намерены много заниматься анализом данных и графопостроением, то можете скачать и использовать бесплатный пакет, такой как Anaconda или Enthought Canopy, который работает с Windows, Linux и macOS. Эти пакеты избавят вас от задачи поиска и установки всех необходимых библиотек для науки о данных в нужной версии. Список такого рода пакетов, а также ссылки на их веб-сайты можно найти по адресу **https://scipy.org/install.html**.

Как вариант можно скачать эти продукты напрямую с помощью менеджера пакетов pip. Я сделал это, используя инструкции по адресу **https://scipy.org/install.html**. Поскольку библиотека `matplotlib` требует большого числа модулей, от которых она зависит, они должны быть установлены одновременно. В Windows я выполнил следующую специфичную для Python 3 команду из интерактивной оболочки PowerShell, запущенную из моей папки Python35 (тройку в `python3` можно убрать, если у вас не установлено несколько версий):

```
$ python3 -m pip install --user numpy scipy matplotlib ipython jupyter pandas
sympy nose
```

Все остальные модули, которые вам понадобятся, поставляются в комплекте с Python. Что касается программного кода в листингах 10.1 и 10.2, то вы можете набрать его самостоятельно либо скачать его копию с веб-сайта книги **https://www.nostarch.com/impracticalpython/**.

Вычисление вероятности обнаружения для диапазона цивилизаций

Листинг 10.1 импортирует модули и выполняет всю только что описанную работу, за исключением вписывания многочлена и изображения результатов проверки качества на основе библиотеки `matplotlib`.

Листинг 10.1. Импортирует модули, случайно выбирает местоположения радиопузырей, эквивалентных по объему, и вычисляет вероятность наличия нескольких цивилизаций в каждом местоположении.
Файл probability_of_detection.py, часть 1

```
❶ from random import randint
  from collections import Counter
  import numpy as np
  import matplotlib.pyplot as plt

❷ NUM_EQUIV_VOLUMES = 1000      # число местоположений,
                                # в которые помещать цивилизации
  MAX_CIVS = 5000               # максимальное число развитых цивилизаций
  TRIALS = 1000                 # число моделирований заданного
                                # числа цивилизаций
  CIV_STEP_SIZE = 100           # размер шага подсчета цивилизаций

❸ x = []      # значения x для вписывания многочлена
  y = []      # значения y для вписывания многочлена
```

```
❹ for num_civs in range(2, MAX_CIVS + 2, CIV_STEP_SIZE):
      civs_per_vol = num_civs / NUM_EQUIV_VOLUMES
      num_single_civs = 0
  ❺ for trial in range(TRIALS):
        locations = []  # эквивалентные объемы, содержащие цивилизацию
    ❻ while len(locations) < num_civs:
          location = randint(1, NUM_EQUIV_VOLUMES)
          locations.append(location)
    ❼ overlap_count = Counter(locations)
      overlap_rollup = Counter(overlap_count.values())
      num_single_civs += overlap_rollup[1]

  ❽ prob = 1 - (num_single_civs / (num_civs * TRIALS))

    # напечатать отношение цивилизаций на объем против
    # вероятности 2 + цивилизаций на местоположение
  ❾ print("{:.4f} {:.4f}".format(civs_per_vol, prob))
  ❿ x.append(civs_per_vol)
    y.append(prob)
```

Импортируйте уже знакомые модуль `random` и контейнерный тип `Counter` для подсчета числа цивилизаций в каждом местоположении (обозначаемом по тому, сколько раз местоположение было выбрано) ❶. Работа контейнерного типа `Counter` будет объяснена через минуту. Импортированные библиотеки `NumPy` и `matplotlib` будут использоваться для вписывания многочлена и его графопостроения на экране.

Задайте несколько констант, которые представляют введенные пользователем значения для числа эквивалентных объемов, максимального числа цивилизаций, числа испытаний (т. е. сколько раз следует повторить эксперимент для заданного числа цивилизаций), и размера шага для подсчета количества ❷. Поскольку результаты являются предсказуемыми, то без ущерба для точности можно использовать большое значение шага в размере 100. Обратите внимание, что вы получите очень похожие результаты независимо от того, каким будет число эквивалентных объемов: 100 или 100 000+.

Для многочленного выражения вам понадобится серия парных значений (x, y), поэтому создайте два списка для хранения этих значений ❸. Значение x будет отношением цивилизаций к объему, а значение y — соответствующей вероятностью обнаружения.

Начните серию вложенных циклов, причем самый внешний цикл представляет число цивилизаций в модели ❹. Для того чтобы одна из них обнаружила другую, вам потребуются по крайней мере две цивилизации. Установите максимум равным `MAX_CIVS` плюс 2, для того чтобы перерегулировать (overshoot — проскочить) многочлен при вычислении. Для значения шага используйте константу `CIV_STEP_SIZE`.

Далее вычислите совокупное отношение `civs_per_vol` и создайте счетчик с именем `num_single_civs` для отслеживания числа местоположений, содержащих одну-единственную цивилизацию.

Вы выбрали число распределяемых цивилизаций, поэтому теперь примените цикл `for` для прохода по числу испытаний ❺. Для каждого испытания вы распределяете одно и то же число цивилизаций. Назначьте пустой список переменной `locations`, а затем для каждой цивилизации ❺ выберите произвольный номер местоположения и добавьте его в список. Дублирующие значения в списке будут представлять местоположения, содержащие несколько цивилизаций.

Выполните `Counter` на этом списке ❺ и получите значения. Завершите цикл, получив число местоположений, которые встречаются только один раз, и добавьте их в счетчик `num_single_civs`. Вот пример того, как эти три инструкции работают:

```
>>> from collections import Counter
>>> alist = [124, 452, 838, 124, 301]
>>> count = Counter(alist)
>>> count
Counter({124: 2, 452: 1, 301: 1, 838: 1})
>>> value_count = Counter(count.values())
>>> value_count
Counter({1: 3, 2: 1})
>>> value_count[1]
3
```

Список `alist` содержит пять чисел, причем одно из них (124) дублируется. При выполнении контейнерного типа `Counter` на этом списке создается словарь с числами в качестве ключей и количеством их появлений в качестве значений. Передачей значения контейнерного типа `Counter` в переменную `count` — с помощью метода `values()` — создается еще один словарь с предыдущими значениями в качестве ключей и количеством их появлений в качестве новых значений. Вы хотите знать, сколько чисел встречается только один раз, поэтому для возврата количества неупорядоченных чисел используйте метод словаря `value_count[1]`. Они, разумеется, будут представлять объемы эквивалентных радиопузырей, содержащих одну-единственную цивилизацию.

Теперь примените результаты, полученные от контейнерного типа `Counter`, для вычисления вероятности нескольких цивилизаций в расчете на местоположение для текущего числа распределяемых цивилизаций ❾. Это 1 минус число одноместных расположений, деленное на число цивилизаций в каждом испытании, умноженное на число испытаний.

Проследите за этим вычислением, печатая это соотношение цивилизаций к объему и вероятность того, что несколько цивилизаций имеют общее местоположение ❽. Первые несколько строк этого вычисления выглядят следующим образом:

```
0.0020 0.0020
0.1020 0.0970
0.2020 0.1832
```

```
0.3020 0.2607
0.4020 0.3305
0.5020 0.3951
0.6020 0.4516
0.7020 0.5041
```

Данная распечатка служит в качестве первоначального шага контроля качества и является необязательной; в случае, если вы хотите ускорить время выполнения, то ее следует закомментировать. Завершите программу, добавив значения в списки `x` и `y` ❷.

Генерирование предсказательной формулы и проверка результатов

В листинге 10.2 применяется библиотека `NumPy` для выполнения многочленной регрессии вероятности обнаружения против соотношения цивилизаций на объем, рассчитанного в листинге 10.1. Это многочленное уравнение будет использоваться в следующей ниже программе получения оценочных значений вероятности. Для проверки того, что результирующая кривая вписывается в точки данных, библиотека `matplotlib` строит график фактических и предсказываемых значений.

Листинг 10.2. Выполняет многочленную регрессию и строит график проверки ее качества. Файл probability_of_detection.py, часть 2

```
❶ coefficients = np.polyfit(x, y, 4) # вписывание многочлена 4-й степени
❷ p = np.poly1d(coefficients)
  print("\n{}".format(p))
❸ xp = np.linspace(0, 5)
❹ _ = plt.plot(x, y, '.', xp, p(xp), '-')
❺ plt.ylim(-0.5, 1.5)
❻ plt.show()
```

Начните с назначения переменной `coefficients` результата метода `polyfit()` библиотеки `NumPy` ❶. Указанный метод в качестве аргументов принимает списки `x` и `y`, а также целое число, представляющее степень подгонки многочлена. Он возвращает вектор коэффициентов `p`, который минимизирует квадратическую ошибку.

Если вы напечатаете переменную `coefficients`, то на выходе получите следующее:

```
[-0.00475677   0.066811  -0.3605069   0.92146096   0.0082604 ]
```

Для получения полного выражения передайте переменную `coefficients` в метод `poly1d()` и назначьте результаты новой переменной ❷. Напечатайте эту переменную, и вы увидите уравнение, аналогичное показанному на рис. 10.5:

```
           4           3          2
-0.004757 x + 0.06681 x - 0.3605 x + 0.9215 x + 0.00826
```

С целью проверки адекватности воспроизведения многочленом входных данных вы захотите построить взаимосвязь отношения цивилизаций к объему на оси *x* от вероятности на оси *y*. Для получения значений по оси *x* можно воспользоваться методом `linspace()` библиотеки `NumPy`, который возвращает числа, равномерно расположенные через заданный интервал ❸. Используйте диапазон `(0, 5)`, благодаря чему будет покрыт почти весь вероятностный диапазон.

Для того чтобы разнести символы, соответствующие вычисленным и предсказанным значениям, сначала передайте методу `plot()` списки `x` и `y`, выводя их на график с помощью точки (`'.'`), которая эквивалентна точкам на рис. 10.5 ❹. Затем передайте значения, предсказанные для оси *x* (`xp`), и для получения вероятности, предсказанной для оси *y*, передайте функции `p` ту же самую переменную, выведя результаты на график с помощью тире (`'-'`).

Закончите программу, ограничив ось *y* значениями `-0.5` и `1.5` ❺ и применив метод `show()`, который фактически покажет график на экране (рис. 10.6) ❻. Результирующий график является простым и разреженным, поскольку его единственная цель — подтвердить, что многочленная регрессия работает по назначению. Многочленную подгонку можно изменить, увеличив или уменьшив третий аргумент в шаге ❶.

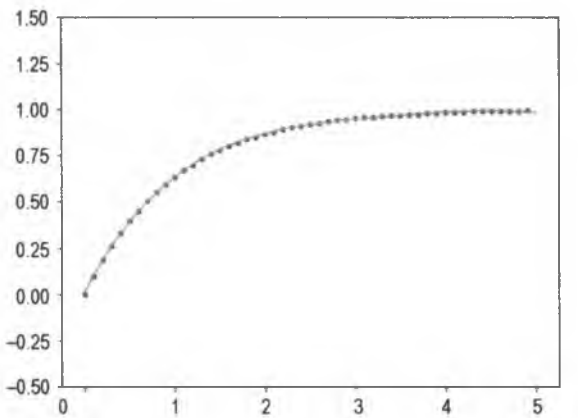


Рис. 10.6. Вычисленные результаты (точки) против результатов, предсказанных многочленом (линия)

Вооружившись этими результатами, вы теперь способны в мгновение ока оценить вероятность обнаружения для любого числа цивилизаций. Языку Python нужно только решить многочленное уравнение.

Построение графической модели

Графическая модель будет представлять собой двухмерный вид галактического диска сверху вниз. Нанесение размера нынешнего эмиссионного пузыря Земли на этом изображении позволит увидеть в перспективе как размер галактики, так и наше крошечное место в ней.

Моделирование Млечного Пути всецело касается моделирования спиральных рукавов. Каждый спиральный рукав представляет собой логарифмическую спираль — геометрическую фигуру, настолько распространенную в природе, что ее окрестили *spira mirabilis* — "чудесной спиралью". Если вы сравните рис. 10.7 с рис. 10.1, то сможете увидеть, насколько близко структура урагана напоминает структуру галактики. Глаз урагана можно даже представить, как сверхмассивную черную дыру, где облачный вихрь вокруг ядра ("глаз бури") представляет горизонт событий!

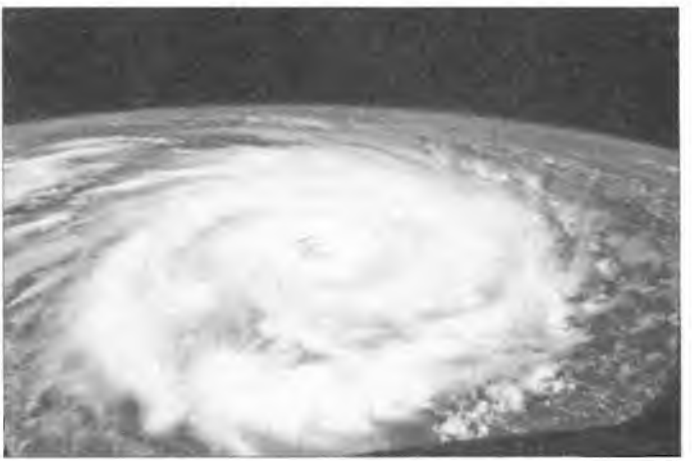


Рис. 10.7. Ураган Игорь

Поскольку спирали расходятся от центральной точки, или полюса, построить их график будет легче с помощью полярных координат (рис. 10.8).

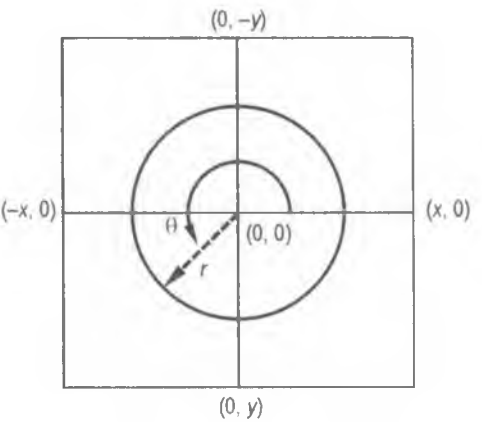


Рис. 10.8. Пример системы полярных координат

При переходе к полярным координатам координаты (x, y), используемые в более знакомой декартовой системе координат, заменяются парами (r, θ), где r — это расстояние от центра, а θ — угол, образованный радиусом r и осью x. Координаты полюса равны (0, 0).

Полярное уравнение логарифмической спирали имеет вид:

$$r = ae^{b\theta},$$

где r — расстояние от начала координат; θ — угол от оси x; e — основание натурального логарифма; a и b — произвольные константы.

Указанную формулу можно использовать для начертания одиночной спирали; затем повернуть и перечертить спираль три раза, получив четыре рукава Млечного Пути. В результате получатся спирали из кругов разных размеров, которые будут представлять звезды. На рис. 10.9 приведен пример одной такой имплементации графической модели. Поскольку симуляции являются стохастическими, то каждая из них будет немного отличаться, причем в модели имеется много переменных, которые можно отрегулировать, изменив ее внешний вид.

Рис. 10.9. Млечный Путь, смоделированный с использованием логарифмических спиралей

Я сгенерировал изображение на рис. 10.9 с помощью `tkinter` (произносится "ти-кэйинтер") — библиотеки графического интерфейса, используемой по умолчанию, для разработки настольных приложений на Python. Хотя `tkinter` в основном предназначена для графических элементов, таких как окна, кнопки, полосы прокрутки и т. д., она также может создавать графики, диаграммы, заставки, простые игры и многое другое. Одним из ее преимуществ является то, что в рамках стандартного дистрибутива Python она переносится во все операционные системы, и нет необходимости устанавливать внешние библиотеки. Она также хорошо документирована и проста в использовании.

Большинство машин на основе Windows, macOS и Linux поставляются с предустановленной библиотекой `tkinter`. Если у вас ее нет или вам нужна ее последняя версия, то ее можно скачать и установить с **https://www.activestate.com/**. Как всегда, если модуль уже установлен, то его можно без ошибок импортировать в окно интерпретатора:

```
>>> import tkinter
>>>
```

Книги с вводными курсами по Python иногда включают обзоры библиотеки `tkinter`, а официальную онлайновую документацию можно найти по адресу **https://docs.python.org/3/library/tk.html**. Несколько других ссылок на `tkinter` включены в *разд. "Дальнейшее чтение" далее в этой главе*.

Шкалирование графической модели

Шкала графической модели находится в световых годах на пиксел, а ширина каждого пиксела будет равна диаметру радиопузыря. В связи с этим, когда исследуемый радиопузырь меняет диаметр, меняются и единицы шкалы, и графическую модель необходимо будет перестроить. Следующее ниже уравнение шкалирует модель в пузырь:

$$\text{шкалированный радиус диска} = \frac{\text{радиус диска}}{\text{диаметр радиопузыря}},$$

где радиус диска равен 50 000, а единица длины — это световые годы.

Когда отобранный радиопузырь мал, графическая модель "увеличивается", а когда он велик, то она "уменьшается" (рис. 10.10).

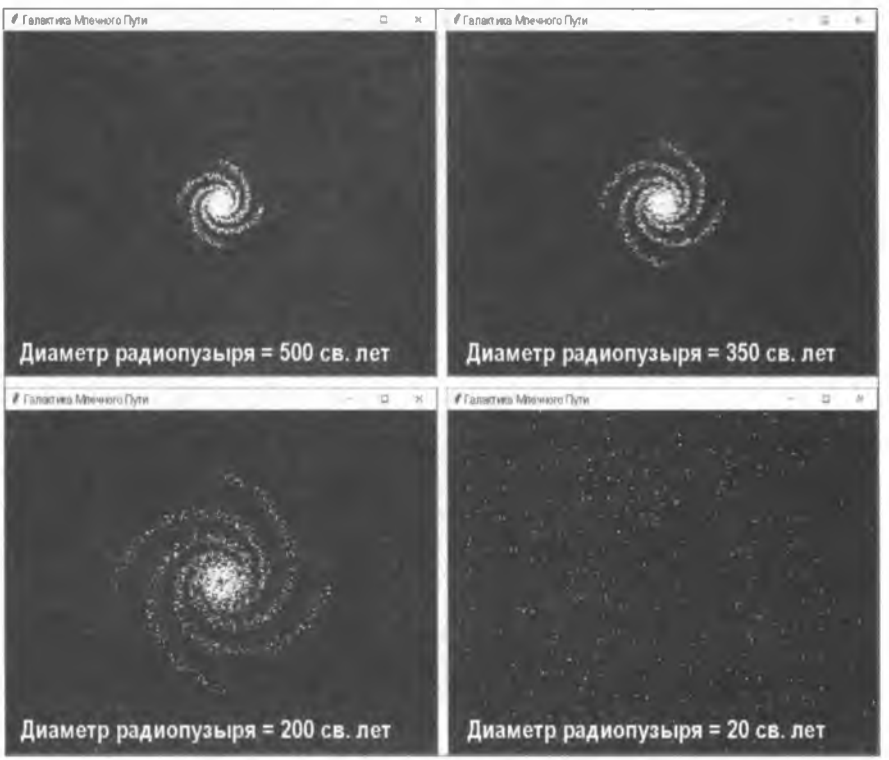


Рис. 10.10. Влияние диаметра радиопузыря на внешний вид галактической модели

Код симулятора галактики

Программный код симулятора галактики будет вычислять вероятность обнаружения для любого числа цивилизаций и размеров радиопузырей, а затем сгенерирует графическую галактическую модель. Когда используется пузырь размером с наш текущий эмиссионный пузырь, он разместит и проаннотирует наш пузырь красным цветом в приблизительном местоположении нашей Солнечной системы. Указанный код можно скачать код с веб-сайта

https://www.nostarch.com/impracticalpython/.

Ввод данных и ключевых параметров

Листинг 10.3 начинает программу galaxy_simulator.py с импорта модулей и размещения вверху часто запрашиваемой информации о вводимых пользователем значениях.

Листинг 10.3. Импортирует модули и задает константы. Файл galaxy_simulator.py, часть 1

```
❶ import tkinter as tk
  from random import randint, uniform, random
  import math
  #======================================================================
❷ # ГЛАВНЫЕ ВХОДНЫЕ ДАННЫЕ

  # шкала (диаметра радиопузыря) в световых годах:
❸ SCALE = 225  # ввести 225, чтобы увидеть радиопузырь Земли

  # число развитых цивилизаций из уравнения Дрейка:
❹ NUM_CIVS = 15600000
  #======================================================================
```

Для того чтобы не набирать полное имя модуля `tkinter` при вызове его классов, импортируйте указанный модуль как `tk` ❶. Если вы работаете с Python 2, то используйте `Tkinter` — с прописной `T`. Вам также понадобятся модули `random` и `math`.

Используйте комментарий, для того чтобы выделить главный раздел вводимых пользователем данных ❷, и задайте два входных значения. Задайте: `SCALE` для диаметра обнаруживаемого электромагнитного пузыря вокруг каждой цивилизации в световых годах ❸; `NUM_CIVS` — для числа моделируемых цивилизаций, которое можно задать, используя все что угодно, начиная с уравнения Дрейка до чистой догадки ❹.

Настройка холста *tkinter* и задание значений константам

Программный код в листинге 10.4 создает экземпляр объекта окна `tkinter` с холстом, на котором можно рисовать объекты. Здесь появится карта галактики, или

графическая модель. Он также задает значения констант, связанных с размерами Млечного Пути.

Листинг 10.4. Настраивает окно и холст модуля `tkinter` и задает константы. Файл galaxy_simulator.py, часть 2

```
    # настроить изображение холста
❶   root = tk.Tk()
    root.title("Галактика Млечного Пути")
❷   c = tk.Canvas(root, width=1000, height=800, bg='black')
❸   c.grid()
❹   c.configure(scrollregion=(-500, -400, 500, 400))

    # фактические размеры Млечного Пути (св. годы)
❺   DISC_RADIUS = 50000
    DISC_HEIGHT = 1000
❻   DISC_VOL = math.pi * DISC_RADIUS**2 * DISC_HEIGHT
```

Начните с создания окна с обычным именем `root` (корневое) ❶. Оно является окном верхнего уровня, которое будет содержать все остальное. В следующей строке кода назначьте окну заголовок — `"Галактика Млечного Пути"`, который появится в левом верхнем углу рамки окна (рис. 10.9 с примером).

Далее в корневое окно добавьте компонент графического интерфейса, именуемый *виджетом*. Английское слово widget расшифровывается как "Windows gadget" (графический компонент в Windows). В `tkinter` существует 21 стержневой виджет, включая надписи, рамки, переключатели и полосы прокрутки. Задайте виджет холста `Canvas`, который будет содержать все рисуемые объекты ❷. Указанный виджет является универсальным и предназначен для графики и других сложных макетов. Укажите родительское окно, ширину и высоту экрана, а также цвет фона. Задайте имя холста как `c` (от англ. *canvas*).

Виджет `canvas` можно разделить на строки и столбцы, как обычную или электронную таблицу. Каждая ячейка в этой решетке может содержать другой виджет, и эти виджеты могут охватывать несколько ячеек. Внутри ячейки виджет можно выровнять с помощью параметра прилипания `STICKY`. Для управления каждым виджетом в окне необходимо использовать диспетчер геометрии `grid`. Поскольку в этом проекте вы используете только один виджет, то в менеджер передавать ничего не потребуется ❸.

Завершите конфигурирование холста `canvas`, настроив использование области прокрутки `scrollregion` ❹. В результате начало координат `(0, 0)` будет установлено в центр холста. Это нужно для того, чтобы нарисовать спиральные рукава галактики с помощью полярных координат. Без этого по умолчанию началом координат будет левый верхний угол холста `Canvas`.

Аргументы, переданные в `configure`, задают пределы холста `canvas`. Они должны составлять половину ширины и высоты холста; например, пределы прокрутки `600`,

500 потребуют размеров холста 1200, 1000. Показанные здесь значения хорошо работают на небольшом ноутбуке, но не стесняйтесь их позже поменять, если вам требуется более крупное окно.

После раздела входных данных следуют размерные константы галактики ❺. Некоторые из этих переменных можно задать в функциях, но наличие их в глобальном пространстве делает объяснение кода логичнее. Первые две — это радиус и высота галактического диска из рис. 10.2. Последняя константа представляет собой объем диска ❻.

Шкалирование галактики и вычисление вероятности обнаружения

В листинге 10.5 определены функции шкалирования размеров галактик (на основе диаметра используемого радиопузыря) и вычисления вероятности обнаружения одной цивилизацией другой. Последняя функция — это то место, где вы применяете многочленное уравнение, построенное с помощью описанной ранее программы probability_of_detection.py.

Листинг 10.5. Шкалирование размеров галактики и вычисление вероятности обнаружения. Файл galaxy_simulator.py, часть 3

```
❶ def scale_galaxy():
      """ Шкалирование размеров галактики на основе
          размера радиопузыря (scale)."""
      disc_radius_scaled = round(DISC_RADIUS / SCALE)
   ❷ bubble_vol = 4/3 * math.pi * (SCALE / 2)**3
   ❸ disc_vol_scaled = DISC_VOL/bubble_vol
   ❹ return disc_radius_scaled, disc_vol_scaled

❶ def detect_prob(disc_vol_scaled):
      """Вычислить вероятность, что галактические цивилизации
         обнаружат друг друга."""
      # отношение цивилизаций к шкалированному объему галактики
   ❻ ratio = NUM_CIVS / disc_vol_scaled
   ❼ if ratio < 0.002:        # установить очень низкие соотношения
          detection_prob = 0   # равными нулевой вероятности
      elif ratio >= 5:         # установить верхние соотношения
          detection_prob = 1   # равными единичной вероятности
   ❽ else:
          detection_prob = -0.004757 * ratio**4 +  \
                           0.06681 * ratio**3 - 0.3605 * \
                           ratio**2 + 0.9215 * ratio + 0.00826
   ❾ return round(detection_prob, 3)
```

Определите функцию scale_galaxy(), которая будет шкалировать размеры галактики в размер радиопузыря ❶. Она будет использовать константы из глобального

пространства, поэтому передавать ей какие-либо аргументы не потребуется. Вычислите шкалированный радиус диска, а затем объем радиопузыря, используя уравнение объема сферы, и назначьте результаты переменной `bubble_vol` ❸.

Затем разделите фактический объем диска на `bubble_vol`, получив прошкалированный объем диска ❸. Это число показывает количество "эквивалентных" по радиопузырям объемов, которые могут поместиться в галактике. Каждый пузырь представляет возможное местоположение цивилизации.

Завершите работу функции, вернув переменные `disc_radius_scaled` и `disc_vol_scaled` ❹.

Теперь определите функцию `detect_prob()`, которая будет вычислять вероятность обнаружения. В качестве аргумента она принимает прошкалированный объем диска ❸. Для члена `x` в многочлене вычислите отношение числа цивилизаций к прошкалированному объему диска ❸. Поскольку многочленная регрессия в конечных точках может иметь проблемы, примените условные выражения, которые установят очень малые соотношения равными 0 и большие соотношения равными 1 ❼. В противном случае примените многочленное выражение, сгенерированное кодом probability_of_detection.py ❸, затем верните вероятность, округленную до трех знаков после точки ❹.

Использование полярных координат

Листинг 10.6 определяет функцию отбора случайных местоположений (x, y) с использованием полярных координат. Указанная функция будет выбирать местоположение нескольких звезд, размещенных в графической модели. Поскольку изображение является двухмерным, отсутствует необходимость выбирать местоположение координаты z.

Листинг 10.6. Определяет функцию случайного подбора пары (x, y) с помощью полярных координат. Файл galaxy_simulator.py, часть 4

```
❷ def random_polar_coordinates(disc_radius_scaled):
      """Сгенерировать случайную (x, y) внутри диска
         из равномерного распределения для 2-мерного изображения."""
   ❷ r = random()
   ❷ theta = uniform(0, 2 * math.pi)
   ❹ x = round(math.sqrt(r) * math.cos(theta) * disc_radius_scaled)
     y = round(math.sqrt(r) * math.sin(theta) * disc_radius_scaled)
   ❺ return x, y
```

Указанная функция в качестве аргумента принимает прошкалированный радиус диска ❶. Используйте функцию `random()` для выбора вещественного значения в диапазоне от 0,0 до 1,0 и назначьте его переменной `r` ❸. Далее, из равномерного распределения случайно выберите `theta` между 0 и 360 градусами (2π — это радианный эквивалент 360°) ❸.

Преобразование, равномерно генерирующее точки на единичном диске, выглядит следующим образом:

$$x = \sqrt{r} \cdot \cos\theta ;$$

$$y = \sqrt{r} \cdot \sin\theta .$$

Указанные уравнения дают значения (x, y) в диапазоне от 0 до 1. Для шкалирования результатов в галактический диск умножьте на прошкалированный радиус диска ❹. Завершите функцию, вернув `x` и `y` ❺.

Построение спиральных рукавов

Листинг 10.7 определяет функцию, которая строит спиральные рукава, используя уравнение логарифмической спирали. Данная спираль может показаться волшебной, но бóльшая часть волшебства крутится вокруг первоначальной голой спирали, облекая рукав плотью. Вы достигнете этого, варьируя размер звезд, случайно меняя их местоположение на крошечное число и дублируя спираль для каждого рукава, немного сдвигая ее назад и приглушая ее звезды.

Листинг 10.7. Определяет функцию `spirals()`. Файл galaxy_simulator.py, часть 5

```
❶ def spirals(b, r, rot_fac, fuz_fac, arm):
      """Построить спиральные рукава для изображения в tkinter,
         используя формулу логарифмической спирали.

      b - произвольная константа в уравнении логарифмической спирали
      r - радиус прошкалированного галактического диска
      rot_fac - коэффициент поворота
      fuz_fac - случайный сдвиг в позиции звезды в рукаве,
                применительно к переменной 'fuzz'
      arm - рукав спирали (0 - главный рукав, 1 - замыкающие звезды)
   """
   ❷ spiral_stars = []
   ❸ fuzz = int(0.030 * abs(r)) # случайно сдвинуть местоположения звезд
     theta_max_degrees = 520
   ❹ for i in range(theta_max_degrees):     # range(0, 600, 2) не для черной
                                            # дыры
         theta = math.radians(i)
         x = r * math.exp(b * theta) * math.cos(theta + \
             math.pi * rot_fac) + randint(-fuzz, fuzz) * fuz_fac
         y = r * math.exp(b * theta) * math.sin(theta + math.pi * \
             rot_fac) + randint(-fuzz, fuzz) * fuz_fac
         spiral_stars.append((x, y))
   ❺ for x, y in spiral_stars:
      ❻ if arm == 0 and int(x % 2) == 0:
            c.create_oval(x-2, y-2, x+2, y+2, fill='white', outline='')
         elif arm == 0 and int(x % 2) != 0:
            c.create_oval(x-1, y-1, x+1, y+1, fill='white', outline='')
```

```
❼ elif arm == 1:
      c.create_oval(x, y, x, y, fill='white', outline='')
```

Определите функцию `spirals()` ❶. Ее параметры перечислены в литерале документирования функции. Первые два параметра, `b` и `r`, принадлежат уравнению логарифмической спирали. Следующий, `rot_fac`, является коэффициентом поворота, который позволяет перемещать спираль вокруг центральной точки для обеспечения возможности создания нового спирального рукава. Коэффициент напыления `fuz_fac` позволяет отрегулировать то, как далеко вы перемещаете звезды от центра спиральной линии. Наконец, параметр `arm` позволяет указать либо ведущий рукав, либо тянущийся позади рукав из затухающих звезд. Тянущийся позади рукав будет сдвинут, т. е. нанесен немного позади ведущего рукава, и его звезды будут меньше.

Инициализируйте пустой список для хранения местоположения звезд, которые будут составлять спираль ❷. Задайте переменную `fuzz`, где вы умножаете произвольную константу на абсолютное значение радиуса прошкалированного диска ❸. Звезды создаются только спиральным уравнением — они выстраиваются в линию (см. две левые части на рис. 10.11). Напыление будет немного перемещать звезды по спирали вперед и назад, по обе стороны от спиральной линии. Его эффект можно увидеть на ярких звездах в самой правой панели рис. 10.11. Я определил эти значения методом проб и ошибок; не стесняйтесь поэкспериментировать с ними, если захотите.

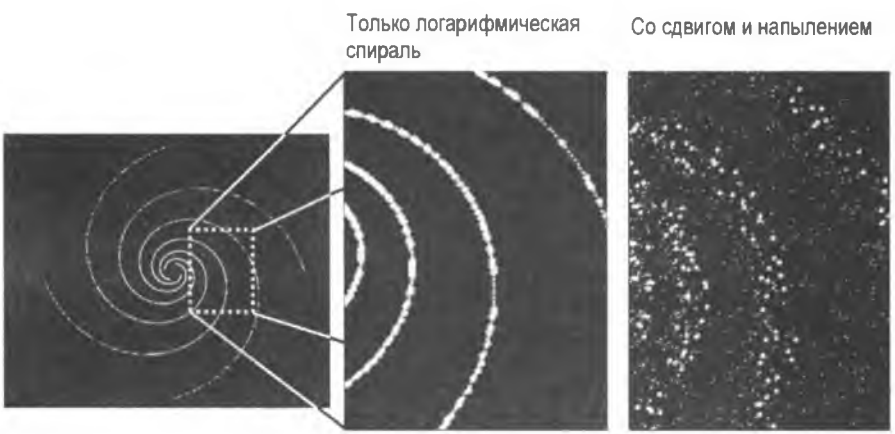


Рис. 10.11. Заполнение спиральных рукавов путем сдвига спиралей и случайного изменения местоположения звезд

Теперь самое время построить спиральные линии. Сначала примените диапазон значений, которые будут представлять θ в уравнении логарифмической спирали ❹. Диапазон около `520` произведет галактику, как на рис. 10.9, которая имеет центральную "черную дыру". В противном случае используйте диапазон `(0, 600, 2)` — или аналогичный — для получения яркого центрального ядра, полностью заполненного звездами (рис. 10.12). С этими значениями вы можете повозиться до тех пор, пока не получите желаемый результат. Пройдите в цикле по значениям в `theta`

и примените уравнение логарифмической спирали, используя косинус для значения x и синус для значения y. Обратите внимание, что к результату добавляется значение `fuzz`, умноженное на коэффициент напыления. Добавьте каждую пару (x, y) в список `spiral_stars`.

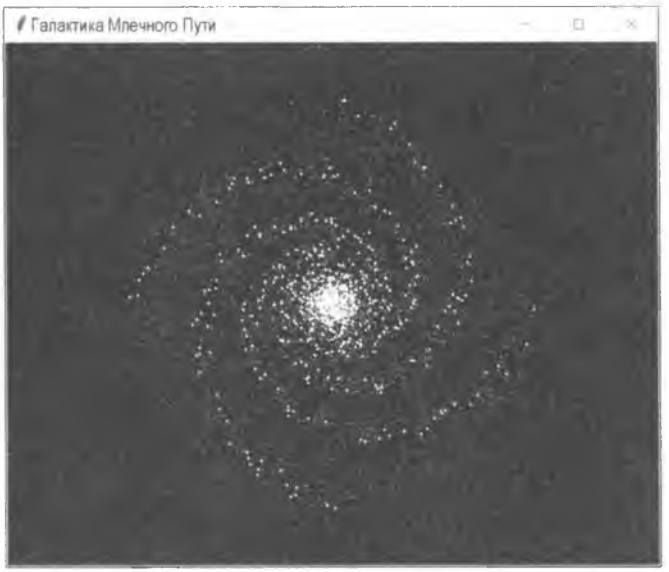


Рис. 10.12. Графическая модель без центральной черной дыры (сравните с рис 10.9)

Позже, в функции `main()`, вы сможете указать переменную `rot_fac`, которая будет двигать спираль вокруг центра. После того как программа построит четыре основных рукава, она применит `rot_fac` для построения четырех новых рукавов, слегка сдвинутых от первых четырех, создавая полосу тусклых, тянущихся за ними звезд, видимых слева от каждой дуги, состоящей из ярких звезд, на рис. 10.11.

Теперь, когда у вас есть список местоположений звезд, начните цикл `for` по координатам (x, y) ❺. Затем примените условную инструкцию для выбора главного, ведущего рукава и местоположений, для которых `x` является четным ❻. Для этого используйте метод `create_oval()` виджета `canvas`, который создает размещаемый на графике звездный объект. Первые четыре аргумента этого метода определяют ограничительную рамку, в которую помещается овал. Чем крупнее число после `x` и `y`, тем больше овал. Сделайте заливку белым цветом и не используйте контур; по умолчанию контур изображается тонкой черной линией.

Если значение `x` является нечетным, то сделайте звезду на шаг меньше. И если значение рукава равно 1, то звезда находится в смещенном рукаве, поэтому сделайте ее как можно меньше ❼.

ПРИМЕЧАНИЕ

Звездные объекты предназначены только для визуального воздействия. Ни их размер, ни количество не подлежит шкалированию. На самом деле они должны иметь гораздо меньший размер и быть намного многочисленнее (более 100 млрд!).

Рассеяние звездной дымки

Пространство между спиральными рукавами не лишено звезд, поэтому следующая функция (листинг 10.8) случайно рассеивает точки в галактической модели без учета спиральных рукавов. Думайте об этом как о свечении, которое вы видите на фотографиях далеких галактик.

Листинг 10.8. Определяет функцию `star_haze()`. Файл galaxy_simulator.py, часть 6

```
❶ def star_haze(disc_radius_scaled, density):
      """Случайно распределите тусклые звезды в галактическом диске.
      disc_radius_scaled = радиус галактического диска,
                           прошкалированный в плотность радиопузыря
      density = множитель для варьирования числа размещенных звезд
      """
   ❷ for i in range(0, disc_radius_scaled * density):
      ❸ x, y = random_polar_coordinates(disc_radius_scaled)
      ❹ c.create_text(x, y, fill='white', font=('Helvetica', '7'),
                      text='.')
```

Определите функцию `star_haze()` и передайте ей два аргумента: прошкалированный радиус диска и целочисленный множитель, которые функция будет использовать для увеличения базового числа случайных звезд ❶. Поэтому если вместо легкой дымки вы предпочитаете густой туман, то при вызове указанной функции в `main()` увеличьте значение плотности.

Начните цикл `for`, где максимальное значение диапазона равно прошкалированному радиусу диска, умноженному на плотность ❷. Используя значение радиуса, вы шкалируете число звезд в размер изображаемого диска. Затем вызовите функцию `random_polar_coordinates()` для получения пары (x, y) ❸.

Завершите создание изображаемого на холсте объекта с помощью пары (x, y) ❹. Поскольку вы уже использовали наименьший размер овала для звезд вдоль и вокруг спирали, то вместо `create_oval()` примените метод `create_text()`, с помощью которого можно для представления звезды использовать точку. Параметр размера шрифта позволит вам масштабировать звезды звездной дымки до тех пор, пока вы не найдете нечто эстетически приемлемое.

На рис. 10.13 представлено сравнение галактической модели без звездной дымки (слева) и со звездной дымкой (справа).

С дымкой можно поступить поизобретательнее. Например, вы можете сделать звезды более многочисленными и окрасить их в серый цвет или же применить цикл для изменения их размера и цвета. Только не используйте зеленый цвет, т. к. зеленых звезд во Вселенной нет!

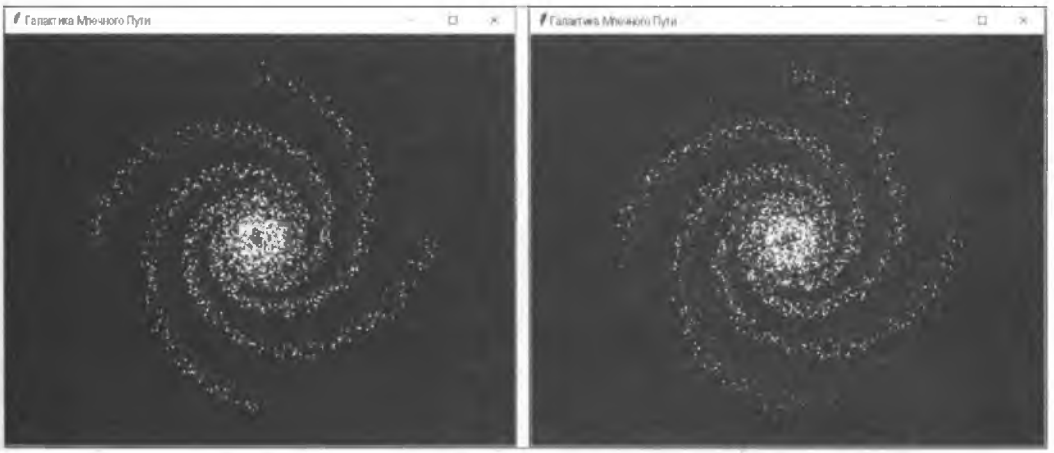


Рис. 10.13. Галактические модели без звездной дымки (слева) и со звездной дымкой (справа)

Определение функции *main()*

В листинге 10.9 определена функция `main()` программы galaxy_simulator.py. Она будет делать вызовы функций шкалирования галактики, вычислять вероятность обнаружения, строить изображение галактики и размещать статистику. Она также будет выполнять основной цикл модуля `tkinter`.

Листинг 10.9. Определяет и вызывает функцию `main()`. Файл galaxy_simulator.py, часть 7

```
def main():
    """Рассчитать вероятность обнаружения и изобразить
       галактику и статистику."""
    ❶ disc_radius_scaled, disc_vol_scaled = scale_galaxy()
       detection_prob = detect_prob(disc_vol_scaled)

       # построить 4 основных спиральных рукава и 4 тянущихся позади рукава
    ❷ spirals(b=-0.3, r=disc_radius_scaled, rot_fac=2, fuz_fac=1.5, arm=0)
       spirals(b=-0.3, r=disc_radius_scaled, rot_fac=1.91, fuz_fac=1.5, arm=1)
       spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=2, fuz_fac=1.5, arm=0)
       spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=-2.09, fuz_fac=1.5,
                                                                    arm=1)
       spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=0.5, fuz_fac=1.5, arm=0)
       spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=0.4, fuz_fac=1.5, arm=1)
       spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=-0.5, fuz_fac=1.5, arm=0)
       spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=-0.6, fuz_fac=1.5, arm=1)
       star_haze(disc_radius_scaled, density=8)

       # изобразить легенду
    ❸ c.create_text(-455, -360, fill='white', anchor='w',
                    text='Один пиксел = {} св. лет'.format(SCALE))
```

```
    c.create_text(-455, -330, fill='white', anchor='w',
                  text='Диаметр радиопузыря = {} св. лет'
                  .format(SCALE))
    c.create_text(-455, -300, fill='white', anchor='w',
              text='Вероятность обнаружения для {:,} цивилизаций = {}'.
                  format(NUM_CIVS, detection_prob))

    # разместить пузырь Земли диаметром 225 св. лет и аннотировать 0
❹   if SCALE == 225:
❺       c.create_rectangle(115, 75, 116, 76, fill='red', outline='')
        c.create_text(118, 72, fill='red', anchor='w',
                      text="<---------- Радиопузырь Земли")

    # запустить цикл tkinter
❻   root.mainloop()

❼ if __name__ == '__main__':
    main()
```

Начните функцию `main()` с вызова функции `scale_galaxy()`, которая получит шкалированный объем диска и радиус ❶. Затем вызовите функцию `detect_prob()` и передайте ей переменную `disc_vol_scaled`. Назначьте результаты переменной `detection_prob`.

Теперь постройте изображение галактики (графическую модель) ❷. Эта процедура предусматривает неоднократный вызов функции `spirals()` с небольшими изменениями при каждом вызове. Параметр `arm` наносит яркие главные рукава и тусклые рукава, тянущиеся позади. Переменная `rot_fac` (коэффициент поворота) определяет место, где наносится спираль. Незначительное изменение коэффициента поворота между рукавами 0 и 1 (например, с `2` до `1.91`) заставляет тусклый рукав немного смещаться от яркого рукава. Завершите сборку изображения, вызвав функцию `star_haze()`. Опять же, не стесняйтесь поэкспериментировать с любым из ее параметров.

Далее изобразите легенду и статистику. Начните со шкалы ❸ и диаметра радиопузыря, за которыми следует вероятность обнаружения для заданного числа цивилизаций. Аргументы включают координаты *x* и *y*, цвет заливки (текста), якорь выравнивания — левая сторона представлена буквой `w` (от англ. west — запад) — и текст. Обратите внимание на использование цепочки символов `{:,}` для вставки пробела в качестве разделителя тысяч. Это часть нового метода форматирования символьных цепочек. Подробнее о данном методе можно прочитать на веб-странице **https://docs.python.org/3/library/string.html#string-formatting**.

Если пользователь выбрал радиопузырь диаметром 225 световых лет ❹, то изображение будет иметь тот же масштаб, что и наш собственный эмиссионный пузырь, поэтому разместите красный пиксел в приблизительном местоположении нашей Солнечной системы и аннотируйте его ❺. Изобразить пиксел с помощью

модуля `tkinter` можно несколькими способами. Здесь используется метод `create_rectangle()`. Помимо него также можно создать линию длиной в один пиксел. Это делается с помощью следующей ниже инструкции:

```
c.create_line(115, 75, 116, 75, fill='red')
```

При использовании метода `create_rectangle()` первыми двумя аргументами являются точки (`x0`, `y0`), соответствующие левому верхнему углу, и (`x1`, `y1`), т. е. расположение пиксела сразу за правым нижним углом. При использовании метода `create_line()` аргументами являются начальная и конечная точки. Ширина линии по умолчанию составляет один пиксел.

Завершите функцию `main()`, выполнив функцию `mainloop()` модуля `tkinter`, также именуемую *событийным циклом* ●. Она оставляет корневое окно открытым до тех пор, пока вы его не закроете.

Вернувшись в глобальное пространство, завершите программу, разрешив ей запускаться автономно либо вызываться как модуль из другой программы ●.

Окончательное изображение будет выглядеть так, как на рис. 10.14, показанном с радиопузырем Земли и центральной черной дырой.

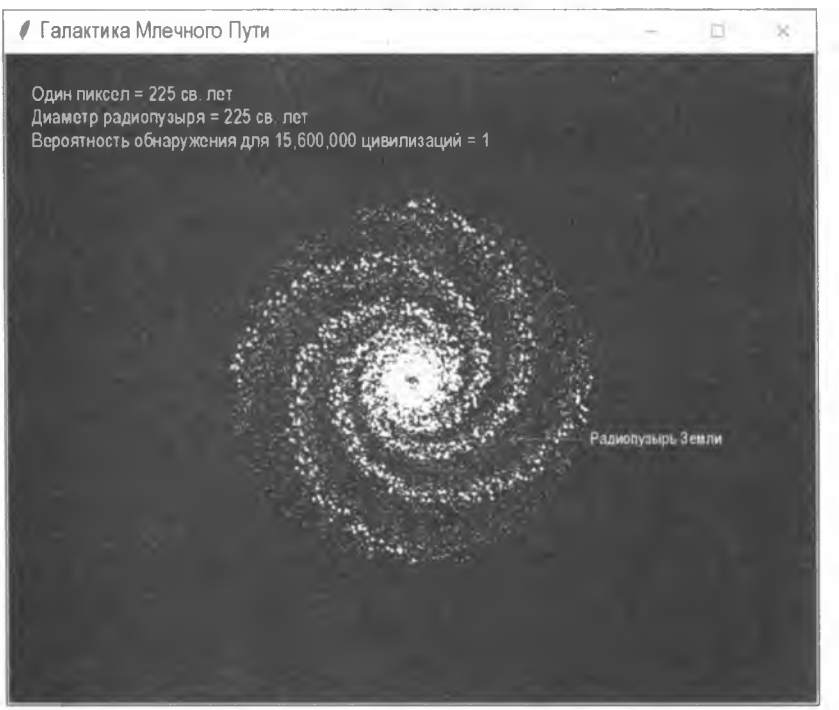


Рис. 10.14. Окончательное изображение с радиопузырем Земли радиусом 225 световых лет, размещенным на галактической карте

Обратите внимание, что, несмотря на то что наш радиопузырь в этой шкале не превышает булавочного ушка, если бы цивилизации имели дальность обнаружения, равную 112,5 световым годам, и если бы таких цивилизаций было столько, сколько

предсказывают текущие верхние параметры уравнения Дрейка, то вероятность обнаружения составила бы 1!

Результаты

Учитывая огромную неопределенность во входных данных и использование упрощающих допущений, вы не отыщете здесь точности. То, что вы отыщете, — это направленность. Следует ли нам (или кому-то вроде нас) ожидать обнаружения другой цивилизации, которая не пытается активно связаться с нами? Судя по рис. 10.15, скорее всего, нет.

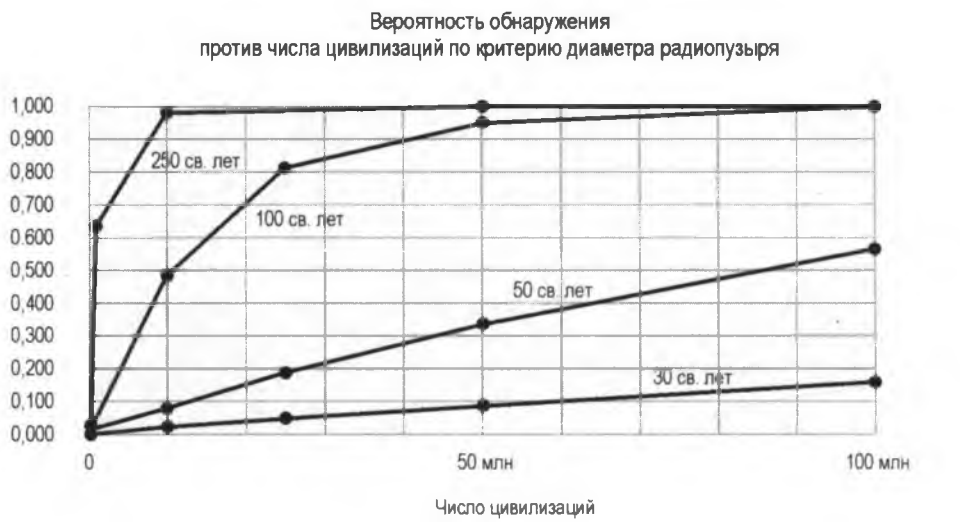


Рис. 10.15. Вероятность того, что одна цивилизация обнаружит другую в условиях различных диаметров радиопузырей и разного числа цивилизаций в галактике

При нашей текущей технологии мы могли бы обнаружить эмиссии от цивилизации на расстоянии до 16 световых лет, что соответствует радиопузырю диаметром 32 световых года. Даже если галактика была бы заполнена 15,6 млн развитых цивилизаций, как предсказано обновленным уравнением Дрейка в статье Википедии, вероятность обнаружения радиопузырей диаметром 32 световых года составляет менее 4%!

Взгляните еще раз на рис. 10.14, и вы сможете оценить всю огромность и пустынность нашей галактики. У астрономов для этого даже есть слово — *ланиакея*, что по-гавайски означает "необъятные небеса".

Земля, как описал ее Карл Саган, — это всего лишь "пылинка, зависшая в солнечном луче". И недавние исследования показывают, что окно возможностей для обнаружения цивилизаций с помощью радиоволн гораздо меньше, чем мы думали. Если другие цивилизации последуют нашему примеру и перейдут на цифровые сигналы и спутниковую связь, то их случайные радиоутечки снизятся по меньшей мере в четыре раза. Мы все становимся непреднамеренно скрытными, расцветаем в течение ста лет или около того, а затем затухаем.

Учитывая эти факты, неудивительно, что правительство больше не финансирует поиск внеземного разума с помощью радиотелескопов. В наши дни усилия переключаются на оптические методы, которые ищут в атмосфере экзопланет сигнатурные газы, такие как отходы жизнедеятельности и промышленной активности.

Резюме

В этой главе вы получили опыт использования модулей библиотек `tkinter`, `matplotlib` и `NumPy`. Вы сгенерировали многочленное выражение для выполнения обоснованных оценок вероятности обнаружения случайных радиопередач инопланетян и использовали всегда доступный модуль `tkinter` с целью добавления в анализ крутой визуальной компоненты.

Дальнейшее чтение

Книга Пола Дэйвиса "Мы — одни? Философские последствия открытия внеземной жизни" (Davies P. Are we alone? Philosophical implications of the discovery of extraterrestrial life. BasicBooks, 1995) предлагает нам вдумчиво взглянуть на поиск инопланетной жизни. Идеи в ней изложены выдающимся ученым и удостоенным наград писателем научно-популярных книг.

В работе Гарри И. Рингермахера и Лоуренса Р. Мида "Новая формула, описывающая структуру каркаса спиральных галактик" (Harry I. Ringermacher H. I., Mead L. R. A new formula describing the scaffold structure of spiral galaxies // Monthly Notices of the Royal Astronomical Society. — 2009. — July 21; **https://arxiv.org/abs/0908.0892v1**) приводятся формулы для моделирования форм спиральных галактик, наблюдаемых телескопом Хаббла.

Учебное пособие Джона Ую Шипмэна "Справочник по Tkinter 8.5: GUI для Python" (Shipman J. W. Tkinter 8.5 Reference: GUI for Python. New Mexico Tech Computer Center, 2013) является полезным дополнением к официальной документации по `tkinter`. Его можно найти по адресу **http://infohost.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf**.

Еще один полезный интернет-ресурс по `tkinter` можно найти по адресу **https://wiki.python.org/moin/TkInter/**.

В книге Баскара Чодхари "Горячее пособие по разработке приложений с GUI на основе tkinter" (Chaudhary B. Tkinter GUI application development HOTSHOT. Packt Publishing, 2013) для подачи материала по `tkinter` используется проектный подход.

Практические проекты

Попробуйте три приведенных ниже дополнительных проекта. Их можно найти в приложении к книге либо скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**.

Далекая-предалекая галактика

Надоело жить в галактике Млечный Путь? Черт возьми, а кому нет? К счастью, на небе и на Земле есть нечто большее, чем просто логарифмические спирали. Примените Python и `tkinter`, для того чтобы построить нам новый дом, но не обязательно реалистичный. Для вдохновения посетите онлайновые статьи, такие как пост Александра Деверта (Alexandre Devert) в его блоге Marmakoide под названием "Распределение точек на диске и на сфере" (Spreading Points on a Disc and on a Sphere, **http://blog.marmakoide.org/**). Показанный на рис. 10.16 пример был построен с помощью программы galaxy_practice.py.

Рис. 10.16. Галактика, порожденная программой galaxy_practice.py

Построение галактической империи

Выберите место в галактике, среднюю скорость перемещения от 5 до 10% скорости света и временной шаг в 500 тыс. лет. Затем смоделируйте расширение космической империи. На каждом временном шаге вычисляйте размер расширяющегося пузыря колонизации и обновляйте карту галактики. Проверьте свои результаты, установив местоположение родного мира в центре галактики, задав скорость равной 1 и подтвердив, что для достижения края галактики потребуется 50 тыс. лет.

Когда у вас будет отлаженная программа, вы сможете проводить интересные эксперименты. Например, вы сможете проверить, с какой скоростью мы должны были бы разведывать галактику через 10 млн лет, как было упомянуто во введении к этой главе (рис. 10.17).

Вы также сможете оценить, какую часть галактики могла разведать Федерация "Звездный путь" (Star Trek) в свои первые 100 лет, исходя из того, что в среднем их скорость составляла 100-кратную скорость света при кривизне 4 (рис. 10.18).

Эти рисунки были построены с помощью программы empire_practice.py.

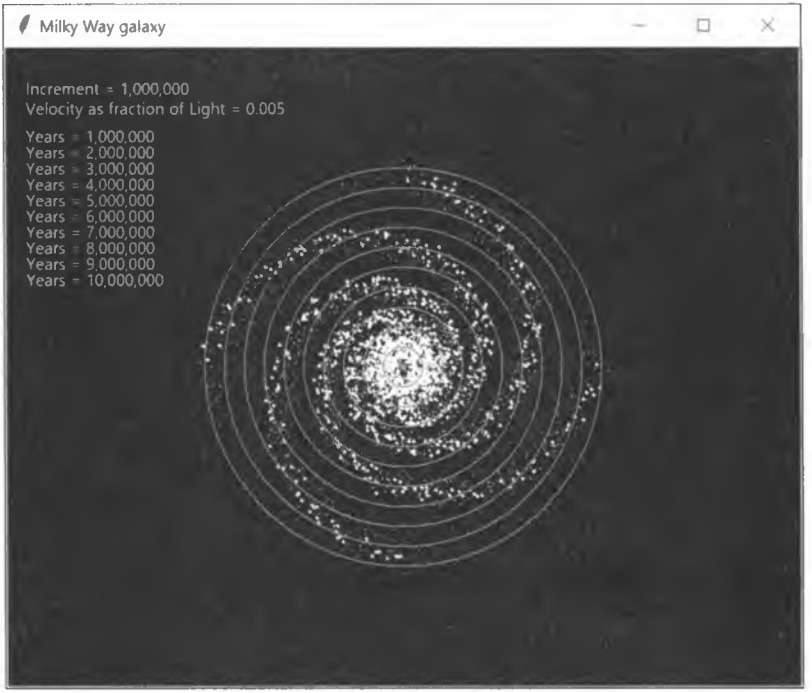


Рис. 10.17. Расширение империи, расположенной в ядре, с использованием перемещения ниже скорости света в течение 10 млн лет

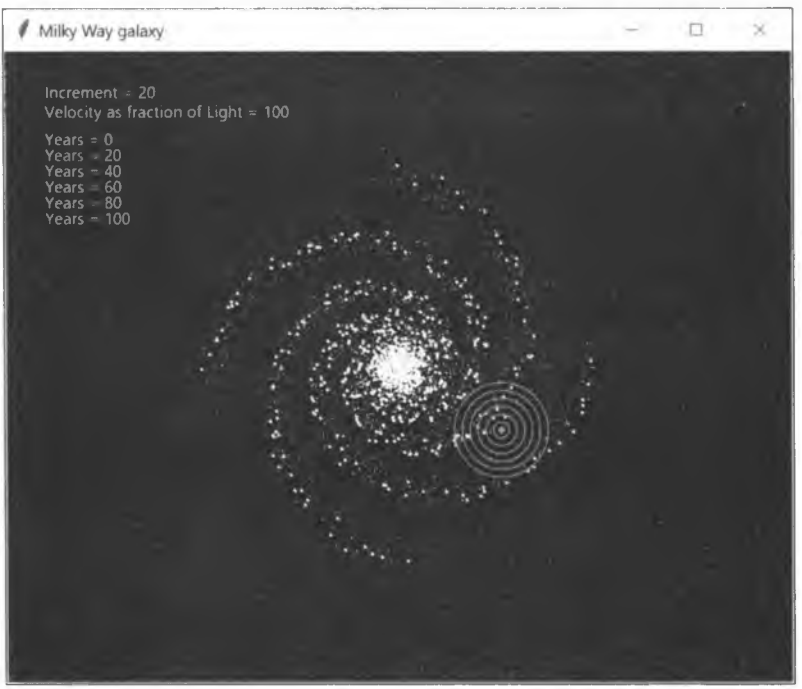


Рис. 10.18. Расширение Федерации из к/ф "Звездный путь" в первые 100 лет под воздействием коэффициента кривизны 4

Окольный путь предсказания обнаруживаемости

Еще один способ предсказать вероятность обнаружения заключается в использовании для распределения цивилизаций полярных координат — в виде точек *xyz* — по галактическому диску, а затем округлении точек до радиуса ближайшего радиопузыря. Точки, расположенные в одном месте, представляют цивилизации, которые могут обнаружить друг друга. Но будьте осторожны — этот метод округляет с использованием кубов, а не сфер, поэтому вам нужно будет преобразовать радиус в сторону куба, который производит тот же объем.

Напишите программу таким образом, чтобы она предсказывала вероятность обнаружения пузырей с радиусом 16 световых лет (предел нашей текущей технологии) с учетом 15,6 млн передающих цивилизаций, случайно распределенных по всей галактике (результата обновленного уравнения Дрейка, взятого из Википедии). При распределении цивилизаций используйте полный радиус размером 50 тыс. световых лет и высоту 1000 световых лет галактической модели.

Для решения этой задачи обратитесь к программе rounded_detection_practice.py. Учтите, что выполнение указанной программы займет несколько минут.

Сложные проекты

Вот несколько последующих проектов, которые вы можете попробовать самостоятельно. Помните, что решения сложных проектов не предоставляются.

Создание спиральной галактики с перемычкой

Наше понимание Млечного Пути развивается по мере получения и анализа новых астрономических данных. Ученые теперь считают, что ядро галактики имеет удлиненную брусковидную форму. Используйте уравнения, приведенные в статье Рингермахера и Мида, процитированной в *разд. "Дальнейшее чтение" ранее в этой главе*, для создания новой визуальной модели галактики на основе `tkinter`, которая соблюдает концепцию спирали с брусковидной перемычкой.

Добавление в галактику обитаемых зон

Солнечные системы имеют зоны Златовласки, которые благоприятны для развития жизни. Планеты, вращающиеся в этих зонах, остаются достаточно теплыми, что позволяет по крайней мере части их воды оставаться в жидком состоянии.

Существует также теория, что галактики, как и солнечные системы, имеют обитаемые зоны, в которых жизнь будет развиваться с большей вероятностью. Одно из определений обитаемой зоны для Млечного Пути устанавливает ее внутреннюю границу примерно в 13 тыс. световых лет от центра галактики и ее внешнюю границу примерно в 33 тыс. световых лет от ее центра (рис. 10.19). Ядро исключено ввиду высоких уровней излучения, большого числа сверхновых и нарушающих орбиту сложных гравитационных полей, возникающих по причине того, что все

звезды расположены близко. Области обода обречены из-за низкой металличности, которая имеет решающее значение для развития планет.

Уточнение модели обитаемой зоны исключает спиральные рукава по причинам, аналогичным тем, которые применяются к ядру. Наше собственное существование этому не противоречит. Земля расположена в Орионском "отроге", относительно небольшом участке между рукавами Стрельца и Персея.

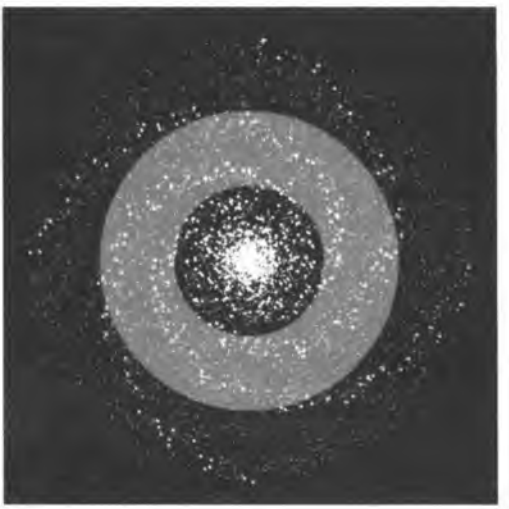


Рис. 10.19. Примерная обитаемая зона галактики (отмечена серым цветом), наложенная на модель Млечного Пути

Отредактируйте программу galaxy_simulation.py так, чтобы она использовала объем только в галактической обитаемой зоне, каким бы образом вы ее ни определили. Вы должны исследовать, какими могут быть эти объемы и какое влияние они окажут на число цивилизаций (N), которое вычисляется уравнением Дрейка. Рассмотрите возможность использования областей, таких как ядро, спирали, внешний обод и т. д., в которых N отличается, но цивилизации по-прежнему распределены случайным образом. Выделите эти области на галактической карте и напечатайте оценочные значения вероятности их обнаружения.

11

ЗАДАЧА МОНТИ ХОЛЛА

Ведущий телевизионного игрового шоу "Заключим сделку" (Let's Make a Deal) Монти Холл (Monty Hall) обычно показывал конкурсантам три закрытые двери и просил их выбрать одну из них. За одной дверью находился ценный приз, за двумя другими — дурно пахнущие старые козы. После того как конкурсант выбирал дверь, Монти открывал одну из оставшихся дверей, показывая козу. Затем конкурсанту предоставлялся окончательный выбор: поменять двери или остаться со своим первоначальным выбором.

В 1990 г. Мэрилин вос Савант (Marilyn vos Savant), "самая умная женщина в мире", в своей еженедельной колонке журнала Parade "Спросите у Мэрилин" ("Ask Marilyn") заявила, что конкурсант должен выбирать смену дверей. Хотя ее ответ был правильным, он вызвал огненную бурю писем ненависти, предвзятых гендерных суждений и академических преследований. По ходу разбирательства многие профессора математики показали себя не в лучшем свете, но у этого неприятного дела была и светлая сторона. Бурное обсуждение бросило общественность в объятия науки статистики, а упражнение, которое предложила вос Савант, обрело свое место в тысячах учебных занятий. Ее ручные тесты — позже продублированные на компьютерах — подтвердили ее высмеянную всеми "женскую логику".

В этой главе вы будете использовать *симуляцию Монте-Карло* — метод моделирования вероятности разных результатов из диапазона случайных входных значений с целью проверки правоты вос Савант. После этого вы воспользуетесь модулем `tkinter` для создания забавного графического интерфейса, который обратится к ее просьбе к школьникам помочь ей с ее экспериментом.

Симуляция Монте-Карло

Представьте: вы хотите узнать, какова вероятность получения другой грани с каждым броском при подбрасывании кубика шесть раз подряд. Если вы — математический гений, то, вероятно, вычислите это, используя детерминированное выражение $6!/6^6$, или

$$\frac{6}{6}\times\frac{5}{6}\times\frac{4}{6}\times\frac{3}{6}\times\frac{2}{6}\times\frac{1}{6},$$

получив 0,015. Если вы не особо одарены математически, то получить тот же ответ вы сможете с помощью языка Python и большого числа бросаний:

```
>>> from random import randint
>>> trials = 100000
>>> success = 0
>>> for trial in range(trials):
        faces = set()
        for rolls in range(6):
            roll = randint(1, 6)
            faces.add(roll)
        if len(faces) == 6:
            success += 1
>>> print("вероятность успеха = {}".format(success/trials))
вероятность успеха = 0.01528
```

В этом примере используются цикл `for` и функция `randint`, которая шесть раз подряд случайно выбирает число от 1 до 6, представляющее одну из граней кубика. Каждый результат добавляется в множество граней с именем `faces`, которое не допускает дубликатов. Длина множества достигнет 6 в единственном случае: когда каждый бросок дает уникальное число, т. е. равняется успешному случаю. Внешний цикл `for` выполняет испытание с шестью бросаниями кубика 100 тыс. раз подряд. Деление числа успехов на число испытаний дает ту же самую вероятность 0,015, что и детерминированное выражение.

Симуляция Монте-Карло использует *многократный случайный отбор* экземпляров — в этом случае каждое бросание кубика является случайной выборкой экземпляра — с целью предсказания разных результатов в заданном диапазоне условий. В данном примере диапазоном условий были один шестигранный кубик, шесть бросаний без повторов в расчете на одно испытание и 100 тыс. испытаний.

Конечно же, симуляция Монте-Карло обычно применяется в более сложных задачах — с бóльшим числом переменных и широким диапазоном неопределенности, где результаты нелегко предсказать.

Существует несколько типов симуляции Монте-Карло, но большинство приложений выполняют следующие базовые действия:

- перечислить входные переменные;
- обеспечить распределение вероятностей для каждой переменной;

- начать цикл:
  - случайно выбрать значение из распределений для каждого входа;
  - использовать значения в детерминированном вычислении, которое всегда будет производить одни и те же выходы из одних и тех же входов;
  - повторить указанное число раз;
- агрегировать результаты и сгенерировать статистику, такую как средний результат вычисления.

На примере бросания кубика эти шаги были бы следующими:

- входные переменные — результаты шести бросаний кубика;
- распределение вероятности для броска — равномерное (1/6 для каждой грани);
- цикл:
  - случайно отобранное значение — бросание кубика (значение, взятое из распределения);
  - вычисление — добавить шесть значений в множество, и если длина множества равна 6, то добавить 1 в переменную успешности `success`;
  - повтор — 100 000 раз;
- агрегирование: разделить переменную успешности `success` на 100 000 для получения вероятности.

Нассим Талеб (Nassim Taleb), признанный критиками автор книг "Черный лебедь" (The Black Swan) и "Одураченные случайностью" (Fooled by Randomness), является поклонником симуляции Монте-Карло. Он утверждает, что наш мозг устроен так, чтобы быстро вытаскивать нас из неприятностей, а вовсе не решать сложные задачи неопределенности или вероятности. Мы не созданы для сильно асимметричных распределений и нелинейностей, но мозг некоторых людей внутренне более способен понимать риски, используя симуляцию Монте-Карло, чем другие методы. В реальной жизни мы не наблюдаем распределения вероятностей, мы просто наблюдаем события.

Каждый прогон симуляции Монте-Карло представляет одно событие, например исчерпались или нет ваши пенсионные накопления. Для многих из нас указанная симуляция делает риск реальным. Она помогает нам понять, насколько плохим или хорошим может все оказаться, т. е. то, что мы не всегда можем извлечь из математических абстракций. Вооруженные сущностным пониманием, полученным благодаря симуляции Монте-Карло, мы можем подготовиться и к обороне от неприятностей, и к эксплуатации преимуществ.

В поддержку математики, лежащей в основе задачи Монти Холла, вы разработаете приложение симуляции Монте-Карло, как в предыдущем примере бросания кубика. Затем, в *главе 12*, вы задействуете симуляцию Монте-Карло для построения симулятора пенсионных накоплений с целью планирования вашей (или ваших родителей) обеспеченной жизни на пенсии.

Проект 18: верификация утверждения вос Савант

Для подтверждения правоты вос Савант мы применим подход Монте-Карло и просимулируем десятки тысяч "игр" для того, чтобы увидеть, как все сложится. Эта программа будет минималистской, т. к. цель состоит в простом подтверждении без какого-либо украшательства.

Цель

Написать простую программу на языке Python, в которой симуляция Монте-Карло используется с целью определения вероятности выигрыша в задаче Монти Холла путем изменения первоначального варианта выбора.

Стратегия

Правильный ответ к задаче Монти Холла — поменять двери после того, как Монти покажет козу. По статистике, это удвоит ваши шансы на победу!

Посмотрите на рис. 11.1. В начале игры все двери закрыты, и шансы на то, что за данной дверью скрывается приз, равны 1 к 3. Конкурсант может выбрать только одну дверь, имея в виду, что шансы на получение приза за одной из двух других дверей равны 2 к 3. После того как будет показана коза, шансы остаются 2 к 3, но они возвращаются к оставшейся двери. Помните, Монти знает, где спрятан приз, и никогда *эту* дверь не покажет. Поэтому вероятность успеха составляет 1/3 в случае, если вы останетесь при своем первом варианте выбора против 2/3 в случае его изменения.

Рис. 11.1. Шансы на победу в задаче Монти Холла до и после раскрытия двери, за которой скрывалась коза

Если вы сомневаетесь в математической формуле, то можете применить симуляцию Монте-Карло, которая предоставит подтверждающие доказательства, точно так же, как мы сделали с примером бросания кубика. Просто нужно выбрать выиг-

рышную дверь наугад, выбрать вариант конкурсанта наугад и зарегистрировать результат, когда оба этих значения совпадают. Если повторить эту процедуру тысячи раз, то результат сойдется на детерминированном математическом решении.

Код верификации утверждения Мэрилин вос Савант

Описанная в этом разделе программа monty_hall_mcs.py автоматизирует процесс выбора дверей и регистрации результатов, благодаря которой вы сможете выполнять тысячи испытаний и оценивать их менее чем за секунду. Код программы можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython.com/**.

Получение входного числа прогонов

Листинг 11.1 начинает программу monty_hall_mcs.py с того, что просит пользователя указать, сколько прогонов — или игр — он хочет симулировать. Вы также предоставите пользователю значение по умолчанию. Это отличный способ направить пользователя к разумному первому отклику, а также сэкономить на нескольких нажатиях клавиш.

Листинг 11.1. Импортирует модули и определяет функцию `user_prompt()`. Файл monty_hall_mcs.py, часть 1

```
❶ import random
❷ def user_prompt(prompt, default=None):
      """Позволить использование значения по умолчанию при вводе."""
    ❸ prompt = '{} [{}]: '.format(prompt, default)
    ❹ response = input(prompt)
    ❺ if not response and default:
          return default
      else:
          return response

  # ввести число раз, которое симуляция будет выполняться
❻ num_runs = int(user_prompt("Введите число прогонов", "20000"))
```

Начните с импорта модуля `random`, необходимого для выполнения симуляции Монте-Карло ❶. Затем определите функцию с именем `user_prompt()`, которая просит пользователя ввести число выполняемых игр либо принимает значение по умолчанию, если оно указано ❷. Данная функция принимает два аргумента: первый — это текстовая подсказка, которая сообщает пользователю о том, что ему нужно сделать; второй — значение по умолчанию, которое с самого начала равно `None`. Сразу же переопределите переменную `prompt`, для того чтобы она выводилась на экран со значением по умолчанию в скобках, как предусмотрено правилами ❸. Назначьте введенное пользователем число переменной `response` ❹. Если пользователь нажимает клавишу <Enter> без ввода числа и значение по умолчанию существует, то

функция `user_prompt()` возвращает значение по умолчанию ❺. В противном случае указанная функция возвращает введенное пользователем число. С помощью этой функции можно определить число прогонов, назначив возвращаемое значение переменной `num_runs` ❻. Каждый прогон будет представлять конкурсанта, играющего в игру один раз.

Выполнение симуляции Монте-Карло и вывод результатов на экран

Листинг 11.2 случайно подбирает выигрышную дверь и первый выбранный пользователем вариант, затем агрегирует и показывает статистику. Интересно, что второй выбранный пользователем вариант — поменять двери или выбрать ту же дверь — для получения правильного ответа не требуется. Если первоначальный вариант выбора является выигрышной дверью, то правильный ответ — не менять двери. Точно так же, если первоначальный вариант выбора и выигрышная дверь различаются, то правильный ответ — поменять двери. Нет никаких причин моделировать то, что конкурсант может сделать или не сделать.

Листинг 11.2. Выполняет симуляцию Монте-Карло и выводит результаты на экран. Файл monty_hall_mcs.py, часть 2

```
    # задать счетчики количеств для способов выигрыша
❶   first_choice_wins = 0
    pick_change_wins = 0
❷   doors = ['a', 'b', 'c']

    # выполнить Монте-Карло
❸   for i in range(num_runs):
        winner = random.choice(doors)
        pick = random.choice(doors)

❹       if pick == winner:
            first_choice_wins += 1
        else:
            pick_change_wins += 1

❺   print("Выигрыши с исходным вариантом выбора = {}".
          format(first_choice_wins))
    print("Выигрыши с измененным вариантом выбора = {}".
          format(pick_change_wins))
    print("Вероятность выигрыша с первого угадывания: {:.2f}"
          .format(first_choice_wins / num_runs))
    print("Вероятность выигрыша при изменении варианта: {:.2f}"
          .format(pick_change_wins / num_runs))

❻   input("\nНажмите клавишу Enter для выхода из программы.")
```

Задайте две переменные для отслеживания того, что является выигрышным результатом: замена двери либо пребывание в том же состоянии ❶. Затем создайте список, представляющий три двери ❸.

Симуляция Монте-Карло начинается с цикла `for`, который перебирает число прогонов ❸. Внутри этого цикла выберите выигрышную дверь и первый вариант, выбранный пользователем из списка дверей, используя функцию `random.choice()`, и назначьте их переменным.

Поскольку мы имеем дело с двоичной системой — пользователь меняет дверь или остается при своем первоначальном выборе — вам понадобится условное выражение, которое увеличивает счетчики только на основе отношения переменной `pick` к переменной `winning` ❹.

Завершите программу, представив окончательные результаты. Покажите фактические значения, а также вычисленные вероятности ❺. Сообщите пользователю о том, что программа завершила работу ❸.

Ниже приведен пример результатов для 20 тыс. прогонов по умолчанию:

```
Введите число прогонов [20000]:
Выигрыши с исходным вариантом выбора = 6628
Выигрыши с измененным вариантом выбора = 13372
Вероятность выигрыша с первого угадывания: 0.33
Вероятность выигрыша при изменении варианта: 0.67

Нажмите клавишу Enter для выхода из программы.
```

Некоторых людей компьютерная распечатка не впечатляет. Им нужно что-то поубедительнее, поэтому в следующем проекте вы переупакуете свой код в более практичный формат — укомплектованный дверьми, призами и козами. Он также будет ответом призыву Мэрилин вос Савант к школьникам присоединиться и помочь восстановить ее честь.

Проект 19: игра в Монти Холла

Трехдверная игра, используемая в задаче Монти Холла, для вас проста настолько, что вы можете ее построить с помощью модуля `tkinter`. Вы начали работать с графикой `tkinter` в *главе 10*. Теперь вы обопретесь на эти знания, добавив нажимаемые пользователем интерактивные кнопки.

Цель

Просимулировать задачу Монти Холла с использованием графического интерфейса, построенного с помощью модуля `tkinter`. Отслеживать то, какая линия поведения приводит к победе: смена двери или бездействие. Вдобавок обновлять и показывать эту статистику по мере прохождения игры.

Краткое введение в объектно-ориентированное программирование

Модуль `tkinter` был написан с использованием *объектно-ориентированного программирования* (ООП). ООП — это языковая модель, построенная вокруг структур данных, именуемых *объектами*, состоящих из *данных*, *методов* и взаимодействий между ними — в отличие от *действий* и *логики*, используемых в процедурном программировании. Объекты создаются из *классов*, которые для объектов подобны схематическим макетам.

ООП — абстрактное понятие, и его легче оценить по достоинству, когда вы пишете большие, сложные программы. Оно уменьшает дублирование кода и упрощает его обновление, сопровождение и повторное использование. По этой причине большинство коммерческих программ теперь создают с использованием ООП.

Если бы вы реализовывали ООП в мелких программах, как те, которые мы писали до сих пор, то большинство из них выглядели бы переусложненными. По сути дела, одна из моих самых любимых цитат, приписываемых британскому компьютерщику Джо Армстронгу, касается как раз этого аспекта ООП: "беда с объектно-ориентированными языками заключается в том, что они несут с собой всю эту неявную среду. Вы хотели получить банан, а получили гориллу с бананом и всеми джунглаям в придачу!"

Несмотря на это, объекты, производимые ООП, оказывают очень хорошую поддержку графических интерфейсов и игр, даже в некоторых небольших проектах. Давайте рассмотрим пример использования настольной игры типа "Подземелья и Драконы", в которой игроки могут быть разными персонажами — гномами, эльфами и волшебниками. В этих играх используются карточки с персонажами, на которых перечисляются важные сведения о каждом типе персонажа. Если ваша игровая фигура будет представлять гнома, то она унаследует характеристики, описанные на карточке (рис. 11.2).

Листинги 11.3 и 11.4 воспроизводят игру в стиле настольной игры, позволяя вам создавать виртуальные карточки для гнома и эльфа, называть свои фигуры и устраивать для них сражение. Исход сражения будет влиять на одну из *жизненных сил персонажа*, которые представляют его здоровье. Обязательно обратите внимание на то, как ООП позволяет вам легко создавать множество идентичных объектов — в данном случае гномов или эльфов — путем их "штамповки" из предопределенного шаблона, именуемого *классом*.

Листинг 11.3. Импортирует модуль `random`, создает класс `Dwarf` и экземпляр объекта `dwarf`

```
❶ >>> import random
❷ >>> class Dwarf(object):
        ❸ def __init__(self, name):
            ❹ self.name = name
              self.attack = 3
```

```
            self.defend = 4
            self.body = 5
        ❺ def talk(self):
            print("Я - человек-клинок, не подходи, а то зарежу!!!")
❻ >>> lenn = Dwarf("Лэнн")
>>> print("Имя гнома = {}".format(lenn.name))
Има гнома = Лэнн
>>> print("Атакующая мощь Лэнна = {}".format(lenn.attack))
Атакующая мощь Лэнна = 3
>>>
❼ >>> lenn.talk()
Я - человек-клинок, не подходи, а то зарежу!!!
```

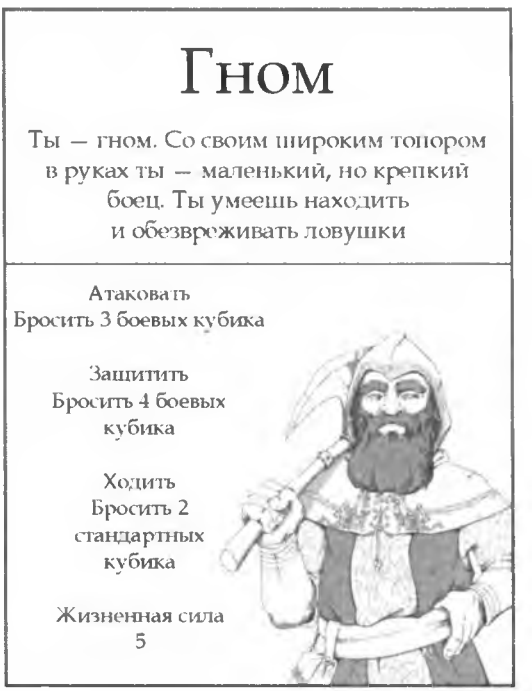


Рис. 11.2. Карточка с персонажем "Гном" из ролевой настольной игры

Начните с импорта модуля `random` для симулирования бросания кубика ❶; это поведение вашего персонажа во время сражения. Теперь определите для гнома его класс, написав с заглавной буквы имя класса, и передайте ему аргумент `object`, который будет именем вашего гнома ❷. Класс — это шаблон для создания объектов определенного типа. Например, когда вы создаете список или словарь, вы создаете их из класса.

Определение класса гномов `Dwarf` выглядит как карточка на рис. 11.2; это, в сущности, генетический макет гнома. Он будет задавать *атрибуты*, такие как мощь и жизненная сила, и *методы*, т. е. то, как персонаж двигается или говорит. Атрибу-

ты — это переменные, *привязанные* к экземпляру класса, а методы — это атрибуты, которые одновременно являются функциями, которым при выполнении передается ссылка на их экземпляр. Класс — это тип данных, и при создании объекта этого типа данных он также называется экземпляром этого класса. Процесс задания начальных значений и поведения экземпляра называется созданием экземпляра, или *инстанцированием*.

Затем определите *метод-конструктор*, также именуемый методом *инициализации*. Он устанавливает начальные значения атрибутов вашего объекта ❼. Метод `__init__()` — это специальный встроенный метод, который Python вызывает автоматически во время создания нового объекта. В этом случае вы передадите два аргумента: ключевое слово `self` и имя вашего объекта.

Параметр `self` является ссылкой на создаваемый экземпляр этого класса, или ссылкой на экземпляр, на котором был вызван метод, технически именуемый экземпляром *контекста*. Если вы создадите нового гнома и назовете его "Стив", то за кулисами `self` станет Стивом. Например, `self.attack` становится "атакой Стива". Если же вы создадите другого гнома по имени Сью, то `self` этого объекта станет "Сью". Благодаря этому область действия атрибута здоровья Стива отделена от здоровья Сью.

Далее под определением конструктора перечислите несколько атрибутов гнома ❹. Вам понадобится имя, для того чтобы можно было отличать одного гнома от другого, а также значение ключевых боевых характеристик. Обратите внимание, как этот список напоминает карточку на рис. 11.2.

Определите метод `talk()` и передайте ему параметр `self` ❺. Передавая ему параметр `self`, вы связываете метод с объектом. В более сложных играх методы могут включать в себя поведение, такое как движение и способность обезвреживать ловушки.

После завершения определения класса создайте экземпляр класса `Dwarf` и назначьте этот объект локальной переменной `lenn`, имени гнома ❻. Теперь напечатайте имя и атрибуты атаки, для того чтобы продемонстрировать, что у вас есть к ним доступ. Завершите вызовом метода `talk()` ❼. Он должен показать на экране сообщение.

Листинг 11.4 создает персонаж эльфа, используя аналогиченые действия, которые вы применяли в листинге 11.3, и пускает его в бой с гномом. Атрибут жизненной силы `body` у эльфа обновляется, отражая исход битвы.

Листинг 11.4. Создает класс эльфов `Elf`, инстанцирует объект `elf`, симулирует битву и обновляет атрибут объекта

```
❶ >>> class Elf(object):
          def __init__(self, name):
              self.name = name
              self.attack = 4
              self.defend = 4
              self.body = 4
```

```
>>> esseden = Elf("Эсседен")
>>> print("Имя эльфа = {}".format(esseden.name))
Имя эльфа = Эсседен
>>> print("Жизненная сила Эсседена = {}".format(esseden.body))
Жизненная сила Эсседена = 4
>>>
❷ >>> lenn_attack_roll = random.randrange(1, lenn.attack + 1)
>>> print("Нападающий бросок Лэнна = {}".format(lenn_attack_roll))
Нападающий бросок Лэнна = 3
❸ >>> esseden_defend_roll = random.randrange(1, esseden.defend + 1)
>>> print("Защитный бросок Эсседена = {}".format(esseden_defend_roll))
Защитный бросок Эсседена = 1
>>>
❹ >>> damage = lenn_attack_roll - esseden_defend_roll
>>> if damage > 0:
    esseden.body -= damage
❺ >>> print("Жизненная сила Эсседена = {}".format(esseden.body))
Жизненная сила Эсседена = 2
```

Определите класс эльфов `Elf` и предоставьте несколько атрибутов ❶. Сделайте их немного отличающимися от гномов и хорошо сбалансированными, как у эльфа. Инстанцируйте эльфа по имени Эсседен и обратитесь к его атрибутам `name` и `body` с помощью инструкции `print`.

Дайте вашим двум персонажам повзаимодействовать между собой, используя бросок виртуального кубика с максимальным значением, равным значению атаки или защиты персонажа. Примените модуль `random` для выбора значения броска в диапазоне от 1 до атрибута атаки Лэнна плюс 1 ❷, затем повторите этот процесс, для получения защиты Эсседена ❸. Вычислите урон Эсседену, вычтя значение броска Эсседена из значения броска Лэнна ❹, и если значение урона является положительным числом, то вычтите его из атрибута жизненной силы Эсседена. Примените `print()` для подтверждения текущего состояния здоровья эльфа ❺.

Как можно себе представить, создание многих подобных персонажей и отслеживание их изменяющихся атрибутов с использованием процедурного программирования может быстро усложниться. ООП обеспечивает вашей программе модульную структуру, позволяет легко скрывать сложность и принадлежность области действия с помощью инкапсуляции, позволяет решать задачи небольшими фрагментами кода и создает совместные шаблоны, которые можно модифицировать и использовать в любом месте.

Стратегия и псевдокод

Теперь вернемся к нашей трехдверной игре. Правила игры формируют львиную долю псевдокода программы:

```
Инициализировать окно игры и показать закрытые двери и инструкции
Выбрать выигрышную дверь наугад
```

```
Получить от игрока вариант выбора двери
Открыть дверь, которая не является выигрышной либо выбранным игроком вариантом
Получить выбранный игроком вариант со сменой или без смены двери
Если игрок поменял дверь:
    Открыть новую дверь
    Если это победная дверь:
        Зарегистрировать как выигрыш за смену двери
    В противном случае:
        Зарегистрировать как выигрыш за бездействие
В противном случае если игрок бездействует:
    Открыть выбранную дверь
    Если это победная дверь:
        Зарегистрировать как выигрыш за бездействие
    В противном случае:
        Зарегистрировать как выигрыш за смену двери
Показать число выигрышей в каждой стратегии в игровом окне
Обнулить игру и закрыть все двери
```

Полезно начинать разработку игры с набросков того, как должно выглядеть игровое окно, дополненное инструкциями, сообщениями и типами кнопок. Сомневаюсь, что вы хотите видеть мои грубые каракули, поэтому вместо них взгляните на рис. 11.3.

Именно так будет выглядеть состояние игры после первого раунда, причем статистика побед будет видна в правом нижнем углу. Обратите внимание, что переключатели для смены дверей не активны до тех пор, пока не будет сделан первоначальный выбор.

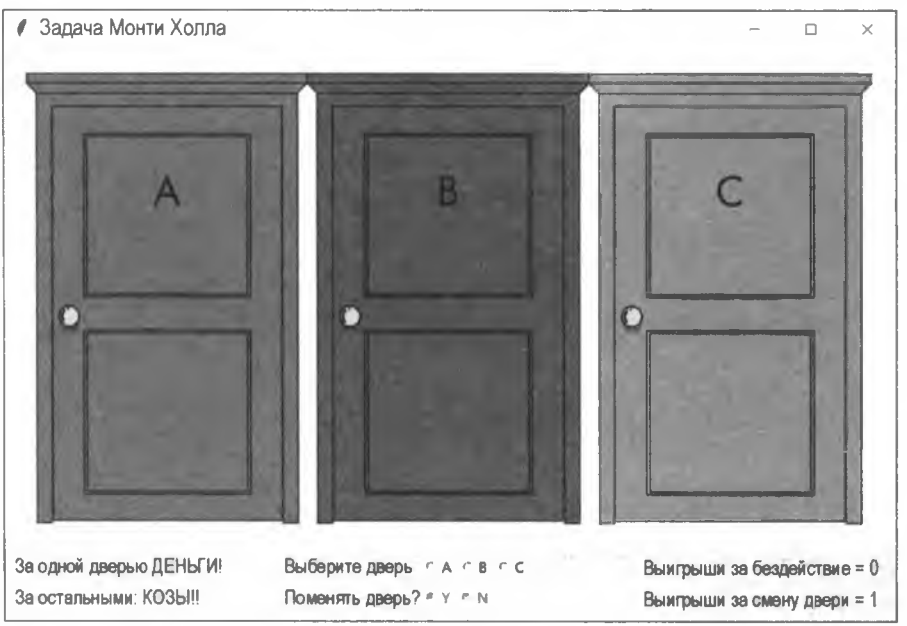


Рис. 11.3. Вид игрового окна после первого раунда игры

Игровые активы

Игровые активы (игровой ассет) — это причудливый термин для всего того, что вам понадобится для создания игры. Они будут состоять из серии изображений, представляющих двери, коз и приз (рис. 11.4).

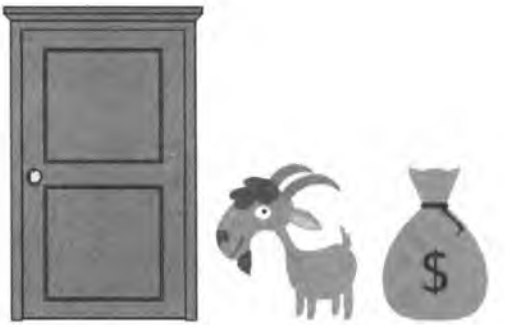


Рис. 11.4. Изображения строительных блоков для программы monty_hall_gui.py

Для объединения 3 базовых изображений в 10 изображений, которые представляют все возможные состояния игры (рис. 11.5), я использовал Microsoft PowerPoint. Это было проектным решением; с помощью дополнительных строк кода я мог бы получить те же результаты, используя только базовые изображения.

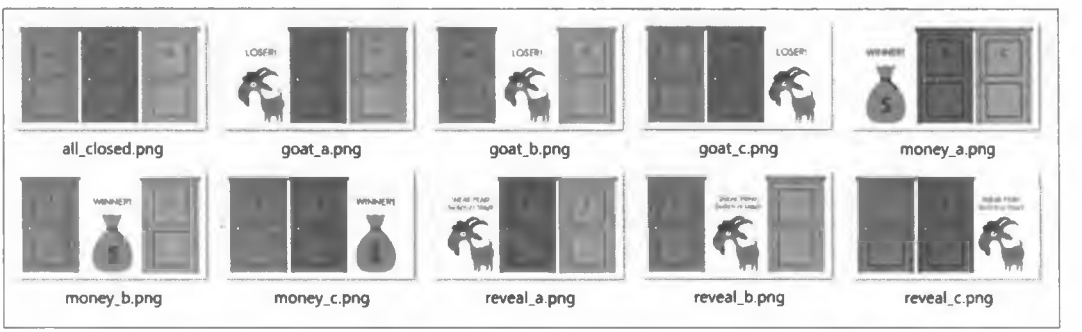


Рис. 11.5. Составные изображения для программы monty_hall_gui.py

Код игры в Монти Холла

Описанная в этом разделе программа monty_hall_gui.py превращает задачу Монти Холла в веселую и познавательную игру. Вам также понадобятся 10 игровых активов, показанных на рис. 11.5. Скачайте их с веб-сайта **https://www.nostarch.com/impracticalpython/** и сохраните все файлы в одной папке.

Импорт модулей и определение класса игры

Листинг 11.5 импортирует модули и определяет класс игры `Game` и метод инициализации `__init__()`.

Листинг 11.5. Импортирует модули и определяет класс `Game` и метод `__init__()`. Файл monty_hall_gui.py, часть 1

```
❶ import random
  import tkinter as tk
❷ class Game(tk.Frame):
      """GUI-приложение для игры с задачей Монти Холла."""

    ❸ doors = ('a', 'b', 'c')

    ❹ def __init__(self, parent):
          """Инициализировать рамку."""
        ❺ super(Game, self).__init__(parent)  # родитель будет
                                              # корневым окном
        ❻ self.parent = parent
          self.img_file = 'all_closed.png'  # текущее изображение дверей
          self.choice = ''                  # вариант двери, выбранный игроком
          self.winner = ''                  # выигрышная дверь
          self.reveal = ''                  # раскрытая дверь с козой
        ❼ self.first_choice_wins = 0        # счетчик для статистики
          self.pick_change_wins = 0         # счетчик для статистики
        ❽ self.create_widgets()
```

Начните с импорта модулей `random` и `tkinter` ❶. Затем определите класс под названием `Game` ❷. *Родителем* этого класса, показанным в скобках, будет класс `Frame` модуля `tkinter`. Это означает, что класс `Game` является *производным* от существующего "базового" класса `Frame` и будет без труда наследовать у него полезные методы. Виджет `Frame` в основном служит в качестве геометрического шаблона для других виджетов, помогая группировать их в многосложные макеты.

Обратите внимание, что классы имеют собственные правила документирования в литералах, которые можно найти по адресу **https://www.python.org/dev/peps/pep-0257**. Как уже говорилось в *главе 1*, приводимые в этой книге литералы документирования для краткости в основном показаны однострочными.

Каждый экземпляр класса `Game` будет использовать те же три двери, а раз так, то для этого можно использовать атрибут класса ❸. Любая переменная, которой присваивается значение вне метода, становится атрибутом класса, подобно тому как в процедурных программах переменные, получающие значения вне функций, становятся глобальными. Вы не хотите, чтобы этот атрибут был случайно изменен, поэтому сделайте его немутируемым с помощью кортежа. Позже вы будете изготавливать списки из этого кортежа всякий раз, когда вы захотите оперировать дверьми.

Теперь, как и в предыдущих примерах с гномом и эльфом, определите метод-инициализатор игрового объекта ❹. Параметр `self` обязателен, но вам также понадобится родитель `parent`, т. е. корневое окно, которое содержит игру.

Базовый класс также можно назвать *суперклассом*, и функция `super()` позволяет вызывать метод суперкласса для получения доступа к унаследованным методам — в данном случае у родительского класса. Сначала передайте `Game` в `super()`, имея в виду, что из `Game` вы хотите вызвать метод суперкласса, которым является класс `Frame` ❺. Затем передайте ему ключевое слово `self` в качестве аргумента для ссылки на вновь инстанцированный объект `Game`.

Часть инструкции с `__init__(parent)` вызывает метод-инициализатор класса `Frame` с родителем `parent` (корневым окном) в качестве аргумента. Теперь атрибуты из предопределенного класса `Frame` модуля `tkinter` могут использоваться вашим объектом `Game`. Обратите внимание, что эту инструкцию можно упростить до `super().__init__()`.

Далее задайте значения серии экземплярных атрибутов ❻. Атрибуты лучше всего инициализировать посредством метода `__init__()`, т. к. это первый метод, который вызывается после создания объекта. Благодаря этому указанные атрибуты будут немедленно доступны для любых других методов в классе. Начните с того, что назначьте родителя, который будет корневым окном `root`, экземпляру. Затем объявите атрибут для хранения одного из файлов изображений (показано на рис. 11.5) и назначьте ему изображение, в котором все двери открыты, — именно это изображение игрок будет видеть в начале каждой игры. Далее объявите атрибуты для выбираемого игроком варианта, выигрышной двери и двери, используемой для раскрытия первой козы.

Используйте счетчик для отслеживания количества выигрышей, достигнутых, если игрок придерживается первого варианта выбора двери, и еще один для регистрации выигрышей, достигнутых, когда игрок меняет дверь ❼. Наконец, вызовите метод, который создаст необходимые для выполнения игры виджеты надписей, кнопок и текста ❽.

Создание виджетов для изображений и инструкций по игре

В листинге 11.6 определяется первая часть метода `create_widgets()`, используемого для построения виджетов надписей, кнопок и текста для игры. Первые два виджета — это надписи `tkinter`, используемые для вывода на экран изображений из рис. 11.5 и предоставления инструкций по игре.

Листинг 11.6. Определяет метод создания виджетов. Файл monty_hall_gui.py, часть 2

```
❶ def create_widgets(self):
      """Создать виджеты надписей, кнопок и текста для игры."""
      # создать надпись с изображением дверей
   ❷ img = tk.PhotoImage(file='all_closed.png')
   ❸ self.photo_lbl = tk.Label(self.parent, image=img,
                               text='', borderwidth=0)
   ❹ self.photo_lbl.grid(row=0, column=0, columnspan=10, sticky='W')
   ❺ self.photo_lbl.image = img
```

```
        # создать надпись с инструкциями по игре
    ❺ instr_input = [
            ('За одной дверью ДЕНЬГИ!', 1, 0, 5, 'W'),
            ('За остальными: КОЗЫ!!!', 2, 0, 5, 'W'),
            ('Выберите дверь:', 1, 3, 1, 'E')
        ]
    ❼ for text, row, column, columnspan, sticky in instr_input:
            instr_lbl = tk.Label(self.parent, text=text)
            instr_lbl.grid(row=row, column=column, columnspan=columnspan,
                           sticky=sticky, ❻ ipadx=30)
```

Определите метод с именем `create_widgets()`, который в качестве аргумента принимает ключевое слово `self` ❶. Затем задайте атрибут для хранения изображения дверей ❷. Обратите внимание, что предварять это имя атрибута ключевым словом `self` не требуется, т. к. оно будет использоваться лишь локально в методе. Класс `PhotoImage`, который в качестве аргумента принимает имя файла изображения, используется модулем `tkinter` для вывода на экран изображений в виджетах холста, надписи, текста или кнопки. После этого шага можно использовать изображение в надписи `tkinter`, поэтому задайте переменную `photo_lbl`, передайте ей родителя и изображение в качестве аргументов и укажите отсутствие текста и тонкую границу ❸.

Для размещения надписи в родительском окне примените метод `grid()` и передайте ему первую строку и первый столбец, сделайте так, чтобы изображение охватывало 10 столбцов, выравняйте его по левому краю, применив `W` ❹. Это позволит заполнить верхнюю часть окна изображением закрытых дверей. Настроечный параметр `columnspan` позволяет виджету охватывть несколько столбцов. Это значение не повлияет на размер изображения, но изменит количество мест, доступных для размещения текста с инструкцией и других виджетов под изображением. Например, если задать `columnspan=2`, то для размещения инструкций, кнопок и сообщений будет доступно только два столбца.

Завершите фотонадпись, создав ссылку на объект изображения ❺. Если вы этого не сделаете, то изображение не всегда будет появляться.

Согласно документации по `tkinter`, модуль `tkinter` представляет собой слой, надстраиваемый над другим программным продуктом (`Tk`), и интерфейс между ними не обрабатывает ссылки на объекты-изображения надлежащим образом. Виджет `Tk` содержит ссылку на внутренний объект, но `tkinter` этого не делает. В целях автоматического восстановления памяти, потребляемой объектами, которые больше не нужны, Python использует модуль-сборщик мусора. Когда сборщик мусора в распределителе памяти Python высвобождает объект `tkinter`, модуль `tkinter` поручает `Tk` высвободить изображение. Но поскольку оно используется, то `Tk` это сделать не в состоянии, поэтому он делает его прозрачным. Предложения по решению указанной проблемы включают использование глобальной переменной, применение экземплярного атрибута либо, как вы сделали тут, добавление атрибута в экземпляр

виджета (`photo_lbl.image = img`). Дополнительную информацию на сей счет можно получить по адресу **http://effbot.org/tkinterbook/photoimage.htm**.

Наконец, добавьте текст инструкции в виде виджетов надписей `Label`. Данная процедура заключается в предоставлении списка параметров, а затем их обходе в цикле с целью построения виджетов. Начните со списка кортежей ❻, где каждый кортеж содержит параметры для создания объекта `Label`; вы увидите, что конкретно каждый из них представляет в следующей инструкции ❼. По мере прохождения цикла `for` создайте все надписи в родительском окне и назначьте им некоторый текст. Затем примените метод `grid()`, который размещает текст в окне, основываясь на информации из списка кортежей.

В методе `grid()` используйте настроечный параметр `ipadx` ❽. Этот параметр относится к внутреннему заполнению в направлении `x` внутри надписи, поэтому с ним можно поиграть, настроив внешний вид текста в окне. В этом случае вы добавляете в надпись 30 пикселов, выравнивая текст в визуально привлекательном виде.

Создание радиокнопок и текстовых виджетов

Листинг 11.7 продолжает определение метода `create_widgets()`, создавая виджеты с радиокнопками (переключателями) для трех дверей. Игрок делает свой первоначальный выбор двери, выбирая радиокнопку **A**, **B** или **C**. Выбранный им вариант затем обрабатывается методом `win_reveal()`, который вы создадите позже. Этот метод позволит определить выигрышную дверь и выявить козу.

Еще один набор радиокнопок создается для того, чтобы получить выбранный игроком вариант — со сменой двери или без смены. Результат будет обрабатываться методом `show_final()`, также определенным позже. Помимо раскрытия предмета, который стоит за выбранным игроком окончательным вариантом двери, этот метод обновит статистику выигрышей, используя текстовые виджеты, определенные в конце данного листинга.

Листинг 11.7. Построение виджетов радиокнопок и текста для метода `create_widgets()`. Файл monty_hall_gui.py, часть 3

```
        # Создать радиокнопки для получения выбираемого
        # пользователем начального варианта
    ❶   self.door_choice = tk.StringVar()
        self.door_choice.set(None)
    ❷   a = tk.Radiobutton(self.parent, text='A',
                           variable=self.door_choice,
                           value='a', command=self.win_reveal)
        b = tk.Radiobutton(self.parent, text='B',
                           variable=self.door_choice,
                           value='b', command=self.win_reveal)
        c = tk.Radiobutton(self.parent, text='C',
                           variable=self.door_choice,
                           value='c', command=self.win_reveal)
```

```
        # создать виджеты для изменения вариантов выбора дверей
❸       self.change_door = tk.StringVar()
        self.change_door.set(None)

❹       instr_lbl = tk.Label(self.parent, text='Поменять дверь?')
        instr_lbl.grid(row=2, column=3, columnspan=1, sticky='E')

❺       self.yes = tk.Radiobutton(self.parent, state='disabled', text='Y',
                                  variable=self.change_door, value='y',
                                  command=self.show_final)
        self.no = tk.Radiobutton(self.parent, state='disabled', text='N',
                                 variable=self.change_door, value='n',
                                 command=self.show_final)

        # создать текстовые виджеты для статистики выигрышей
❻       defaultbg = self.parent.cget('bg')
❼       self.unchanged_wins_txt = tk.Text(self.parent, width=20,
                                          height=1, wrap=tk.WORD,
                                          bg=defaultbg, fg='black',
                                          borderwidth=0)
        self.changed_wins_txt = tk.Text(self.parent, width=20,
                                        height=1, wrap=tk.WORD,
                                        bg=defaultbg,
                                        fg='black', borderwidth=0)
```

Начните с создания радиокнопок для дверей A, B и C. Когда пользователь взаимодействует с виджетом `tkinter`, результатом является событие. Для отслеживания этих событий можно использовать переменные, например когда игрок выбирает дверь, нажав радиокнопку. Для переменных, специфичных для конкретного виджета, `tkinter` имеет класс переменных. Используйте класс `StringVar`, предназначенный для символьных цепочек, назначив его переменной `door_choice` ❶. Сразу же примените метод `set()`, установив указанную переменную равной `None`.

Затем настройте кнопочные виджеты для трех дверей ❷. При первом выборе двери игрок будет нажимать на одну из них. Используйте класс `Radiobutton` и передайте ему родительское окно, изображаемый текст, переменную `door_choice`, которой вы только что дали значение, равное имени двери, и команду. Команда вызывает метод `win_reveal()`, который будет определен чуть ниже. Обратите внимание, что после имени метода скобки отсутствуют.

Повторите этот процесс для кнопок B и C, который, в сущности, состоит в копировании и вставке, т. к. вам нужно лишь изменить обозначения дверей.

Теперь соберите радиокнопки для смены дверей. Начните с создания еще одной строковой переменной, как вы сделали для начального выбора двери ❸. Она будет содержать `y` либо `n` в зависимости от того, какая радиокнопка выбрана.

Постройте надпись с инструкцией, используя класс `Label` ❹. Затем постройте радиокнопку `self.yes` ❺. Используйте класс `RadioButton`, передайте ему родитель-

ское окно и установите его состояние равным `disabled` (отключено). Благодаря этому окно будет инициализироваться кнопками да/нет, затененными серым цветом, поэтому игрок не сможет бежать впереди паровоза и пробовать изменить дверь, прежде чем одна из них будет выбрана. Параметр `text` — это имя кнопки; используйте сокращение "Y" от "yes". Установите аргумент переменной виджета равным переменной `change_door`, установите его значение равным `y` и вызовите функцию `show_final()`. Повторите процедуру для кнопки "no".

Вам понадобятся еще несколько последних виджетов. Это будут текстовые виджеты для вывода на экран числа раз, когда игрок менял дверь, относительно числа раз, когда игрок бездействовал. Используйте класс `Text` для вывода статистики и установите цвет текстового поля в соответствии с родительским окном. Для этого используйте `cget()` для получения цвета фона (`bg`) родителя, а затем назначьте его переменной ❻. Метод `cget()` возвращает текущее значение параметра `tkinter` в виде символьной цепочки.

Создайте текстовый объект для изображения выигрышей за приверженность первому выбранному варианту ❼. Для этого нужно передать виджету родительское окно, ширину и высоту, способ переноса текста, если он выходит за пределы строки, цвет фона, цвет переднего плана — цвет текста — и ширину границы для текстового поля. Обратите внимание, что никакого фактического текста включать не требуется; он будет добавлен позже методом `show_final()`.

Закончите еще одним текстовым виджетом для показа числа выигрышей, приписываемых смене дверей.

Расстановка виджетов

Листинг 11.8 завершает метод `create_widgets()`, используя диспетчер геометрии `Grid` модуля `tkinter` для размещения оставшихся незарегистрированными виджетов в игровом окне.

Листинг 11.8. Вызывает метод `grid()` на виджетах с целью их размещения в рамке. Файл monty_hall_gui.py, часть 4

```
        # поместить виджеты в рамку
❶       a.grid(row=1, column=4, sticky='W', padx=20)
        b.grid(row=1, column=4, sticky='N', padx=20)
        c.grid(row=1, column=4, sticky='E', padx=20)
        self.yes.grid(row=2, column=4, sticky='W', padx=20)
        self.no.grid(row=2, column=4, sticky='N', padx=20)
❷       self.unchanged_wins_txt.grid(row=1, column=5, columnspan=5)
        self.changed_wins_txt.grid(row=2, column=5, columnspan=5)
```

Примените метод `grid()` для размещения кнопок дверей в родительском окне ❶. Сгруппируйте три кнопки двери в одной строке и столбце и отделите их с помощью липкого выравнивания `sticky`: `W` означает влево, `N` — в центре и `E` — вправо.

Используйте padx для настройки позиции сбоку. Повторите этот процесс для остальных кнопок, затем расположите текстовые виджеты со статистикой выигрышей и дайте им разместиться в пяти столбцах в правой части окна ❷.

Обновление изображения двери

Вам придется открывать и закрывать двери на протяжении всей игры, поэтому листинг 11.9 определяет вспомогательный метод, который по мере необходимости обновляет изображение двери. Обратите внимание, что благодаря ООП передавать в метод имя файла в качестве аргумента не требуется. Все методы имеют прямой доступ к атрибутам объекта, которые начинаются с ключевого слова self.

Листинг 11.9. Определяет метод обновления текущего изображения двери. Файл monty_hall_gui.py, часть 5

```
❶ def update_image(self):
      """Обновить текущее изображение двери."""
   ❷ img = tk.PhotoImage(file=self.img_file)
   ❸ self.photo_lbl.configure(image=img)
   ❹ self.photo_lbl.image = img
```

Определите функцию update_image(), которая в качестве аргумента принимает self ❶. Затем используйте класс PhotoImage, как в листинге 11.6 ❷. Имя файла, self.img_file, будет обновляться в других методах.

Поскольку вы уже создали надпись с изображением двери, то для изменения надписи используйте метод configure() — в данном случае путем загрузки нового изображения ❷. Можно использовать configure() либо config(). Завершите программу, назначив изображение атрибуту виджета с целью предотвращения сборки мусора ❹, как описано в листинге 11.6.

Выбор выигрышной двери и показ козы

В листинге 11.10 определен метод, который выбирает выигрышную дверь и дверь раскрытия, а затем открывает и закрывает дверь раскрытия. Он также активирует кнопки "да/нет", которые неактивны до тех пор, пока игрок не сделает свой первый выбор двери.

Листинг 11.10. Определяет метод случайного выбора выигрышной двери и двери раскрытия. Файл monty_hall_gui.py, часть 6

```
❶ def win_reveal(self):
      """Случайно выбрать выигрышную деврь и раскрыть невыбранную
         дверь, за которой спрятана коза."""
   ❷ door_list = list(self.doors)
   ❸ self.choice = self.door_choice.get()
      self.winner = random.choice(door_list)
```

```
❺ door_list.remove(self.winner)

❹ if self.choice in door_list:
      door_list.remove(self.choice)
      self.reveal = door_list[0]
  else:
      self.reveal = random.choice(door_list)

❻ self.img_file = ('reveal_{}.png'.format(self.reveal))
  self.update_image()

  # активировать и очистить кнопки да/нет
❼ self.yes.config(state='normal')
  self.no.config(state='normal')
  self.change_door.set(None)

  # закрыать двери через 2 секунды после открытия
❽ self.img_file = 'all_closed.png'
  self.parent.after(2000, self.update_image)
```

Определите метод `win_reveal()`, который в качестве аргумента принимает ключевое слово `self` ❶. Сразу же составьте список дверей из атрибута `doors` ❷ класса. Этот список будет изменяться на основе первого выбранного игроком варианта двери, а затем выигрышной двери, выбранной программой наугад.

Теперь назначьте атрибут `self.choice` строковой переменной `door_choice`, доступ к которой осуществляется с помощью метода `get()` ❸. Значение этого атрибута было определено радиокнопкой двери, которую пользователь нажал в качестве первого варианта выбора. Далее выберите наугад выигрышную дверь из списка дверей.

Удалите выигрышную дверь из списка дверей ❹. Затем примените условное выражение, для того чтобы увидеть, находится ли выбранный игроком вариант по-прежнему в списке дверей; если он там, то удалите его, для того чтобы он не мог быть открыт ❺. В результате в списке останется только одна дверь, поэтому назначьте ее атрибуту `self.reveal`.

Если игрок выбрал выигрышную дверь, то в списке остаются две двери, поэтому случайно выберите одну из них и назначьте ее атрибуту `self.reveal`. Обновите атрибут `self.img_file` этой двери ❻, а затем вызовите метод, который обновляет фотонадпись, для того чтобы показать новое изображение. Рис. 11.6 является примером изображения раскрытой двери B.

Затем установите кнопки "да/нет" в состояние `normal` ❼. После этого они больше не будут затенены. Завершите метод, изменив файл изображения на all_closed.png и вызвав метод `self.update_image()` в родительском окне `parent` по истечении 2000 миллисекунд ❽. Это действие держит двери открытыми не более 2 секунд.

Рис. 11.6. Изображение раскрытой двери B

Раскрытие выбранного игроком окончательного варианта

Листинг 11.11 определяет первую часть функции, которая принимает окончательно выбранную игроком дверь и показывает, что за ней находится. Указанная функция также будет отслеживать число выигрышей за смену двери или за бездействие.

Листинг 11.11. Определяет метод, который показывает окончательно выбранный игроком вариант и обновляет списки выигрышей. Файл monty_hall_gui.py, часть 7

```
❶ def show_final(self):
      """Раскрыть изображение за окончательно выбранной дверью
         и подсчитать выигрыши."""
   ❷ door_list = list(self.doors)

   ❸ switch_doors = self.change_door.get()

   ❹ if switch_doors == 'y':
         door_list.remove(self.choice)
         door_list.remove(self.reveal)
      ❺ new_pick = door_list[0]
      ❻ if new_pick == self.winner:
            self.img_file = 'money_{}.png'.format(new_pick)
            self.pick_change_wins += 1
         else:
            self.img_file = 'goat_{}.png'.format(new_pick)
            self.first_choice_wins += 1
   ❼ elif switch_doors == 'n':
      ❽ if self.choice == self.winner:
            self.img_file = 'money_{}.png'.format(self.choice)
            self.first_choice_wins += 1
         else:
            self.img_file = 'goat_{}.png'.format(self.choice)
            self.pick_change_wins += 1
```

```
        # обновить изображение двери
    ⓫ self.update_image()
```

Определите метод с именем `show_final()`, который в качестве аргумента принимает — как вы уже догадались — ключевое слово `self` ❶. Сделайте новую копию списка дверей ❷, а затем получите переменную `self.change_doors` и назначьте ей атрибут `switch_doors` ❸. Эта переменная будет содержать либо `'y'`, либо `'n'`, в зависимости от того, какую радиокнопку игрок нажал.

Если игрок решил поменять двери ❹, то удалите из списка его первый вариант выбора и открытую дверь и назначьте оставшейся двери атрибут `new_pick` ❺. Если этот новый вариант выбора является выигрышной дверью ❻, то сошлитесь на правильное изображение и продвиньте счетчик `self.pick_change_wins`. В противном случае поменяйте изображение на козу и продвиньте счетчик `self.first_choice_wins`.

Если игрок решает не менять двери ❼ и его первым вариантом выбора была выигрышная дверь ❽, то покажите денежный мешок и продвиньте счетчик `self.first_choice_wins`. В противном случае покажите козу и продвиньте счетчик `self.pick_change_wins`.

Завершите программу, вызвав метод `update_image()` для обновления изображения ❾. Опять же, передавать ему имя нового файла изображения не требуется, т. к. он может обратиться к атрибуту `self.img_file`, который вы изменили в предыдущем фрагменте кода.

Вывод статистики на экран

Листинг 11.12 завершает метод `show_final()` обновлением игрового окна статистикой числа выигрышей, деактивированием кнопок "да/нет" и закрытием всех дверей.

Листинг 11.12. Выводит на экран статистику выигрышей, деактивирует кнопки "да/нет" и закрывает все двери. Файл monty_hall_gui.py, часть 8

```
        # обновить выводимую на экран статистику
    ❶ self.unchanged_wins_txt.delete(1.0, 'end')
    ❷ self.unchanged_wins_txt.insert(1.0,
                                     'Выигрыши за бездействие = {:d}'
                                      .format(self.first_choice_wins))
      self.changed_wins_txt.delete(1.0, 'end')
      self.changed_wins_txt.insert(1.0, 'Выигрыши за смену двери = {:d}'
                                   .format(self.pick_change_wins))

        # деактивировать кнопки да/нет и очистить кнопки выбора дверей
    ❸ self.yes.config(state='disabled')
      self.no.config(state='disabled')
    ❹ self.door_choice.set(None)
```

```
❺ закрывать двери через 2 секунды после открытия
   self.img_file = 'all_closed.png'
   self.parent.after(2000, self.update_image)
```

Начните с удаления любого текста в текстовом виджете `self.unchanged_wins_txt` ❶. При этом начните удалять текст в индексе `1.0`. Так как используется формат `line.column`, вы указываете первую строку и первый столбец текстового виджета (нумерация строк начинается с 1, нумерация столбцов — с 0). В конце поставьте спецификатор `'end'`, который обеспечивает удаление всего текста после начального индекса.

Затем примените метод `insert()` для добавления значения атрибута `self.first_choice_wins` вместе с некоторым описательным текстом в текстовый виджет ❷. Начните вставку с текстового индекса `1.0`.

Повторите этот процесс для текстового виджета `self.changed_wins_txt`, а затем деактивируйте кнопки "да/нет", установив их состояние конфигурации `config` равным `'disabled'` (отключено) ❸. Установите строковую переменную `self.door_choice` опять равной `None`, и вы будете готовы начать новую игру ❹.

Завершите метод, закрыв двери, как это было сделано в листинге 11.10 ❺.

Настройка корневого окна и запуск событийного цикла

Листинг 11.13 завершает программу monty_hall_gui.py, настраивая корневое окно модуля `tkinter`, создавая экземпляр объекта игры и запуская главный цикл `mainloop()`. Как вариант, этот код может быть инкапсулирован в функцию `main()`.

Листинг 11.13. Настраивает корневое окно, создает объект игры и запускает главный цикл `mainloop()`. Файл monty_hall_gui.py, часть 9

```
   # настроить корневое окно и выполнить событийный цикл
❶ root = tk.Tk()
❷ root.title('Задача Монти Холла')
❸ root.geometry('1280x820')  # Размерность изображений 1280 x 720
❹ game = Game(root)
   root.mainloop()
```

Экземпляр класса `Tk` инстанцируется без аргументов ❶. В результате создается верхнеуровневый виджет `tkinter`, который будет главным окном игрового приложения. Назначьте его переменной с именем `root`.

Назначьте окну заголовок ❷ и размер в пикселах ❸. Обратите внимание, что размер изображений влияет на геометрию, позволяя им привлекательно вписываться в окно, с достаточным пространством ниже для сообщений и инструкций по игре.

Теперь создайте игру ❹. Передайте ей корневое окно `root`, т. е. главное окно, которое будет содержать игру, в результате чего новая игра будет помещена в это окно.

Завершите программу, вызвав метод `mainloop()` в окне `root`, который оставляет окно открытым и ожидает обработки событий.

Резюме

В этой главе вы применили простую симуляцию Монте-Карло с целью подтверждения того, что смена дверей в задаче Монти Холла является наилучшей стратегией. Затем вы применили модуль `tkinter` для создания забавного интерфейса, который позволяет школьникам проверить этот вывод вручную, игра за игрой. Но лучше всего то, что вы узнали, как использовать объектно-ориентированное программирование для построения интерактивных виджетов, которые реагируют на вводимые пользователем данные.

Дальнейшее чтение

Полезные ссылки на модуль `tkinter` можно найти в *разд. "Дальнейшее чтение" главы 11*.

Краткое изложение задачи Монти Холла 1990 года можно найти в Интернете по адресу **http://marilynvossavant.com/game-show-problem/**.

Практический проект: парадокс дня рождения

Сколько людей должно находиться в комнате для появления такого шанса 50/50, что двое из них родились в один и тот же день одного и того же месяца? Согласно парадоксу дня рождения, не так уж и много! Как и в случае с задачей Монти Холла, результат противоречит здравому смыслу.

Примените симуляцию Монте-Карло для определения числа людей, необходимого для того, чтобы достичь отметки 50%. Программа должна выводить на экран число людей и вероятность для нескольких присутствующих в комнате. Если вы обнаружите, что ищете способ форматирования дат, то остановитесь и упростите! Решение указанной задачи можно найти в программе birthday_paradox_practice.py в приложении к книге либо на веб-сайте по адресу **https://www.nostarch.com/impracticalpython/**.

12

ОБЕСПЕЧЕНИЕ СОХРАННОСТИ ПЕНСИОННЫХ СБЕРЕЖЕНИЙ

Бэбибумеры — это американцы, родившиеся между 1946 и 1964 гг. Они образуют большую демографическую когорту — около 20% американского населения — и поэтому оказали огромное влияние на все аспекты американской культуры. Финансовая индустрия быстро удовлетворила их потребности, которые на протяжении десятилетий касались роста инвестиций. Но в 2011 г. самые старые бумеры достигли возраста 65 лет и начали толпами уходить на пенсию, со скоростью 10 тыс. человек *в день*! При средней продолжительности жизни больше, чем у предыдущих поколений, бумер может наслаждаться жизнью на пенсии соразмерно продолжительности его карьеры. Финансирование этого 30–40-летнего периода представляет большую проблему и является большим бизнесом.

В те годы, когда финансовые консультанты были в основном сосредоточены на росте благосостояния бумеров, они полагались на простое *правило 4 процентов* при планировании жизни на пенсии. Проще говоря, каждый год, когда вы находитесь на пенсии, если вы тратите сумму, не превышающую 4% сбережений, которые у вас были в первый год выхода на пенсию, то вы никогда не истратите все деньги. Но как заметил Марк Твен, "все обобщения — ложны, в том числе и это!". Стоимость наших инвестиций и сумма, которую мы тратим, постоянно меняются, часто из-за сил, находящихся вне нашего контроля.

В качестве более изощренной альтернативы правилу 4 процентов финансовая индустрия приняла симуляцию Монте-Карло (см. обзор симуляции Монте-Карло в *главе 11*). С помощью указанного метода можно проверять и сравнивать пенсионные стратегии на тысячах жизней. Цель состоит в том, чтобы выявить количество денег, которое можно тратить каждый год без исчерпания сбережений с учетом ожидаемой продолжительности жизни на пенсии.

Преимущество симулирования Монте-Карло перед другими методами возрастает по мере увеличения источников неопределенности. В *главе 11* вы применили симуляцию Монте-Карло к одной-единственной величине с простым распределением вероятности. Здесь вы обратитесь к неопределенности, связанной с продолжительностью жизни, а также уловите истинную цикличность и взаимозависимость фондового и облигационного рынков и инфляции. Это позволит вам оценить и сравнить разные стратегии с целью достижения обеспеченной и счастливой жизни на пенсии.

Проект 20: симулирование продолжительности жизни на пенсии

Если вы думаете, что вы слишком молоды, для того чтобы беспокоиться о жизни на пенсии, то подумайте еще раз. Бэбибумеры думали то же самое, и теперь более половины из них имеют сбережения, недостаточные для пенсионной жизни. Для большинства пенсионеров различие между употреблением в пищу японской мраморной говядины или собачьего корма означает то, как скоро они начинают экономить. Благодаря волшебству начисления сложных процентов даже скромные сбережения могут накапливаться десятилетиями. Зная заранее цифры, которые вам понадобятся позже, вы сможете установить реалистичные цели для безболезненного перехода к вашим "золотым годам".

Цель

Построить симуляцию Монте-Карло для оценки вероятности исчерпания пенсионных денег. Рассмотреть годы пенсионной жизни как ключевую неопределенность и использовать исторические фондовые, облигационные и инфляционные данные, для того чтобы уловить наблюдаемую цикличность и зависимость этих переменных величин.

Стратегия

Планируя свой проект, не стесняйтесь обратиться к конкурентам. Многочисленные калькуляторы пенсионных сбережений доступны онлайн бесплатно. Если вы с ними поиграете, то увидите, что они показывают высокий уровень изменчивости входных параметров.

Калькуляторы со многими параметрами на вид могут показаться лучше (рис. 12.1), но с добавлением каждой новой детали вы начинаете спускаться в кроличьи норы, в особенности когда речь заходит о сложном Налоговом кодексе США. Когда вы предсказываете результаты на 30–40 лет вперед, детали могут стать шумом. Поэтому лучше придерживаться простоты, сосредоточившись на самых важных и контролируемых вопросах. Можно контролировать время выхода на пенсию, распределение своих инвестиционных активов, сколько средств экономится и сколько тратится, но невозможно контролировать фондовый рынок, процентные ставки и инфляцию.

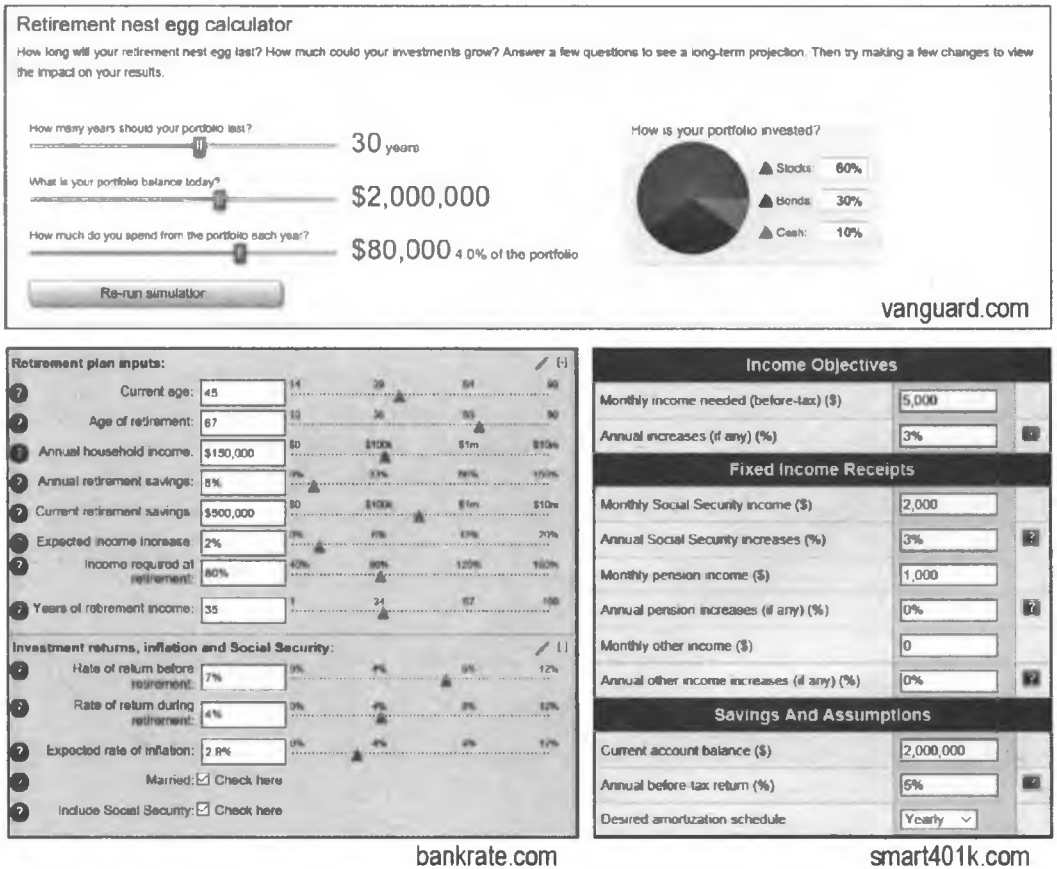


Рис. 12.1. Пример панелей ввода
в трех онлайновых калькуляторах пенсионных сбережений

Когда вы не можете узнать "правильный" ответ на поставленную задачу, лучше всего посмотреть на ряд сценариев и принять решения, основанные на вероятностях. Для решений, которые связаны с "фатальной ошибкой", например с нехваткой денег, желательными решениями являются те, которые снижают вероятность этого события.

Прежде чем начать, вы должны научиться говорить на соответствующем языке, поэтому я собрал список финансовых терминов, которые вы будете использовать в этом проекте.

Облигации (Bonds). Облигация — это долговые инвестиции, при которых вы ссужаете деньги организации — обычно правительству или корпорации — на определенный срок. Заемщик выплачивает вам согласованную процентную ставку (облигационный доход) и в конце срока возвращает полную стоимость ссуды с учетом допущения, что эмитент не разорится и не объявит дефолт. Стоимость облигаций с течением времени может повышаться и понижаться, и поэтому вы можете потерять деньги, если продадите облигации досрочно. Облигации привлекательны для пенсионеров, поскольку они предлагают безопасные, стабильные, предсказуемые

финансовые возвраты[1]. Казначейские облигации, выпущенные правительством США, считаются самыми безопасными из всех. К сожалению, возвратность большинства облигаций, как правило, является низкой, и поэтому они уязвимы перед инфляцией.

Эффективная налоговая ставка (Effective tax rate). Это средняя ставка, по которой физическое лицо или супружеская пара облагается налогом. Она включает в себя местные, государственные и федеральные налоги. Налоги могут быть сложными, с большими различиями в ставках государственных и местных налогов, многими возможностями для вычетов и корректировок, а также переменными ставками для разных видов доходов (например, краткосрочный и долгосрочный прирост капитала). Налоговый кодекс также является прогрессивным, т. е. вы платите пропорционально больше по мере увеличения вашего дохода. По данным компании Motley Fool, предоставляющей финансовые услуги, совокупная ставка налога на прибыль среднего американца в 2015 г. составила 29,8%. И сюда не входят налоги с продаж и имущества! Вы также можете рассчитывать на то, что по крайней мере один раз в 30-летний пенсионный срок конгресс будет эти ставки пересматривать. В связи с этими сложностями в данном проекте вы должны корректировать свой параметр вывода средств (расходы) с учетом налогов.

Индекс (Index). Безопаснее всего не класть все свои (пенсионные) яйца в одну корзину, а инвестировать их в разные активы. Индекс — это гипотетический портфель ценных бумаг или группа корзин, предназначенных для того, чтобы представлять широкую часть финансового рынка. Например, индекс Standard & Poor 500 (S&P 500)[2] представляет 500 крупнейших компаний США, которые в основном выплачивают дивиденды. Индексные инвестиции, такие как индексный взаимный фонд, позволяют инвестору без труда купить одну ценную бумагу, которая содержит акции сотен компаний.

Инфляция (Inflation). Это рост цен с течением времени из-за увеличения спроса, девальвации валюты, роста цен на энергоносители и т. д. Инфляция является коварным разрушителем благосостояния. Темп инфляции является переменным, но с 1926 г. в США он в среднем составляет около 3% в год. При таких темпах стоимость денег уменьшается вдвое каждые 24 года. Небольшая инфляция (от 1 до 3%) обычно указывает на то, что экономика растет, а заработная плата поднимается. Более высокая инфляция и отрицательная инфляция являются нежелательными.

---

[1] Финансовый возврат (return) на инвестированный капитал — это упрощенно разница между ценой актива в предыдущий период времени и его ценой в настоящий период. Возврат в процентном соотношении вычисляется либо как частное этих показателей, либо по формуле $R_t = (P_t - P_{t-1})/P_{t-1}$, где P_{t-1} — это цена актива в предыдущем периоде, причем последняя формула представляет собой не что иное, как расчет надбавки/скидки, применяемой в розничной торговле. Возвратность (rate of return) за весь интересуемый интервал времени k вычисляется как $(1 + R_1) \cdots (1 + R_k)$. — *Прим. перев.*

[2] Standard & Poor's — это дочерняя компания американской корпорации McGraw-Hill, занимающаяся аналитическими исследованиями финансовых рынков. Компания принадлежит к тройке самых влиятельных международных рейтинговых агентств, известна главным образом как создатель и редактор американского фондового индекса S&P 500.
См. **https://ru.wikipedia.org/wiki/Standard_%26_Poor%E2%80%99s**. — *Прим. перев.*

Число случаев (Number of cases). Это испытания или прогоны, выполняемые во время симуляции Монте-Карло; каждый случай представляет собой одну продолжительность жизни на пенсии и симулируется новым множеством случайно отобранных значений. В выполняемых вами симуляциях где-то между 50 тыс. и 100 тыс. случаев должны обеспечивать надлежащий повторимый ответ.

Вероятность разорения (Probability of ruin). Это вероятность исчерпания денег до окончания пенсионного срока. Ее можно рассчитать как число случаев, которые заканчиваются без денег, деленное на суммарное число случаев.

Начальная стоимость (Start value). Это суммарная стоимость ликвидных инвестиций, включая текущие (расчетные) счета, брокерские счета, индивидуальные пенсионные счета с отсроченным налоговым платежом и т. д., которыми владеют на момент начала пенсионного срока. Она не приравнивается к чистой стоимости, которая включает в себя такие активы, как дома, автомобили и яйца Фаберже.

Акции (Stocks). Акция — это ценная бумага (финансовое обеспечение безопасности, от англ. *security*), которая означает право собственности в корпорации и представляет собой требование на часть активов и доходов корпорации. Многие акции выплачивают дивиденды, регулярные выплаты, аналогичные процентам, выплачиваемым по облигациям или банковскому счету. Для среднего человека акции — это самый быстрый способ увеличить свое благосостояние, но они не лишены риска. Цена акции в течение короткого времени может быстро подниматься и опускаться как из-за производительности компании, так и из-за спекуляций, возникающих из-за жадности или страха инвесторов. Пенсионеры, как правило, инвестируют в крупнейшие компании США, выплачивающие дивиденды, потому что они предлагают регулярный доход и менее волатильную цену акций, чем небольшие компании.

Суммарные финансовые возвраты (Total returns). Сумма прироста капитала (изменения стоимости активов, таких как цена акций), процентов и дивидендов считается суммарным финансовым возвратом от инвестиций. Он обычно котируется на ежегодной основе.

Вывод средств (Withdrawal). Также именуемый затратами или расходами, вывод средств — это валовой доход до налогов, который вам понадобится для покрытия всех расходов в данном году. В случае правила 4 процентов он будет представлять собой 4% начальной стоимости в первый год выхода на пенсию. В каждом последующем году это число должно корректироваться с учетом инфляции.

Исторические финансовые возвраты имеют значение

Симуляторы пенсионных сбережений, которые используют постоянные значения для инфляции и финансовых возвратов от инвестиций (рис. 12.1), грубо искажают реальность. В целом предсказательная способность значит столько же, сколько значат лежащие в ее основе допущения, а возвратность может быть очень волатильной, взаимозависимой и цикличной. Эта волатильность влияет на пенсионеров больше всего, когда начало пенсионного срока — или большие неожиданные расходы — совпадает с началом большого рыночного спада.

График на рис. 12.2 показывает годовые финансовые возвраты индекса S&P 500 крупнейших американских компаний и 10-летних казначейских облигаций, т. е. достаточно безопасных, среднерисковых инвестиций с фиксированным доходом. Он также включает в себя ежегодные темпы инфляции и значительные финансовые события, такие как Великая депрессия.

Рис. 12.2. Годовой темп инфляции плюс суммарная возвратность на фондовом и облигационном рынках, 1926–2013 гг.

Длительное изучение трендов (рис. 12.2) финансовыми исследователями привело к некоторым полезным наблюдениям о рынках США.

- Восходящие (бычьи) рынки, как правило, длятся в пять раз дольше, чем нисходящие (медвежьи) рынки.
- Вредные высокие темпы инфляции могут сохраняться в течение всего десятилетия.
- Облигации, как правило, обеспечивают низкие финансовые возвраты, которые с трудом идут в ногу с инфляцией.
- Финансовые возвраты от акций легко опережают инфляцию, но ценой большой ценовой волатильности.
- Финансовые возвраты акций и облигаций часто обратно коррелируют; это означает, что возвратность облигаций уменьшается по мере увеличения возвратности акций, и наоборот.
- Ни акции крупных компаний, ни казначейские облигации не могут гарантировать вам легкую прогулку.

Основываясь на этой информации, финансовые консультанты рекомендуют большинству пенсионеров иметь диверсифицированный портфель, включающий не-

сколько типов инвестиций. Эта стратегия использует один тип инвестиций в качестве "хеджирования" против другого, демпфируя максимумы, но поднимая минимумы, тем самым теоретически снижая волатильность.

На рис. 12.3 годовая возвратность инвестиций строится с использованием S&P 500 и гипотетического 40/50/10-процентного сочетания соответственно S&P 500, 10-летних казначейских облигаций и наличных денежных средств. Трехмесячный казначейский вексель, являющийся очень краткосрочной облигацией со стабильной ценой и низкой возвратностью (все равно что деньги, засунутые под матрас), представляет наличные денежные средства.

Рис. 12.3. Годовая возвратность S&P 500 по сравнению с сочетанием S&P 500, 10-летних казначейских облигаций и наличных денежных средств, 1926–2013 гг.

Этот диверсифицированный портфель обеспечивает более легкую прогулку, чем только фондовый рынок, в то же время обеспечивая защиту от инфляции. Но он явно даст другие результаты, чем онлайновые калькуляторы, которые исходят из допущения, что возвратность *всегда* остается постоянной и положительной по величине.

Используя исторические данные, вы улавливаете истинную измеренную продолжительность хороших и плохих времен, а также самые высокие максимумы и самые низкие минимумы. Вы также учитываете то, что полностью игнорируется правилом 4 процентов: *черного лебедя*.

Черные лебеди — это логически вытекающие, невероятные события. Они могут быть хорошими, как встреча с вашим супругом, либо плохими, как Черный понедельник, крах фондового рынка в октябре 1987 г. Преимущество симуляции Монте-Карло заключается в том, что она может учитывать эти неожиданные события; а недостаток в том, что вам нужно их запрограммировать, и если они действительно являются непредсказуемыми, то как узнать, что включать?

Черные лебеди, которые уже произошли, подобные Великой депрессии, запечатлены в годовых величинах в списках исторических возвратов. Поэтому общий подход заключается в использовании исторических результатов и допущении того, что в будущем не произойдет ничего хуже или лучше. Когда в симуляции используются данные Великой депрессии, моделируемый портфель будет испытывать то же самое поведение акций, облигаций и инфляции, что и реальные портфели в то время.

Если использование прошлых данных кажется слишком ограничительным, то прошлые результаты всегда можно отредактировать с тем, чтобы отразить более низкие минимумы и более высокие максимумы. Но большинство людей практичны и счастливее, имея дело с событиями, о которых *известно*, что они произошли — в отличие от зомби-апокалипсисов или инопланетных вторжений, — поэтому истинные исторические результаты предлагают заслуживающий доверие способ вбрызнуть реальность в финансовое планирование.

Некоторые экономисты утверждают, что данные по инфляции и возвратности до 1980 г. имеют ограниченное применение, поскольку в настоящее время Федеральная резервная система играет более активную роль в денежно-кредитной политике и контроле инфляции. С другой стороны, именно такое мышление оставляет нас беззащитными перед черными лебедями!

Самая большая неопределенность

Самая большая неопределенность в пенсионном планировании заключена в той дате, когда вы — либо ваш оставшийся в живых супруг — умрете, которую финансовые консультанты эвфемистически именуют "концом плана". Эта неопределенность влияет на каждое решение, связанное с пенсией, например когда вы выходите на пенсию, сколько вы тратите, находясь на пенсии, когда вы начинаете принимать социальное обеспечение, сколько вы оставляете своим наследникам и т. д.

Страховые компании и правительство работают с этой неопределенностью с помощью *актуарных таблиц продолжительности жизни*. Исходя из опыта смертности населения, актуарные таблицы продолжительности жизни предсказывают ожидаемый срок жизни на пенсии в данном возрасте, выраженный как среднее оставшееся число лет, ожидаемое до наступления смерти. Таблицу продолжительности жизни для страхования жизни можно найти по адресу **https://www.ssa.gov/oact/STATS/table4c6.html**. Исходя из этой таблицы, 60-летняя женщина в 2014 г. имела бы ожидаемый срок жизни на пенсии, равный 24,48 годам; это означает, что конец плана произойдет в течение ее 84-го года жизни.

Актуарные таблицы прекрасно работают для больших групп населения, но для отдельных людей они являются лишь отправной точкой. При подготовке своего собственного пенсионного плана следует проэкзаменовать ряд величин, приспособленных под семейную историю и личные проблемы со здоровьем.

Для того чтобы справиться с этой неопределенностью в своей симуляции, рассматривайте годы на пенсии как *случайную величину*, значения которой выбираются случайно из частотного распределения. Например, можно ввести наиболее вероят-

ное, минимальное и максимальное число лет, в течение которых вы ожидаете находиться на пенсии, и использовать эти значения для построения треугольного распределения. Наиболее вероятное значение может быть взято из актуарной таблицы, но конечные точки должны основываться на видах относительно вашего личного здоровья и семейной истории.

Примерный результат, основанный на треугольном распределении, по годам на пенсии для 60-летнего мужчины показан на рис. 12.4. Минимальный пенсионный срок был установлен равным 20 годам, наиболее вероятный — 22 годам, а максимальный — 40 годам. Число значений, выбираемых из распределения, составило 1000.

Рис. 12.4. Число продолжительностей жизни в сравнении с годами на пенсии на основе 1000 значений, выбранных из треугольного распределения

Как видите, для симуляции доступен каждый возможный временной интервал между минимальным и максимальным значениями, но интервалы сужаются по частоте от наиболее вероятного значения до максимального значения, указывая на то, что жизнь до 100 лет возможна, но маловероятна. Отметим также, что график значительно скошен в высокую сторону. Этим обеспечиваются консервативные результаты, поскольку с финансовой точки зрения преждевременная смерть является оптимистичным результатом, а жизнь, более продолжительная, чем ожидалось, представляет наибольший финансовый риск.

Качественный способ представления результатов

Трудность с симуляцией Монте-Карло состоит в отыскании смысла в тысячах прогонах и представлении результатов в легко усваиваемом виде. Большинство онлайновых калькуляторов представляют результаты, используя график, подобный графику на рис. 12.5. В этом примере для симуляции из 10 тыс. прогонов калькулятор строит график нескольких отобранных результатов, где возраст расположен по оси x и значение инвестиций — по оси y. Кривые сходятся слева на начальном значении инвестиций при выходе на пенсию и заканчиваются справа их значением

в конце плана. Также может быть представлена совокупная вероятность того, что во время пенсионного срока деньги останутся. Финансовые консультанты считают вероятности ниже 80–90% рискованными.

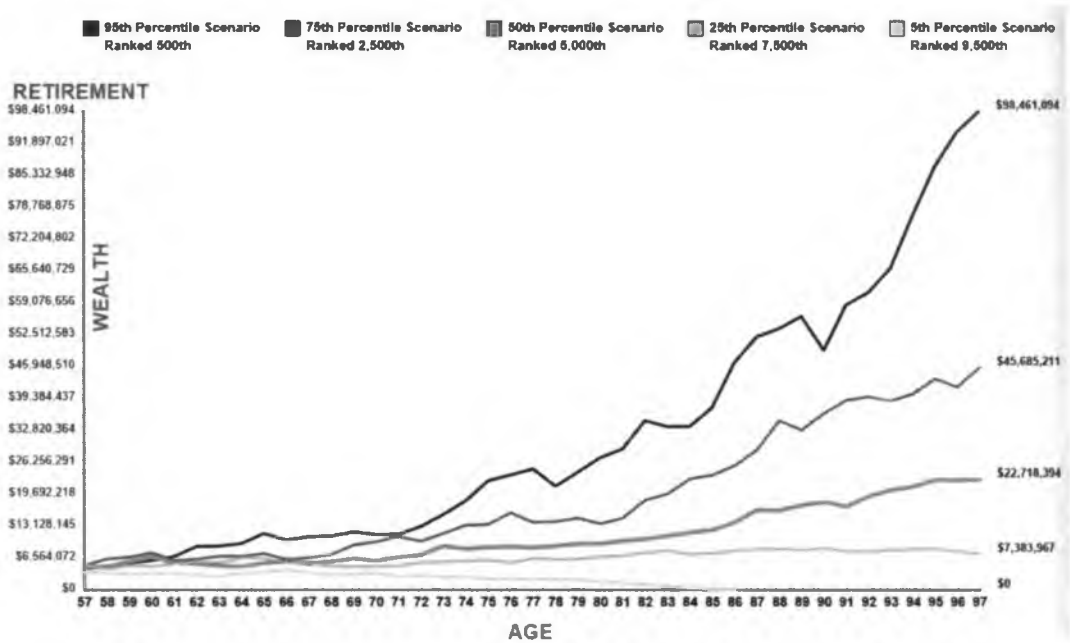


Рис. 12.5. Пример изображения из типичного симулятора пенсионных накоплений в финансовой отрасли

Наиболее важной частью информации из этого типа анализа является вероятность исчерпания денег. Также интересно посмотреть конечные и средние результаты и сводку входных параметров. В вашем симуляторе на языке Python вы можете распечатать их в окне интерпретатора, как показано ниже:

```
Тип инвестиций: облигации
Начальная стоимость: $ 1 000 000
Годовой вывод средств: $ 40 000
Годы на пенсии (мин.-наиболее вероятн.-макс.): 17-25-40
Число прогонов: 20 000

Шансы исчерпания денег: 36.1%

Средний результат: $883 843
Минимальный результат: $0
Максимальный результат: $7 607 789
```

Что касается графического представления, то вместо дублирования того, что сделали другие, давайте найдем новый способ представления результатов. Подмножество результатов каждого случая, т. е. деньги, остающиеся в конце пенсионного срока, можно представить в виде вертикальной линии на столбчатом графике (рис. 12.6).

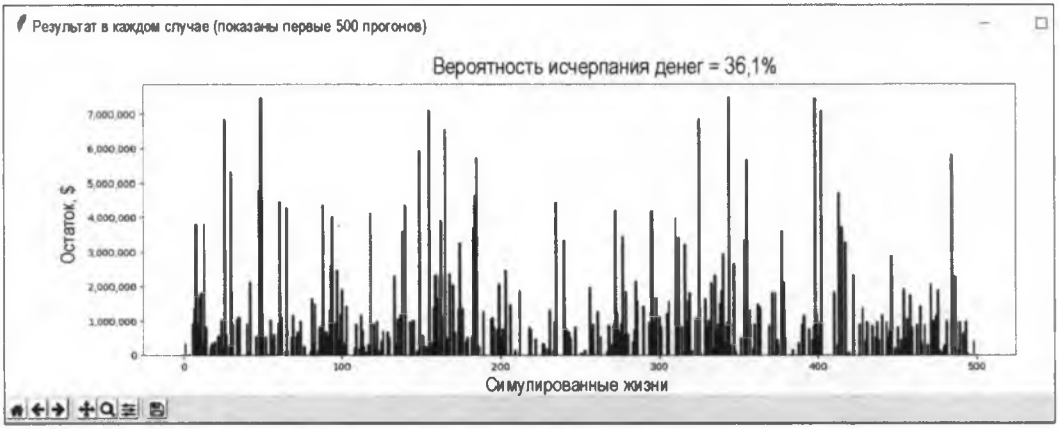


Рис. 12.6. Результаты симулированных пенсионных сроков, показанные на столбчатом графике в виде вертикальных столбиков

На данном графике каждый столбик представляет собой пенсионную часть одной смоделированной продолжительности жизни, а высота каждого столбика — деньги, оставшиеся в конце этого срока. Поскольку каждый столбик представляет не интервалы непрерывного измерения, а отдельную категорию, то столбики можно расположить в любом порядке, не затрагивая данные. Зазоры, представляющие случаи, когда деньги были исчерпаны, могут включаться в порядке их возникновения в симуляции. Вместе с количественной статистикой, регистрируемой в окне интерпретатора, этот график обеспечивает качественный способ представления результатов.

Пики и впадины этого графика представляют собой меняющиеся судьбы многих возможных фьючерсов. В одной жизни вы можете умереть нищим, а в другой мультимиллионером. Это напоминает старую поговорку "Иду туда, если бы не милость Божья", но с другой стороны, она подкрепляет замечание генерала Эйзенхауэра о том, что "планы бесполезны, но планирование необходимо". С помощью финансового планирования вы можете "поднять впадины" на графике и устранить либо значительно уменьшить ваши шансы разориться на пенсии.

Для создания указанного график вы будете использовать библиотеку `matplotlib`, которая поддерживает двухмерное и рудиментарное трехмерное графопостроение. Дополнительные сведения о библиотеке `matplotlib` и ее установке *см. в разд. "Код вероятности обнаружения" главы 10.*

Псевдокод

Основываясь на предыдущем обсуждении, мы должны сфокусировать стратегию разработки программы на нескольких важных параметрах жизни на пенсии и симулировании результатов с использованием исторического поведения финансовых рынков. Ниже приведен высокоуровневый псевдокод:

```
Получить от пользователя входное значение типа инвестиций (акции, облигации или их сочетание)
Сопоставить выбранный вариант типа инвестиций со списком исторических финансовых возвратов
Получить от пользователя входное значение начальной стоимости инвестиций
```

```
Получить от пользователя входное значение начальной годовой суммы выводимых средств
Получить от пользователя входное значение минимальной, наиболее вероятной и
  максимальной продолжительностью жизни на пенсии
Получить от пользователя входное значение числа выполняемых случаев
Создать список для хранения результатов
Перебрать случаи в цикле:
  Для каждого случая:
    Извлечь случайную непрерывную выборку из списка финансовых возвратов
     для пенсионного срока
    Извлечь тот же интервал из инфляционного списка
    Для каждого года в выборке:
      Если год не равен году 1:
        Скорректировать выводимые средства на инфляцию
      Вычесть выводимые средства из инвестиций
      Скорректировать инвестиции на финансовые возвраты
      Если инвестиции <= 0:
        Инвестиции = 0
        Выйти из цикла
  Добавить значение инвестиций в список результатов
Вывести на экран входные параметры
Вычислить и вывести на экран вероятность разорения
Вычислить и вывести на экран статистику
Вывести на экран подмножество результатов в виде столбчатого графика
```

Поиск исторических данных

Информацию о финансовых возвратах и инфляции можно найти на многочисленных веб-сайтах (несколько примеров *см. в разд. "Дальнейшее чтение" далее в этой главе*), но я уже собрал необходимую информацию в виде серии скачиваемых текстовых файлов. Если вы решите составить собственные списки, имейте в виду, что оценки инфляции и возвратности от сайта к сайту могут незначительно различаться.

Для финансовых возвратов я использовал три инвестиционных инструмента: фондовый индекс S&P 500, 10-летние казначейские облигации и трехмесячный казначейский вексель — все за период с 1926 по 2013 г. (значения казначейского векселя за 1926–1927 гг. являются оценочными). Я применил эти данные для генерирования дополнительных смешанных финансовых возвратов за тот же период. В следующем ниже списке описаны имена файлов и их содержимое:

- SP500_returns_1926-2013_pct.txt — суммарные финансовые возвраты по индексу S&P 500 (1926–2013 гг.);
- 10-yr_TBond_returns_1926-2013_pct.txt — суммарные финансовые возвраты по 10-летним казначейским облигациям (1926–2013 гг.);
- 3_mo_TBill_rate_1926-2013_pct.txt — курс трехмесячного казначейского векселя (1926–2013 гг.);

- S-B_blend_1926-2013_pct.txt — 50/50-процентное сочетание S&P 500 и 10-летних казначейских облигаций (1926–2013 гг.);
- S-B-C_blend_1926-2013_pct.txt — 40/50/10-процентное сочетание S&P 500, 10-летних казначейских облигаций, трехмесячных казначейских векселей (1926–2013 гг.);
- annual_infl_rate_1926-2013_pct.txt — среднегодовой темп инфляции в США (1926–2013 гг.).

Вот пример первых семи строк текстового файла с данными по S&P 500:

```
11.6
37.5
43.8
-8.3
-25.1
-43.8
-8.6
```

Значения являются процентами, но вы измените их на десятичные значения при загрузке их в программный код. Обратите внимание, что годы не включены, т. к. значения находятся в хронологическом порядке. Если все файлы охватывают один и тот же временной интервал, то фактические годы не важны, хотя для ведения хорошей бухгалтерии вы должны включить их в имя файла.

Код

Назовите свой симулятор пенсионных накоплений nest_egg_mcs.py. Вам понадобятся текстовые файлы, описанные в предыдущем *разд. "Отыскание исторических данных"*. Скачайте эти файлы с веб-сайта **https://www.nostarch.com/impracticalpython/** и сохраните их в той же папке, что и nest_egg_mcs.py.

Импортирование модулей и определение функций загрузки внешних данных и получения данных от пользователя

Листинг 12.1 импортирует модули и определяет функцию чтения исторических данных о возвратности и инфляции, а также еще одну функцию получения входных данных от пользователя. После доведения программы до рабочего состояния вы можете свободно изменить или добавить исторические данные с целью проведения собственных экспериментов.

Листинг 12.1. Импортирует модули и определяет функции загрузки внешних данных и получения входных данных от пользователя. Файл nest_egg_mcs.py, часть 1

```
import sys
import random
import matplotlib.pyplot as plt ❶
```

```
❶ def read_to_list(file_name):
      """Открыть файл данных в процентах, конвертировать
         в десятичную форму и вернуть список."""
   ❸ with open(file_name) as in_file:
      ❹ lines = [float(line.strip()) for line in in_file]
      ❺ decimal = [round(line / 100, 5) for line in lines]
      ❻ return decimal

❼ def default_input(prompt, default=None):
      """Позволить при вводе данных использовать значения по умолчанию."""
   ❽ prompt = '{} [{}]: '.format(prompt, default)
   ❾ response = input(prompt)
   ❿ if not response and default:
         return default
     else:
         return response
```

Инструкции `import` уже должны быть хорошо знакомы. Библиотека `matplotlib` необходима для построения столчатого графика результатов. Вам нужен только функционал построения графика, как указано в инструкции `import` ❷.

Затем определите функцию `read_to_list()`, которая загружает файл данных и обрабатывает его содержимое ❺. В качестве аргумента вы передадите ей имя файла.

Откройте файл инструкцией `with`, которая в конце закроет его автоматически ❸, а затем примените операцию включения в список для построения списка содержимого ❹. Сразу же конвертируйте элементы списка, выраженные в процентах, в десятичную форму, округлив до пяти знаков после точки ❺. Исторические результаты обычно представлены в форме не более чем до двух знаков после точки, поэтому округления до пяти знаков должно быть достаточно. В отдельных используемых здесь файлах данных можно заметить более высокую точность значений, но это всего лишь результат некоторой предварительной обработки в Excel. Завершите функцию, вернув список `decimal` ❻.

Теперь определите функцию `default_input()`, которая получает входные данные от пользователя ❼. Указанная функция в качестве аргументов принимает подсказку и значение по умолчанию, которые будут указаны при вызове функции, и программа покажет значение по умолчанию в скобках ❽. Назначьте переменной `response` введенное пользователем значение ❾. Если пользователь ничего не вводит и значение по умолчанию существует, то возвращается значение по умолчанию; в противном случае возвращается ответ пользователя ❿.

Получение входных данных от пользователя

Листинг 12.2 загружает файлы данных, ставит в соответствие полученным спискам простые имена с помощью словаря и получает входные данные от пользователя. Словарь будет использоваться для предоставления пользователю выбора из

нескольких типов инвестиций. В целом вводимые пользователем значения состоят из:

- типа используемых инвестиций (акции, облигации или сочетание того и другого);
- начальной стоимости пенсионных накоплений;
- суммы ежегодного вывода средств или расходы;
- минимального, наиболее вероятного и максимального числа лет, которые ожидается прожить на пенсии;
- числа выполняемых случаев.

Листинг 12.2. Загружает данные, ставит варианты выбора в соответствие спискам и получает входные данные от пользователя. Файл nest_egg_mcs.py, часть 2

```
    # загрузка файлов исходных данных в процентной форме
❶   print("\nПримечание: входные данные должны быть в процентах, \
          а не в десятичной форме!\n")
    try:
        bonds = read_to_list('10-yr_TBond_returns_1926-2013_pct.txt')
        stocks = read_to_list('SP500_returns_1926-2013_pct.txt')
        blend_40_50_10 = read_to_list('S-B-C_blend_1926-2013_pct.txt')
        blend_50_50 = read_to_list('S-B_blend_1926-2013_pct.txt')
        infl_rate = read_to_list('annual_infl_rate_1926-2013_pct.txt')
    except IOError as e:
        print("{}. \nЗавершение программы.".format(e), file=sys.stderr)
        sys.exit(1)

    # получить входные значения от пользователя;
    # применить словарь для аргументов с типом инвестиций
❷   investment_type_args = {'bonds': bonds, 'stocks': stocks,
                            'sb_blend': blend_50_50,
                            'sbc_blend': blend_40_50_10}

❸   # напечатать инструкцию по вводу пользователем данных
    print("   stocks = акции SP500")
    print("    bonds = 10-летние казначейские облигации")
    print(" sb_blend = 50% акции SP500/50% казн.обл.")
    print("sbc_blend = 40% акции SP500/50% казн.обл./10% нал.\n")
    print("Нажмите ENTER для принятия значения по умолчанию, показанного
                                                        в [скобках]. \n")

    # получить входные значения от пользователя
❹   invest_type = default_input("Введите тип инвестиции: (акции, "\
                                "облигации, их сочетание,"\
                                " sbc_blend): \n", 'облигаций').lower()
❺   while invest_type not in investment_type_args:
        invest_type = input("Недопустимая инвестиция. \
```

```
                            Введите тип инвестиции \
                            как перечислено в подсказке: ")
    start_value = default_input("Введите начальное \
                                значение инвестиций: \n", '2000000')
❻   while not start_value.isdigit():
        start_value = input("Недопустимое входное значение! \
                            Введите только целое число: ")
❼   withdrawal = default_input("Введите годовой вывод средств до налогов \
                               (в текущих $): \n", '80000')
    while not withdrawal.isdigit():
        withdrawal = input("Недопустимое входное значение! \
                           Введите только целое число: ")

    min_years = default_input("Введите минимум лет жизни \
                              на пенсии: \n", '18')
    while not min_years.isdigit():
        min_years = input("Недопустимое входное значение! \
                          Введите только целое число: ")

    most_likely_years = default_input("Введите наиболее вероятное \
                                      число лет жизни на пенсии: \n", '25')
    while not most_likely_years.isdigit():
        most_likely_years = input("Недопустимое входное значение! \
                                  Введите только целое число: ")

    max_years = default_input("Введите максимум лет жизни на пенсии: \n", '40')
    while not max_years.isdigit():
        max_years = input("Недопустимое входное значение! \
                          Введите только целое число: ")

    num_cases = default_input("Введите число выполняемых случаев: \n", '50000')
    while not num_cases.isdigit():
        num_cases = input("Недопустимое входное значение! \
                          Введите только целое число: ")
```

После вывода предупреждения о том, что входные значения должны быть в процентах, примените функцию `read_to_list()`, которая загрузит шесть файлов данных ❶. При открытии файлов используйте конструкцию `try` с целью перехвата исключений, связанных с отсутствующими файлами или неправильными именами файлов. Затем обработайте исключения с помощью блока `except`. Если вам требуется освежить свои знания по инструкциям `try` и `except`, обратитесь к *разд. "Обработка исключений при открытии файлов" главы 2.*

Пользователь получит выбор тестируемых инвестиционных инструментов. Для того чтобы дать ему возможность вводить простые имена, используйте словарь, который увяжет имена со списками только что загруженных данных ❷. Позже вы

передадите этот словарь и его ключ, которые будут служить аргументами для функции `montecarlo(investment_type_args[invest_type])`. Прежде чем запрашивать входные значения, напечатайте инструкцию по вводу данных в помощь пользователю ❸.

Далее получите выбранный пользователем тип инвестиции ❹. Примените функцию `default_input()` и перечислите имена вариантов, которы увязаны со списками данных. Установите значение по умолчанию равным `'bonds'` (облигации), для того чтобы увидеть, какие результаты показывает этот якобы "безопасный" вариант выбора. Обязательно прикрепите метод `.lower()` на случай, если пользователь ненароком добавит прописную букву. Другие возможные ошибки ввода проверьте в цикле `while` относительно имен в словаре `investment_type_args`; если вы не найдете там входное значение, то запросите у пользователя правильный ответ ❸.

Продолжайте сбор входных значений и используйте значения по умолчанию, чтобы направить пользователя к разумным входным данным. Например, $ 80 тыс. — это 4% начальной стоимости $ 2 млн; кроме того, 25-летний ожидаемый срок жизни на пенсии является хорошим наиболее вероятным значением для женщин, выходящих на пенсию в 60 лет, максимальное значение 40 позволит им достичь 100 лет, и 50 тыс. случаев должны быстро дать хорошее оценочное значение вероятности разорения.

Для числовых входных значений используйте цикл `while` с целью проверки того, что введенное значение является цифрой, на случай, если пользователь включит в число знак доллара (`$`) или запятые ❻. А для суммы выводимых средств `withdrawal` используйте подсказку с сообщением о том, что требуется ввести сумму в текущих долларах и не беспокоиться об инфляции ❸.

Проверка на наличие других ошибок ввода данных

Листинг 12.3 выполняет проверку на наличие дополнительных ошибок ввода. Порядок минимума лет, наиболее вероятного числа лет и максимума лет на пенсии должен быть логичным, и в программе реализована максимальная продолжительность, равная 99 годам. Предусмотрение возможности долгой жизни на пенсии позволяет оптимистичному пользователю оценить случаи, когда медицина сделает значительные успехи в области антистарения!

Листинг 12.3. Выполняет проверку на наличие ошибок и устанавливает ограничения для ввода числа лет жизни на пенсии. Файл nest_egg_mcs.py, часть 3

```
    # проверить наличие других ошибочных входных данных
❷   if not int(min_years) < int(most_likely_years) < int(max_years) \
        or int(max_years) > 99:
❼       print("\nПроблема с введенными годами.", file=sys.stderr)
        print("Требуется, чтобы мин. < наиболее вероятн. < макс, \
              где макс. <= 99.", file=sys.stderr)
        sys.exit(1)
```

Примените условное выражение, обеспечивающее, чтобы введенный минимум лет был меньше, чем наиболее вероятное число лет, а наиболее вероятное число лет было меньше максимума лет, причем максимум лет не должен превышать 99 ❶. При возникновении неполадки предупредите пользователя ❷, предоставьте некоторые уточняющие инструкции и выйдите из программы.

Определение механизма Монте-Карло

В листинге 12.4 определяется первая часть функции, которая будет выполнять симуляцию Монте-Карло. Программа использует цикл для перебора каждого случая, а введенные числа лет жизни на пенсии будут использоваться для выборки исторических данных. Для списков возвратности и инфляции программа случайно выбирает начальный год или индекс. Число лет на пенсии, назначенное переменной `duration`, берется из треугольного распределения, построенного из данных, введенных пользователем. Если выбрано значение 30, то в этот начальный индекс добавляется 30, создавая конечный индекс. Случайный начальный год определит финансовое состояние пенсионера на всю оставшуюся жизнь! Как говорится, время решает все.

Листинг 12.4. Определяет функцию Монте-Карло и выполняет цикл с перебором всех случаев. Файл nest_egg_mcs.py, часть 4

```
❶ def montecarlo(returns):
      """Выполнить симуляцию Монте-Карло и вернуть стоимость инвестиции
         в конце плана и счетчик банкротства."""
   ❷ case_count = 0
      bankrupt_count = 0
      outcome = []

   ❸ while case_count < int(num_cases):
         investments = int(start_value)
      ❹ start_year = random.randrange(0, len(returns))
      ❺ uration = int(random.triangular(int(min_years), int(max_years),
                                        int(most_likely_years)))
      ❻ end_year = start_year + duration
      ❼ lifespan = [i for i in range(start_year, end_year)]
         bankrupt = 'no'

         # построить временные списки для каждого случая
      ❽ lifespan_returns = []
         lifespan_infl = []
         for i in lifespan:
          ❾ lifespan_returns.append(returns[i % len(returns)])
            lifespan_infl.append(infl_rate[i % len(infl_rate)])
```

Функция `montecarlo()` в качестве аргумента принимает список `returns` ❼. Первым шагом является создание счетчика для отслеживания выполняемого случая ❷. Помните, что использовать фактические даты не требуется; первым годом в списках является не 1926, а индекс 0. Вдобавок создайте счетчик числа случаев, когда деньги были исчерпаны досрочно. Затем создайте пустой список для хранения результатов каждого прогона, т. е. суммы денег, оставшейся в конце прогона.

Начните цикл `while`, который будет перебирать все случаи ❸. Назначьте новой переменной с именем `investments` указанное пользователем начальное значение инвестиций. Так как переменная `investments` будет постоянно меняться, необходимо сберечь исходную входную переменную для реинициализации в каждом случае. И поскольку все введенные пользователем значения поступают в виде символьных цепочек, перед использованием их необходимо конвертировать в целые числа.

Далее задайте переменную `start_year` и выберите произвольное значение из диапазона имеющихся лет ❹. Время, проведенное на пенсии, для этой симулируемой жизни можно получить с помощью метода `triangular()` модуля `random`, который делает выборку из треугольного распределения, определенного пользователем значениями `min_years`, `most_likely_years` и `max_years` ❺. Согласно документации, `triangular()` возвращает случайное число *N* с плавающей точкой, такое, что нижняя величина не превышает *N*, которое, в свою очередь, не превышает верхнюю величину, и с указанной модой между этими границами.

Добавьте переменную `duration` в переменную `start_year` и назначьте результат переменной `end_year` ❻ ❼ ❾. Теперь сформируйте новый список с именем `lifespan`, который захватывает все индексы между начальным и конечным годами ❼. Эти индексы потребуются для соотнесения пенсионного срока с историческими данными. Затем присвойте переменной `bankrupt` значение `'no'`. Банкротство означает, что вы исчерпали все деньги, и позже этот результат досрочно завершит цикл `while` с помощью инструкции `break`.

Используйте два списка для хранения применимых финансовых возвратов и данных по инфляции за выбранную продолжительность жизни `lifespan`. Заполните эти списки с помощью цикла `for`, который использует каждый элемент в `lifespan` в качестве индекса для списков финансовых возвратов и инфляции. Если индекс списка `lifespan` находится вне диапазона по сравнению с другими списками, то примените оператор деления по модулю (`%`) для циклического возврата индексов в начало.

Давайте подробнее рассмотрим фоновые вычисления в этом листинге. Случайно выбранная переменная `start_year` и вычисленная переменная `end_year` определяют способ выборки списков возвратности и инфляции. Выборка представляет собой непрерывный фрагмент финансовой истории и представляет собой один случай. Отбор интервала наугад отличает эту программу от онлайновых калькуляторов, которые отбирают наугад отдельные годы и могут использовать другой год для каждого класса активов и инфляции! Рыночные результаты не являются чистым хаосом; бычьи и медвежьи рынки цикличны, как и инфляционные тренды. Те же самые события, которые приводят к снижению акций, также влияют и на цену об-

лигаций, и на темп инфляции. Выбор лет наугад игнорирует эту взаимозависимость и нарушает известные линии поведения, приводя к нереалистичным результатам.

На рис. 12.7 пенсионер (так называемый случай 1) решил уйти на пенсию в 1965 г. — в начале Великой инфляции — и инвестировать в облигации. Поскольку конечный год происходит до конца списка финансовых возвратов, то пенсионный период хорошо вписывается в список. Возвратность и инфляция отбираются за тот же самый интервал.

Рис. 12.7. График списков облигаций и инфляции, с аннотированным пенсионным сроком начиная с 1965 г.

На графике рис. 12.8 пенсионер, или случай 2, решил уйти на пенсию в 2000 г. Поскольку список заканчивается в 2013 г., то 30-летняя выборка, взятая функцией симуляции Монте-Карло, должна "циклически перенестись в начало" и охватить годы с 1926 по 1941. Это вынуждает пенсионера пережить две рецессии и депрессию.

Рис. 12.8. График списков облигаций и инфляции, с аннотированными сегментами, используемыми в случае 2

Программа должна будет просимулировать сегмент циклического возврата в начало, который вы видите в случае 2, — отсюда использование оператора деления по модулю, который позволяет рассматривать списки как бесконечные циклы.

Симулирование каждого года в заданном случае

Листинг 12.5 продолжает функцию `montecarlo()` и перебирает каждый пенсионный срок для заданного случая, увеличивая или уменьшая стоимость инвестиций на основе возвратности за этот год, вычитая скорректированную на инфляцию сумму вывода средств из инвестиций и проверяя исчерпанность инвестиции. Программа сохраняет в списке окончательную инвестиционную стоимость (представляющую собой сбережения, оставшиеся к моменту смерти) с тем, чтобы в конце можно было рассчитать совокупную вероятность разорения.

Листинг 12.5. Симулирует результаты для каждого года пенсионного срока (в каждом случае). Файл nest_egg_mcs.py, часть 5

```
        # перебрать в цикле каждый год пенсионного срока
        # для каждого прогоняемого случая
    ❶ for index, i in enumerate(lifespan_returns):
        ❷ infl = lifespan_infl[index]

        ❸ # не корректировать на инфляцию в первый год
          if index == 0:
              withdraw_infl_adj = int(withdrawal)
          else:
              withdraw_infl_adj = int(withdraw_infl_adj * (1 + infl))

        ❹ investments -= withdraw_infl_adj
          investments = int(investments * (1 + i))

        ❺ if investments <= 0:
              bankrupt = 'yes'
              break

    ❻ if bankrupt == 'yes':
          outcome.append(0)
          bankrupt_count += 1
      else:
          outcome.append(investments)

  ❼ case_count += 1

❽ return outcome, bankrupt_count
```

Начните цикл `for`, который будет перебирать все годы в каждом случае ❶. Используйте встроенную функцию `enumerate()` на списке `returns` и индекс, который эта

функция производит, для того чтобы из инфляционного списка получить среднее значение инфляции за год ❷. Используйте условное выражение, для того чтобы начать применять инфляцию после первого года ❺. С течением времени она будет медленно увеличивать или уменьшать сумму вывода средств в зависимости от того, в какие времена вы живете: инфляционные либо дефляционные.

Вычтите из переменной `investments` значение вывода средств с поправкой на инфляцию, а затем скорректируйте инвестиции на возвратность за год ❹. Убедитесь, что значение `investments` больше 0. Если это не так, то установите переменную `bankrupt` равной `'yes'` и завершите цикл ❻. В случае банкротства добавьте 0 в список результатов `outcome` ❺. В противном случае цикл продолжается до тех пор, пока не будет достигнута продолжительность пенсионного срока, после чего вы добавите оставшуюся стоимость инвестиций в `outcome`.

Человеческая жизнь только что закончилась: от 30 до 40 лет каникул, внуков, игр в бинго и болезней промелькнули быстрее чем за секунду. И прежде чем перейти к циклической обработке следующей жизни, продвиньте счетчик случаев вперед ❼. Завершите работу функции, вернув переменные `outcome` и `bankrupt_count` ❽.

Расчет вероятности разорения

Листинг 12.6 определяет функцию, которая вычисляет вероятность исчерпания денег, также именуемую "вероятностью разорения". Если вы не склонны к риску либо не желаете оставить значительную сумму своим наследникам, то вы, вероятно, хотите, чтобы это число было меньше 10%. Те, у кого повышенный аппетит к риску, останутся довольными уровнем до 20% и более. В конце концов, вы же не можете взять эту сумму с собой!

Листинг 12.6. Вычисляет и выводит на экран "вероятность разорения" и другие статистические данные. Файл nest_egg_mcs.py, часть 6

```
❶ def bankrupt_prob(outcome, bankrupt_count):
      """Вычислить и вернуть шанс исчерпать все деньги,
         а также другую статистику."""
   ❷ total = len(outcome)
   ❸ odds = round(100 * bankrupt_count / total, 1)

   ❹ print("\nТип инвестиций: {}".format(invest_type))
     print("Начальная стоимость: ${:,}".format(int(start_value)))
     print("Годовой вывод средств: ${:,}".format(int(withdrawal)))
     print("Годы на пенсии (мин.-наиболее вероятн.-макс.): {}-{}-{}"
           .format(min_years, most_likely_years, max_years))
     print("Число прогонов: {:,}\n".format(len(outcome)))
     print("Шансы исчерпания денег: {}%\n".format(odds))
     print("Средний результат: ${:,}".format(int(sum(outcome) / total)))
     print("Минимальный результат: ${:,}"
           .format(min(i for i in outcome)))
```

```
    print("Максимальный результат: ${:,}"
          .format(max(i for i in outcome)))

❺ return odds
```

Определите функцию с именем `bankrupt_prob()`, которая в качестве аргументов принимает список `outcome` и переменную `bankrupt_count`, возвращенную из функции `montecarlo()` ❶. Назначьте длину списка результатов `outcome` переменной с именем `total` ❷. Затем вычислите вероятность исчерпания денег в процентах, округлив до одного десятичного знака, разделив число случаев банкротства на суммарное число случаев ❸.

Теперь покажите на экране входные параметры и результаты симуляции ❹. Вы видели пример этой текстовой распечатки в *разд. "Качественный способ представления результатов" ранее в этой главе*. Завершите функцию, вернув переменную `odds` ❺.

Определение и вызов функции *main()*

В листинге 12.7 определена функция `main()`, которая вызывает функции `montecarlo()` и `basket_count()` и создает изображение столбчатого графика. Результаты для различных случаев могут иметь большую дисперсию — в одних случаях вы разоряетесь, а в других вы — мультимиллионер! Если печатная статистика это не выявляет, то столбчатый график, безусловно, это делает.

Листинг 12.7. Определяет и вызывает функцию `main()`. Файл nest_egg_mcs.py, часть 7

```
❶ def main():
      """Вызвать функции симуляции Монте-Карло и определения
         банкротства и изобразить столбчатый график результатов."""
   ❷ outcome, bankrupt_count =
         montecarlo(investment_type_args[invest_type])
      odds = bankrupt_prob(outcome, bankrupt_count)

   ❸ plotdata = outcome[:3000]     # вывести на график только
                                   # для первых 3000 прогонов

   ❹ plt.figure(
            'Результат в каждом случае (показаны первые {} прогонов)'
            .format(len(plotdata)),
            figsize=(16, 5))  # размер - это ширина и высота в дюймах
   ❺ index = [i + 1 for i in range(len(plotdata))]
   ❻ plt.bar(index, plotdata, color='black')
      plt.xlabel('Симулируемые жизни', fontsize=18)
      plt.ylabel('Остаток $', fontsize=18)
   ❼ plt.ticklabel_format(style='plain', axis='y')
```

```
    ❽ ax = plt.gca()
      ax.get_yaxis().set_major_formatter(plt.FuncFormatter(lambda x,
                                                           loc: "{:,}"
                                                          .format(int(x))))
      plt.title('Вероятность исчерпания денег = {}%'.format(odds),
                fontsize=20, color='red')
    ❾ plt.show()

  # выполнить программу
❿ if __name__ == '__main__':
      main()
```

Определите функцию `main()`, которая не требует аргументов ❶, и сразу же вызовите функцию `montecarlo()`, которая получит список результатов `outcome`, и функцию `bankrupt_count()` ❷. Используйте словарь, в котором имена инвестиций увязаны с данными, для получения списков, созданных в листинге 12.2. Функции `montecarlo()` вы передаете аргумент, который является именем словаря, `investment_type_args`, а в качестве ключа используется введенное пользователем значение типа инвестиции `invest_type`. Передав возвращенные значения в функцию `bankrupt_prob()`, вы получите шансы исчерпания денег.

Назначьте новой переменной `plotdata` первые 3000 элементов списка `outcome` ❸. Столбчатый график может вместить гораздо больше элементов, но их вывод на экран будет медленным и ненужным. Поскольку результаты являются стохастическими, то при демонстрации большего количества случаев дополнительной информации получится немного.

Теперь вы примените библиотеку `matplotlib` для создания и вывода на экран столбчатого графика. Начните с построения объекта `figure` ❹. Текстовый элемент будет заголовком нового окна, а параметр `figsize` — шириной и высотой окна в дюймах. Его можно прошкалировать, добавив аргумент точек на дюйм (dots per inch), к примеру, `dpi=200`.

Далее примените операцию включения в список, которая построит индексы, начиная с 1 для первого года, на основе длины списка `plotdata` ❺. Расположение каждого вертикального столбика на оси *x* будет определяться индексом, а высота каждого столбика будет соответствовать элементу `plotdata`, который представляет деньги, оставшиеся в конце каждой симулируемой продолжительности жизни. Передайте их методу `plt.bar()` и сделайте цвет столбиков черным ❻. Обратите внимание, что существуют дополнительные параметры изображения столбиков, такие как изменение цвета контура столбика (`edgecolor='black'`) или его толщины (`linewidth=0`).

Укажите надписи для осей *x* и *y* и установите размер шрифта, равным 18. Результаты могут достигать миллионных значений, и когда библиотека `matplotlib` аннотирует ось *y*, она по умолчанию будет использовать научную форму записи. Для переопределения формы записи вызовите метод `ticklabel_format()` и установите стиль оси *y* равной `'plain'` (простой) ❼. Он уладит вопрос с научной формой запи-

си, но при этом будут отсутствовать разделители тысяч, что сделает числа трудно различимыми. Для исправления этой ситуации сначала получите текущие оси с помощью `plt.gca()` ❽. Затем в следующей строке получите ось *y* и примените методы `set_major_formatter()` и `Func_Formatter()`, а также лямбда-функцию с целью применения принятого в Python способа форматирования символьных цепочек для разделителей в виде запятой.

В названии графика большим шрифтом, выделенным привлекательным красным цветом, покажите вероятность исчерпания денег — зафиксированную в переменной `odds`. Затем с помощью `plt.show()` нарисуйте график на экране ❾. Вернувшись в глобальное пространство, закончите кодом, который позволяет импортировать программу в виде модуля либо запускать ее в автономном режиме ❿.

Использование симулятора

Программа nest_egg_mcs.py значительно упрощает сложный мир пенсионного планирования, но не таите обиды. Простые модели повышают ценность, бросая вызов допущениям, повышая осведомленность и кристаллизуя вопросы. В деталях пенсионного планирования — или любой сложной проблемы — легко потеряться, поэтому лучше начать рекогносцировку с рельефа местности.

Давайте рассмотрим пример, в котором за основу берется начальная стоимость $ 2 млн, "безопасный и обеспеченный" портфель облигаций, 4-процентная ставка вывода средств (равная $ 80 тыс. в год), 29–30–31-летний пенсионный диапазон и 50 тыс. случаев. Если вы выполните этот скрипт, то получите результаты, аналогичные показанным на рис. 12.9. Вы исчерпаете все деньги почти в половине случаев! Из-за относительно низкой возвратности облигаций они не способны идти в ногу с инфляцией — помните, не стоит слепо применять правило 4 процентов, потому что важное значение имеет то, как распределены ваши финансовые средства среди портфельных активов.

Рис. 12.9. Столбчатый график, созданный библиотекой `matplotlib`, изображающий симулирование Монте-Карло портфеля, состоящего только из облигаций

Обратите внимание, что вывод $ 80 тыс. осуществляется до налогообложения. Исходя из 25-процентной эффективной налоговой ставки, в результате останется чистый доход в размере лишь $ 60 тыс. По данным исследовательского центра Pew, средний располагаемый (после уплаты налогов) доход американского среднего класса в настоящее время составляет $60 884, поэтому вы вряд ли живете на широкую ногу, несмотря на то что являетесь миллионером. Если вы хотите иметь $ 80 тыс. в располагаемом доходе, то должны разделить эту сумму на 1 минус эффективная налоговая ставка; в данном случае это $ 80 000/(1 – 0,25) = $ 106 667. А эта сумма требует снятия чуть более 5% в год, приводя к 20–70-процентным шансам разориться, в зависимости от типа инвестиций!

В табл. 12.1 представлены результаты предыдущего скрипта при варьировании типа активов и ставки вывода средств. Результаты, которые обычно считаются безопасными, выделены серым цветом. Если вы избегаете портфеля, состоящего исключительно из облигаций, то правило 4 процентов хорошо соблюдается. Выше 4% наилучшие шансы — из рассмотренных вариантов — снизить вероятность разорения, что дает присущий акциям потенциал роста, причем с меньшим риском, чем считает большинство людей. Вот почему финансовые консультанты рекомендуют включать в свой пенсионный портфель здоровую дозу акций.

Таблица 12.1. *Вероятность разорения в зависимости от активов и процента выводимых средств для 30-летнего пенсионного срока*

Тип актива	Годовой процент выводимых средств (до налогообложения)			
	3%	4%	5%	6%
10-летние казначейские облигации	0,135	0,479	0,650	0,876
Акции S&P 500	0	0,069	0,216	0,365
Сочетание 50/50	0	0,079	0,264	0,466
Сочетание 40/50/10	0	0,089	0,361	0,591

Финансовые консультанты также советуют не переусердствовать в первые годы выхода на пенсию. Несколько круизов большой семьи, шикарный новый дом или дорогое новое хобби могут столкнуть вас с обрыва в более поздние годы. Для того чтобы исследовать этот вопрос, скопируйте nest_egg_mcs.py и назовите копию nest_egg_mcs_1st_5yrs.py; скорректируйте код, как описано в листингах 12.8–12.10.

Листинг 12.8. Разбивает вводимую пользователем сумму выводимых средств на две части. Файл nest_egg_mcs_1st_5yrs.py, часть 1

```
start_value = default_input("Введите исходную стоимость \
                            инвестиций: \n", '2000000')
while not start_value.isdigit():
    start_value = input("Недопустимое значение! Введите целое число: ")

❶ withdrawal_1 = default_input("Введите годовой вывод средств до налогов \
                               для первых 5 лет (текущие $): \n", '100000')
```

```
while not withdrawal_1.isdigit():
    withdrawal_1 = input("Недопустимое значение! Введите целое число: ")

❷ withdrawal_2 = default_input("Введите годовой вывод средств до налогов \
                               для остатка срока (текущие $): \n", '80000')
while not withdrawal_2.isdigit():
    withdrawal_2 = input("Недопустимое значение! Введите целое число: ")

min_years = default_input("Введите минимум лет на пенсии: \n", '18')
```

В разделе ввода данных пользователем замените исходную переменную выводимых средств `withdrawal` двумя переменными выводимых средств и отредактируйте подсказку, для первой запросив сумму в течение первых пяти лет пенсионного срока ❶ и для второй сумму в течение оставшейся части ❷. Установите значения по умолчанию, тем самым показывая, что пользователь ожидает более высокий вывод средств в течение первых пяти лет. Включите циклы `while`, которые проверяют вводимые пользователем значения.

В функции `montecarlo()` измените код, который корректирует сумму выводимых средств на инфляцию.

Листинг 12.9. Корректирует две переменные выводимых средств на инфляцию и определяет, какую из них следует использовать. Файл nest_egg_mcs_1st_5yrs.py, часть 2

```
        # не корректировать на инфляцию первые пять лет
        if index == 0:
         ❶ withdraw_infl_adj_1 = int(withdrawal_1)
         ❷ withdraw_infl_adj_2 = int(withdrawal_2)
        else:
         ❸ withdraw_infl_adj_1 = int(withdraw_infl_adj_1 * (1 + infl))
         ❹ withdraw_infl_adj_2 = int(withdraw_infl_adj_2 * (1 + infl))

    ❺ if index < 5:
         ❻ withdraw_infl_adj = withdraw_infl_adj_1
        else:
            withdraw_infl_adj = withdraw_infl_adj_2

        investments -= withdraw_infl_adj
        investments = int(investments * (1 + i))
```

Установите скорректированную на инфляцию сумму выводимых средств, равную введенному значению только за первый год ❶ ❷. В противном случае скорректируйте оба значения на инфляцию ❸ ❹. Благодаря этому вторая сумма выводимых средств будет "готова", когда вы перейдете на нее через пять лет.

Примените условное выражение для того, чтобы указать, когда следует использовать каждую скорректированную на инфляцию сумму выводимых средств ❺. Назначьте их существующей переменной `withdraw_infl_adj` . Это позволит вам избежать внесения дополнительных изменений в программный код ❻.

Наконец, обновите функцию `bankrupt_prob()`, где печатается статистика, с тем чтобы включить новые значения выводимых средств, как показано в листинге 12.10. Эти изменения должны заменить старые инструкции печати выводимых средств.

Листинг 12.10. Печатает суммы выводимых средств для двух периодов вывода. Файл nest_egg_mcs_1st_5yrs.py, часть 3

```
print("Годовой вывод средст первые 5 лет: ${:,}"
      .format(int(withdrawal_1)))
print("Годовой вывод средств после 5 лет: ${:,}"
      .format(int(withdrawal_2)))
```

Теперь можно провести новый эксперимент (табл. 12.2).

Таблица 12.2. *Вероятность разорения в зависимости от типа активов и нескольких значений процентов выводимых средств для 30-летнего пенсионного срока*

Тип актива	Годовой процент выводимых средств (до налогообложения)			
	4%/4%	5%/4%	6%/4%	7%/4%
10-летние казначейские облигации	100,479	0,499	0,509	0,571
Акции S&P 500	0,069	0,091	0,116	0,194
Сочетание 50/50	0,079	0,115	0,146	0,218
Сочетание 40/50/10	0,089	0,159	0,216	0,264

Безопасные результаты в табл. 12.2 заштрихованы серым цветом, и первый столбец в качестве контроля повторяет постоянные 4-процентные результаты. При достаточном количестве акций в вашем портфеле вы можете терпимо понести несколько ранних расходов, и по этой причине некоторые консультанты заменяют правило 4 процентов правилом 4,5 или 5 процентов. Но если вы выйдете на пенсию рано — скажем, в возрасте от 55 до 60 лет, то ваш риск разориться будет больше независимо от того, придется ли вам жить во времена с какими-либо высокими расходами или нет.

Если вы выполните симулятор для сочетания акций и облигаций 50/50, используя разные пенсионные сроки, то должны получить результаты, аналогичные приведенным в табл. 12.3. Только один результат (заштрихованный серым цветом) имеет вероятность разорения ниже 10%.

Выполнение подобных симуляций заставляет людей принимать трудные решения и формировать реалистичные планы на большой отрезок своей жизни. В то время как

симуляция "продает" активы каждый год, финансируя жизнь на пенсии, более оптимальным максимально приближенным к реальной жизни решением является стратегия спасительной соломинки, где в первую очередь расходуются проценты и дивиденды и поддерживается резерв наличных денежных средств во избежание необходимости продавать активы во время рыночных минимумов. Исходя из того, что вы будете оставаться дисциплинированным как инвестор, эта стратегия позволит вам растянуть свои выводимые средства немного за пределы того срока, который симулятор рассчитывает как безопасный.

Таблица 12.3. *Вероятность разорения по сравнению с продолжительностью пенсионного срока для 4%-го вывода средств (50/50-е сочетание акций и облигаций)*

Годы на пенсии	Процентный вывод средств
30	0,079
35	0,103
40	0,194
45	0,216

Резюме

В этой главе вы написали калькулятор пенсионных накоплений на основе симуляции Монте-Карло, который реалистично использует выборки из исторических финансовых данных. Вы также применили библиотеку `matplotlib` для обеспечения альтернативного способа взглянуть на результаты калькулятора. Хотя используемый пример можно было бы смоделировать детерминированно, если сюда добавить больше случайных величин, таких как будущие налоговые ставки, платежи по социальному обеспечению и расходы на здравоохранение, то симуляция Монте-Карло быстро становится единственным практическим подходом к моделированию пенсионных стратегий.

Дальнейшее чтение

Книга Бенджамина Грэхема "Разумный инвестор: полная книга по инвестированию. Пересмотренное издание" (Graham B. The intelligent investor: the definitive book on value investing. Revised Edition HarperBusiness, 2006) многими, в том числе инвестором-миллиардером Уорреном Баффетом, считается величайшей книгой по инвестированию из когда-либо написанных.

Книга Нассима Николаса Талеба "Одураченные случайностью: скрытая роль случая в жизни и на рынках. Пересмотренное издание" (Taleb N. N. Fooled by randomness: the hidden role of chance in life and in the markets. Revised Edition Random House Trade Paperbacks, 2005) представляет собой "привлекательный взгляд на историю и причины нашего пристрастия к самообману, когда речь заходит о статистике".

В ней обсуждается вопрос применения симулирования Монте-Карло в финансовом анализе.

Книга Нассима Николаса Талеба "Черный лебедь: влияние крайне невозможного. 2-е издание" (Taleb N. N. The black swan: the impact of the highly impossible. 2nd Edition. Random House Trade Paperbacks, 2010) пронесет вас "восхитительным галопом по истории, экономике и слабостям человеческой природы" и предложит вам поучаствовать в обсуждении вопроса применения симулирования Монте-Карло в финансах.

Обзор правила 4 процентов можно найти по адресу **https://www.investopedia.com/terms/f/four-percent-rule.asp**.

Возможные исключения из правила 4 процентов обсуждаются на веб-сайте **https://www.cnbc.com/2015/04/21/the-4-percent-rule-no-longer-applies-for-most-retirees.html**.

Исторические финансовые данные можно найти на приведенных ниже веб-сайтах:

- **http://pages.stern.nyu.edu/~adamodar/New_Home_Page/datafile/histretSP.html**;
- **http://www.econ.yale.edu/~shiller/data.htm**;
- **http://www.moneychimp.com/features/market_cagr.htm**;
- **http://www.usinflationcalculator.com/inflation/historical-inflation-rates/**;
- **https://inflationdata.com/Inflation/Inflation_Rate/HistoricalInflation.aspx**.

Сложные проекты

Станьте дипломированным сертифицированным финансовым аналитиком (Certified Financial Analyst, CFA)[3], выполнив следующие ниже сложные проекты.

Ясная картинка стоит тысячи слов

Представьте, что вы — сертифицированный финансовый аналитик и ваш потенциальный клиент, богатый техасский нефтепромышленник, не понимает ваших результатов симуляции Монте-Карло по его портфелю в размере 10 млн долларов. "Чертовщина какая-то! И какая же дурацкая причина может заставить меня разориться в одном случае и стоить мне 80 млн в другом?"

Разложите ему все по полочкам, отредактировав программу nest_egg_mcs.py так, чтобы она обрабатывала одиночные 30-летние случаи с использованием исторических интервалов, которые приведут к плохим и хорошим результатам, например начиная с начала Великой депрессии относительно конца Второй мировой войны, но только рассматривая крайние случаи. Для каждого года в каждом случае напечатайте год, уровень возвратности, темп инфляции и результат. Будет еще лучше, если вы отредактируете изображение столбчатого графика так, чтобы ради убеди-

[3] Вообще-то нет.

тельности визуального объяснения в нем вместо результата каждого случая использовался результат каждого года.

Перемешать и увязать

Отредактируйте программу nest_egg_mcs.py так, чтобы пользователь мог генерировать свое собственное сочетание инвестиций. Используйте текстовые файлы с акциями S&P 500, 10-летними казначейскими облигациями и трехмесячными казначейскими векселями, которые я предоставил в начале данной главы, и добавьте все, что вам нравится, например акции с небольшой капитализацией, международные акции или даже золото. Просто помните, что временные интервалы в каждом файле или списке должны быть одинаковыми.

Попросите пользователя выбрать типы инвестиций и процент от каждого. Обеспечьте, чтобы вводимые им значения в сумме составляли 100%. Затем создайте смешанный список, взвешивая и складывая финансовые возвраты за каждый год. Завершите программу, показав вверху столбчатого графика типы инвестиций и проценты.

Масть пошла, а деньги кончились

Отредактируйте программу nest_egg_mcs.py так, чтобы рассчитать вероятность столкнуться с Великой депрессией (1939–1949) или Великой рецессией (2007–2009) во время 30-летнего пенсионного срока. Вам нужно будет выявить, какие индексные числа в списках финансовых возвратов соответствуют этим событиям, а затем подсчитать количество их появлений в течение любого числа случаев. Покажите результаты в интерактивной оболочке.

Все что есть

В качестве другого способа просмотра результатов скопируйте и отредактируйте программу nest_egg_mcs.py так, чтобы столбчатый график показывал все результаты в отсортированном виде от наименьшего элемента к наибольшему.

13

СИМУЛЯЦИЯ ИНОПЛАНЕТНОГО ВУЛКАНА

Быстро! Назовите самое вулканически активное тело в Солнечной системе! Если вы подумали о Земле, то вы ошибаетесь — это Ио, один из четырех галилеевых спутников Юпитера.

Первые свидетельства вулканизма на Ио появились в 1979 г., когда "Вояджер-1" совершил свой знаменитый облет системы Юпитера. Но впечатляющие фотографии, которые он записал, не были настоящим сюрпризом. Астрофизик Стэн Пил (Stan Peale) и два его соавтора уже опубликовали этот результат, основанный на моделях внутренности Ио.

Компьютерное моделирование представляет собой мощный инструмент, предназначенный для понимания природы и составления предсказаний. Вот его общий рабочий поток:

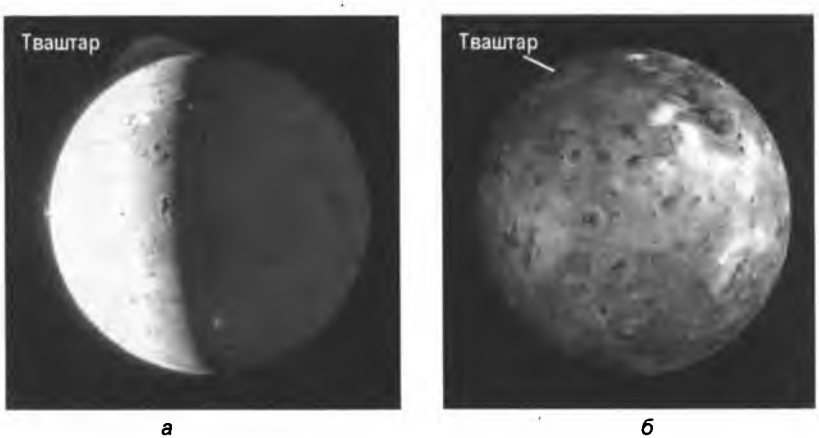
1. Собрать данные.
2. Выполнить анализ, интерпретацию и интеграцию данных.
3. Сгенерировать численные уравнения, которые объясняют данные.
4. Сконструировать компьютерную модель, которая наилучшим образом "укладывается" в данные.
5. Применить модель для составления предсказаний и разведывания диапазонов ошибок.

Применение компьютерного моделирования имеет далеко идущие последствия и включает такие области, как управление дикой природой, метеорологическое прогнозирование, климатические предсказания, добыча углеводородов и моделирование черных дыр.

В этой главе вы будете использовать пакет Python под названием `pygame` — обычно используемый для создания игр — с целью симуляции одного из вулканов Ио. Вы также будете экспериментировать с разными типами выбросов (извергаемых частиц) и сравнивать их симулируемое поведение с фотографией гигантского шлейфа (плюма) вулкана Тваштар на Ио.

Проект 21: шлейфы на Ио

За вулканизм на Ио ответственно приливное нагревание. Поскольку эллиптическая орбита Ио переносит его через гравитационные поля Юпитера и его сестринских лун, он испытывает вариации в приливном притяжении. Его поверхность изгибается вверх и вниз на целых 100 м, что приводит к значительному нагреву и плавлению в его внутренностях. Горячая магма мигрирует на поверхность и образует большие лавовые озера, из которых дегазирующая сера (S_2) и диоксид серы (SO_2) распыляются в небо со скоростью 1 км/с. Из-за низкой гравитации и отсутствия атмосферы на Ио эти шлейфы газа могут достигать высоты в сотни километров (см. шлейф вулкана Тваштар на рис. 13.1, *а*).



а *б*

Рис. 13.1. Ио с 330-километровым шлейфом вулкана Тваштар наверху и более коротким шлейфом вулкана Прометей в положении "9 часов" (*а*); Ио с вулканическими кольцевыми отложениями (снимки NASA) (*б*)

Приятная на вид зонтичная форма шлейфов порождается по мере того, как газ и пыль выстреливают вверх и затем падают назад в разные стороны. Результирующие поверхностные отложения образуют концентрические кольца красного, зеленого, черного и желтого цвета. Если бы рис. 13.1, *б* был цветным, то он выглядел бы немного как заплесневелая пицца Пепперони.

Кусочек пакета *pygame*

Пакет `pygame` представляет собой кросс-платформенный набор модулей Python, который обычно используется для программирования двухмерных аркадных видеоигр. Он поддерживает графику, анимацию, звуковые эффекты, музыку и несколько

устройств ввода, таких как клавиатура и мышь. Изучение пакета pygame — это больше, чем просто интересный способ изучения программирования. Аркадные игры стали популярными благодаря распространению смартфонов и планшетов, а мобильные игры теперь зарабатывают почти столько же, сколько консольные и компьютерные игры, вместе взятые.

Пакет pygame использует простую библиотеку DirectMedia (SDL), которая представляет собой прикладной программный интерфейс, или API. API, — это многоразовые библиотеки кода, которые во многом упрощают обработку графики, позволяя сосредотачиваться на игровом дизайне при использовании языка высокого уровня, такого как Python. API Microsoft DirectX применяется для создания игр и мультимедийных приложений для платформы Windows. Для работы на нескольких платформах предназначены две библиотеки с открытым исходным кодом: SDL в основном для работы в двух измерениях и OpenGL (Open Graphics Library) для трехмерных приложений. Как уже упоминалось, вы будете работать с библиотекой SDL, которая официально поддерживает Windows, macOS, Linux, iOS и Android.

Пакет pygame также использует объектно-ориентированное программирование (ООП). Если вы не знакомы с ООП или вам требуется освежить свои знания, обратитесь к *разд. "Краткое введение в объектно-ориентированное программирование" главы 11.* Вдобавок издания с вводными курсами по Python часто включают раздел о pygame, и об этом пакете написаны целые книги (несколько примеров *см. в разд. "Дальнейшее чтение" далее в этой главе*).

Прежде чем продолжить чтение, вам нужно установить pygame в вашу систему. Инструкции по установке бесплатной копии на предпочитаемой платформе доступны по адресу **http://pygame.org/wiki/GettingStarted#Pygame%20Installation**.

Видеоуроки по установке пакета pygame также доступны онлайн. Для обеспечения соответствия видеоролика вашей ситуации обязательно проверьте дату видео, обсуждаемую платформу и используемые версии pygame и Python. Дополнительные инструкции для пользователей Mac с более старыми установленными версиями Python можно найти по адресу **http://brysonpayne.com/2015/01/10/setting-up-pygame-on-a-mac/**.

Цель

Применить пакет pygame для создания двухмерной гравитационной симуляции вулканического шлейфа вулкана Тваштар на Ио. Откалибровать размеры шлейфа, используя снимок NASA. Использовать несколько типов частиц в шлейфе, проследить траектории полета частиц и позволить вулкану извергаться автоматически до тех пор, пока извержение не прекратится.

Стратегия

Построение всесторонней, полной физической симуляции одного из шлейфов Ио лучше всего выполнять с помощью суперкомпьютера. Поскольку у вас, вероятно, нет никакого и ваша цель — сделать классное изображение на основе pygame, вы

намерены обхитрить судьбу путем обратного инженерного анализа параметров, нужных вам для того, чтобы вписать SO_2 в шлейф вулкана Тваштар. Помните, хитрость — это дар, который люди преподносят себе сами; это то, что отличает нас от животных — за исключением гепардов!

Поскольку состав шлейфов Ио уже известен, вы намерены откалибровать свое гравитационное поле по газам SO_2 и S_2 (серы), которые как раз имеют одинаковую атомную массу. Когда траектории полета этих частиц совпадут с размерами шлейфа вулкана Тваштар на снимке NASA, вы прошкалируете скорость других выброшенных частиц, основываясь на разнице атомных масс новой частицы и SO_2, для того чтобы увидеть, как тип частиц влияет на размеры шлейфа. Более легкие частицы будут выбрасываться выше, и наоборот.

Использование эскиза игры в целях планирования

Предлагаю вам начинать любой проект `pygame` с эскиза того, как должна выглядеть игра и как должно развиваться действие. Бывает, что даже самые простые аркадные игры становятся сложными, а эскиз поможет вам справиться со сложностью. Среди многого из того, что вы должны учитывать в типичной игре, в первых рядах находятся действия игрока, учет очков, сообщения и инструкции, игровые сущности и их взаимодействия (такие как столкновения, звуковые эффекты и музыка), а также условия окончания игры.

Составление эскиза игры — или в данном случае симуляции — лучше всего делать на доске, реальной либо цифровой. Мой макет симулятора вулкана Ио показан на рис. 13.2.

Эскиз на рис. 13.2 содержит рекомендации и основные линии поведения симулятора вулкана.

- **Прямое взаимодействие с игроками отсутствует.** Вы будете управлять симуляцией не с помощью мыши или клавиатуры, а редактируя код Python.
- **Фоном будет снимок шлейфа, принадлежащий NASA.** Для калибровки симуляции по частицам SO_2/S_2 вам понадобится фон фактического шлейфа вулкана Тваштар.
- **Запущенная точка движется по оси.** Частицы должны выбрасываться из центрального основания снимка шлейфа и распыляться в угловом диапазоне, а не просто прямо вверх.
- **Частицы выбираются случайно.** Программа выберет тип частицы наугад. Каждая частица будет иметь уникальный цвет, который отличает ее от других.
- **Траектория полета частиц должна быть видимой и постоянной.** Полет каждой частицы должен регистрироваться в виде линии, которая остается видимой на протяжении всей симуляции, и цвет линии должен соответствовать цвету частицы.
- **Легенда с цветовой кодировкой перечисляет типы частиц.** Программа должна разместить легенду с именами частиц в левом верхнем углу экрана. Шрифто-

вые цвета должны соответствовать цветам частиц, а легенда должна отображаться поверх путей частиц с тем, чтобы она всегда была видна.

- **Движение частиц должно останавливаться на уровне, на котором частицы SO_2 пересекают поверхность Ио.** Симуляция настроена на поведение SO_2, поэтому падающие частицы должны остановиться в соответствующем для шлейфа SO_2 месте.
- **Звуковые эффекты отсутствуют.** В космосе вашего крика никто не услышит.

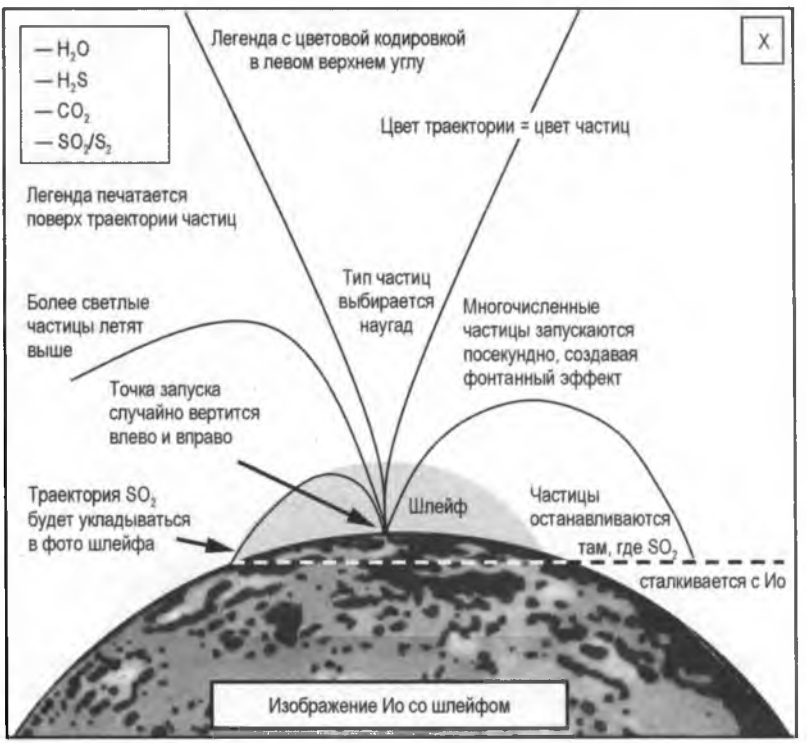


Рис. 13.2. Игровой эскиз симулятора вулкана на Ио

После того как вы закончите свой набросок, можете начать выбирать из него части и перечислять их в логическом порядке; эта процедура разбивает план на ряд управляемых шагов. Например, вам нужно будет найти и подготовить надлежащий фоновой снимок, решить, какие частицы вы хотите просимулировать, и посмотреть их атомные массы, найти точку запуска, откалибровать поведение SO_2 по снимку шлейфа и т. д. Вы по-прежнему пишете псевдокод, но эскиз игры делает этот процесс гораздо приятнее!

Планирование класса частиц

Поскольку эта симуляция основана на частицах, разумно иметь класс частиц `Particle` в стиле ООП, который будет служить основой для нескольких типов частиц. Указанный класс должен поддерживать случайную генерацию типов частиц, а

константы и другие атрибуты, общие для всех частиц, могут храниться как атрибуты класса. Это атрибуты, которые получают значения на том же уровне отступа, что и методы. Класс частиц также должен содержать методы, позволяющие выбрасывать экземпляры класса, воздействовать на них силой гравитации, делать их видимыми и уничтожать, когда они выходят за пределы симуляции.

Используемые в классе атрибуты и методы показаны соответственно в табл. 13.1 и 13.2. Атрибуты класса, т. е. общие для всех экземпляров класса, выделены курсивом; в противном случае показаны атрибуты экземпляра.

Я объясню все эти атрибуты и методы подробнее в следующем разделе.

Таблица 13.1. *Атрибуты класса* `Particle`

Атрибуты	Описание атрибута
Gases_colors	Словарь доступных типов частиц и их цветов
VENT_LOCATION_XY	Координаты *x* и *y* расположения устья вулкана Тваштар на снимке
IO_SURFACE_Y	Значение Y поверхности Ио на границе шлейфа SO_2 на снимке
VELOCITY_SO2	Скорость (пикселов на кадр) частицы SO_2
GRAVITY	Ускорение гравитации в пикселах на кадр
vel_scalar	Словарь соотношений атомных масс SO_2/частица
screen	Экран игры
background	Снимок шлейфа вулкана Тваштар, принадлежащий NASA
image	Квадратная поверхность pygame, представляющая частицу
rect	Прямоугольный объект, используемый для получения размеров поверхности
gas	Тип отдельной частицы (SO_2, CO_2 и др.)
color	Цвет типа отдельной частицы
vel	Скорость частицы относительно скорости SO_2
x	Местоположение *x* частицы
y	Местоположение *y* частицы
dx	Дельта *x* частицы
dy	Дельта *y* частицы

Таблица 13.2. *Методы класса* `Particle`

Метод	Описание метода
`__init__()`	Инициализирует и настраивает параметры для произвольно отбираемого типа частиц
`vector()`	Случайно отбирает ориентацию выброса и вычисляет вектор движения (*dx* и *dy*)
`update()`	Корректирует траекторию движения частиц на гравитацию, рисует траекторию позади частицы и уничтожает частицы, которые выходят за пределы симуляции

Код

Код tvashtar.py будет генерировать основанную на pygame симуляцию шлейфа вулкана Тваштар на Ио. Вам также понадобится фоновый снимок, tvashtar_plume.gif. Скачайте оба файла с **https://www.nostarch.com/impracticalpython/** и сохраните их в одной папке.

Импортирование модулей, инициализирование модуля *pygame* и определение цветов

Начните с нескольких шагов настройки, таких как выбор цветов (листинг 13.1).

Листинг 13.1. Импортирует модули, инициализирует pygame и определяет цветовую таблицу. Файл tvashtar.py, часть 1

```
❶ import sys
  import math
  import random
  import pygame as pg

❷ pg.init()    # инициализировать pygame

❸ # определить цветовую таблицу
  BLACK = (0, 0, 0)
  WHITE = (255, 255, 255)
  LT_GRAY = (180, 180, 180)
  GRAY = (120, 120, 120)
  DK_GRAY = (80, 80, 80)
```

Начните с нескольких уже знакомых инструкций импорта и одной для pygame ❶. Далее вызовите функцию pygame.init(). Она инициализирует пакет pygame и запускает все базовые части, которые позволяют ему использовать звуки, проверять вводимые с клавиатуры данные, выполнять графические построения и т. д. ❷. Обратите внимание, что pygame можно инициализировать из нескольких мест, таких как первая строка в функции main():

```
def main():
    pg.init()
```

либо в конце программы, когда main() вызывается в автономном режиме:

```
if __name__ == "__main__":
    pg.init()
    main()
```

Остановитесь и объявите несколько переменных цвета с помощью цветовой модели RGB ❸. Указанная модель смешивает красный, зеленый и синий цвета, где каждый цвет состоит из значений от 0 до 255. Если вы проведете онлайновый поиск по за-

просу "цветовые коды RGB" (RGB color codes), то сможете найти числовые коды для миллионов цветов. Но поскольку снимок NASA, который вы будете калибровать, представлен в оттенках серого, придерживайтесь черного, белого и оттенков серого. Определив эту таблицу, вы получаете возможность просто вводить имя всякий раз, когда `pygame` нуждается в цвете, который будет определен позже.

Определение класса частиц *Particle*

В листинге 13.2 определяются класс `Particle` и его метод-инициализатор. Вы будете использовать его для инстанцирования объекта-частицы. Ключевые атрибуты частицы, такие как тип, скорость, цвет и т. д., устанавливаются с помощью метода-инициализатора.

Листинг 13.2. Определяет класс `Particle` и его метод-инициализатор. Файл tvashtar.py, часть 2

```
❶ class Particle(pg.sprite.Sprite):
      """Строит извергаемые частицы для симуляции вулкана."""

   ❷ gases_colors = {'SO2': LT_GRAY, 'CO2': GRAY, 'H2S': DK_GRAY,
                     'H2O': WHITE}

   ❸ VENT_LOCATION_XY = (320, 300)
     IO_SURFACE_Y = 308
     GRAVITY = 0.5        # пикселов на кадр;
                          # добавляется в dy каждую итерацию игры
     VELOCITY_SO2 = 8     # пикселов на кадр

     # скаляры (атомная масса SO2/атомная масса частиц),
     # используемые для скорости
   ❹ vel_scalar = {'SO2': 1, 'CO2': 1.45, 'H2S': 1.9, 'H2O': 3.6}

❺ def __init__(self, screen, background):
     super().__init__()
     self.screen = screen
     self.background = background
   ❻ self.image = pg.Surface((4, 4))
     self.rect = self.image.get_rect()
   ❼ self.gas = random.choice(list(Particle.gases_colors.keys()))
     self.color = Particle.gases_colors[self.gas]
   ❽ self.vel = Particle.VELOCITY_SO2 * Particle.vel_scalar[self.gas]
   ❾ self.x, self.y = Particle.VENT_LOCATION_XY
   ❿ self.vector()
```

Определите класс с именем `Particle`, который представляет любую молекулу газа, способную образовывать вулканический шлейф ❶. Родителем этого класса, пока-

занного в скобках, будет класс `Sprite`. Это означает, что класс `Particle` является производным от встроенного в `pygame` типа, именуемого `Sprite`. Спрайты — это просто двухмерные растровые изображения, которые представляют дискретные игровые объекты, такие как ракеты или астероиды. Вы наследуете у класса `Sprite`, т. е. добавляете его атрибуты и методы в свой новый класс, путем передачи `pg.sprite.Sprite` в ваш класс `Particle`, точно так же, как вы передаете аргумент в функцию.

В качестве атрибутов класса задайте свойства, общие для всех частиц. Первым атрибутом является словарь, который увязывает типы частиц с цветом, благодаря чему вы можете различать частицы во время симулирования ❷. Эти цвета будут использоваться для частицы, ее траектории и имени в легенде.

Теперь задайте четыре константы: `VENT_LOCATION_XY`, `IO_SURFACE_Y`, `GRAVITY` и `VELOCITY_SO2` ❸. Первая константа — это координаты *x* и *y* устья изображенного на снимке вулкана Тваштар, которые будут представлять "точку запуска" всех частиц (рис. 13.3). Поначалу эти значения я просто угадал, а потом их донастроил, когда симуляция была доведена до полной готовности.

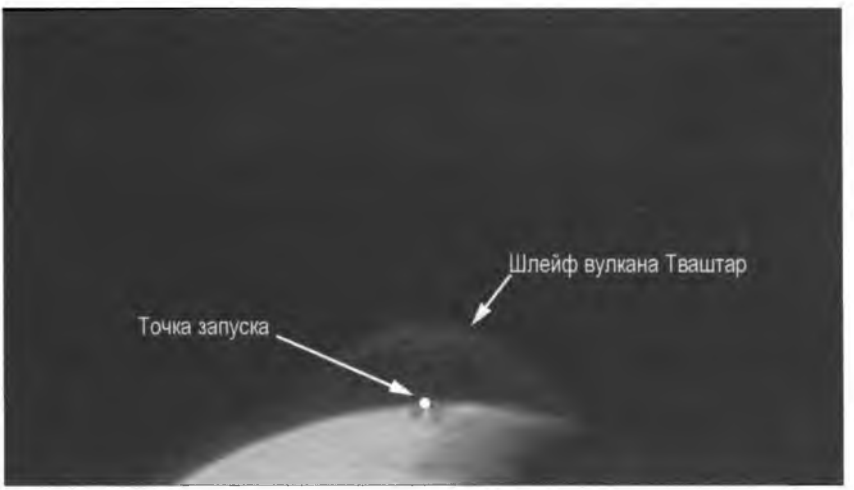


Рис. 13.3. Фоновой снимок для симуляции с аннотированной точкой запуска частиц

Вторая константа — это значение *y* самой высокой точки на поверхности Ио (на снимке), которая пересекается с внешним краем шлейфа SO_2 (см. рис. 13.2). В этом значении *y* вы будете останавливать все падающие частицы, поэтому вид будет оптимизирован для SO_2.

Третья константа представляет собой ускорение гравитации, которая на Земле составляет 9,86 м/с^2, а на Ио — 1,796 м/с^2. Но здесь вы имеете дело с пикселами и кадрами, а не с реальными единицами, и поэтому вам нужно поэкспериментировать с целью отыскать значение, которое будут приемлемо для шкалы вашей игры/симуляции. Выбранный мной вариант `0.5` был произвольным, хотя я и руководствовался знанием того, что конкретно работает хорошо в аркадных играх.

Четвертая константа — это скорость выброса частицы SO_2 в пикселах на кадр. Напомним, что шлейфы в основном состоят из SO_2, и поэтому нужно использовать параметры, которые позволят частицам SO_2 "вписываться" в снимок шлейфа вулкана Тваштар, а затем настроить скорость оставшихся частиц относительно SO_2. Ни гравитация, ни значение `VELOCITY_SO2` не являются уникальными. Если бы для гравитации я выбрал большее значение, то мне нужно было бы увеличить `VELOCITY_SO2`, с тем чтобы частицы SO_2 по-прежнему "заполняли" область шлейфа на снимке NASA.

Далее создайте словарь скаляров для скорости частиц ❹. Для каждой частицы деление атомной массы SO_2 (64) на атомную массу частицы дает скаляр. Поскольку SO_2 является эталонной частицей, ее скаляр равен 1. Позже для получения скорости частиц без SO_2 вы умножите константу `VELOCITY_SO2` на скаляр. Как можно видеть, все остальные частицы являются легче SO_2 и должны производить больший шлейф.

Определите метод-конструктор для объекта-частицы ❺. Для рисования и проверки границ симуляции понадобятся параметр `self` и переменная `screen`, представляющая экран, а также переменная `background` для фона, который будет представлять снимок шлейфа вулкана Тваштар. Вы зададите значения переменных `screen` и `background` позже в функции `main()`, определенной в конце программы. Обратите внимание, что, хотя в этой книге для краткости литералы документирования являются, как правило, однострочными, вы наверняка хотели бы включить эти типы параметров в литерал документирования класса. Дополнительные рекомендации по оформлению литералов документирования классов смотрите на веб-странице **https://www.python.org/dev/peps/pep-0257/**.

Внутри метода `__init()__` сразу же вызовите метод инициализации для встроенного класса `Sprite` с использованием метода `super`. В результате будет инициализирован спрайт и установлены необходимые ему атрибуты `rect` и `image`. Благодаря методу `super()` не нужно явно ссылаться на базовый класс (`Sprite`). Дополнительную информацию о методе `super()` можно получить, посетив веб-страницу с документацией по адресу **https://docs.python.org/3/library/functions.html#super**.

Далее сообщите частице (`self`) о том, что она будет использовать переменные `screen` и `background`, назначив их соответствующим атрибутам.

Пакет `pygame` размещает снимки и рисунки на прямоугольной поверхности. На самом деле, объект-поверхность `Surface` является сердцем и душой пакета `pygame`, и даже атрибут `screen` представляет собой экземпляр поверхности `Surface`. Назначьте изображение частицы объекту `Surface` и сделайте его квадратом со сторонами длиной 4 пиксела ❻.

Далее необходимо получить объект `rect` вашей снимковой поверхности. Он в основном представляет собой прямоугольник, ассоциированный с объектом `Surface`, в котором пакет `pygame` нуждается, и для того надо определить размеры и положение объекта `Surface`.

Выберите тип частиц (`gas`), случайно перебирая ключи в словаре `gases_colors` ❼. Обратите внимание, что для этого он конвертируется в список. Поскольку атрибут

экземпляра с именем `gases_colors` можно задать изнутри метода `__init()__`, включите имя класса — а не ключевое слово `self` — для обеспечения того, чтобы можно было ссылаться на атрибут класса.

Раз у вас есть тип, то вы можете использовать его в качестве ключа в словарях, которые вы создали ранее, для получения доступа к таким вещам, как цвета и скаляры. Начните с получения правильного цвета, соответствующего выбранной частице, затем получите его значение `vel_scalar` и используйте его для определения скорости частицы ❸.

Объект-частица будет создан в устье вулкана, поэтому получите его начальное местоположение в точке с координатами (x, y), распаковав кортеж `VENT_LOCATION_XY` ❾. Завершите метод вызовом метода `vector()`, который вычислит вектор движения частицы ❿.

Выброс частицы

В листинге 13.3 представлен метод `vector()`, который определяет ориентацию запуска частицы и вычисляет ее начальные векторные компоненты dx (дельта x) и dy (дельта y).

Листинг 13.3. Определяет метод `vector()` класса `Particle`. Файл tvashtar.py, часть 3

```
❶ def vector(self):
      """Вычислить вектор частицы при запуске."""
   ❷ orient = random.uniform(60, 120)    # 90 значит вертикальный
   ❸ radians = math.radians(orient)
   ❹ self.dx = self.vel * math.cos(radians)
     self.dy = -self.vel * math.sin(radians)
```

Метод `vector()` ❶ вычисляет вектор движения частицы. Начните с того, что выберите направление запуска частицы, и назначьте это направление переменной ориентации `orient` ❷. Поскольку взрывные вулканические извержения выстреливают материал не прямо вверх, а в многочисленных направлениях, выберите направление наугад, используя диапазон, который составляет 30° по обе стороны от 90°, где 90 является вертикальным запуском.

Диапазон переменной `orient` был выбран методом проб и ошибок. Этот параметр, наряду с константами `VELOCITY_SO2` и `GRAVITY`, представляет собой "ручки" управления, которые можно поворачивать в целях калибровки поведения частицы SO_2 на снимке шлейфа. После того как вы отрегулировали константы так, чтобы максимальная высота частицы соответствовала вершине шлейфа, можно настроить диапазон углов так, чтобы частицы SO_2 достигали — но не превышали — боковых границ шлейфа (рис. 13.4).

Модуль `math` использует *радианы*, а не градусы, поэтому конвертируйте переменную `orient` в радианы ❸. Радиана — это стандартная единица измерения угла, рав-

ная углу, получающемуся, когда радиус делает полный круг по окружности (см. левую часть рис. 13.5). Одна радиана составляет чуть меньше 57,3°. Правая часть рис. 13.5 представляет сравнение радиан и градусов для нескольких общеизвестных углов. Для конвертации градусов в радианы можно либо — как какой-нибудь простак — умножить градусы на π и разделить на 180, либо применить модуль `math`!

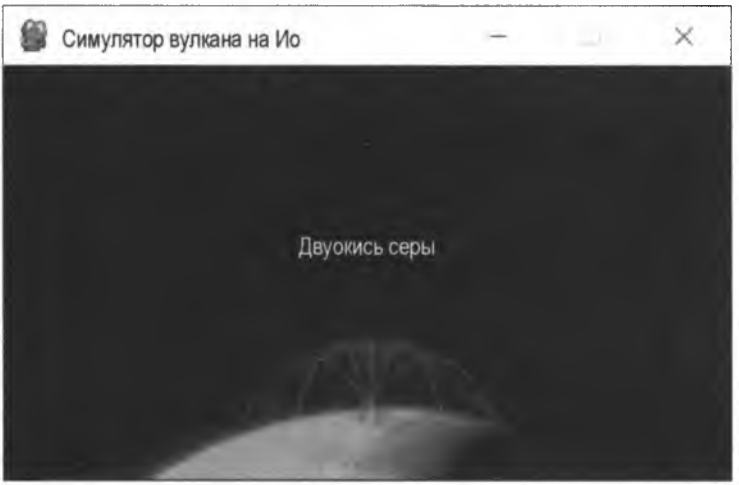


Рис. 13.4. Калибровка переменной ориентации `orient` по шлейфу вулкана Твашстар

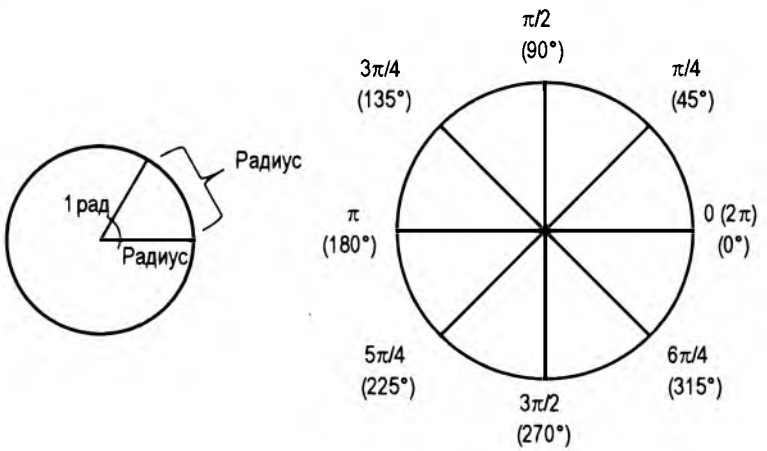


Рис. 13.5. Определение радианы (слева) и общеизвестных углов в радианах и градусах (справа)

Объекты движутся в `pygame` с шагом x и y. Направление и скорость частицы используются для получения ее векторных компонент дельта x (dx) и дельта y (dy). Они представляют разницу между начальной позицией частицы и ее местоположением после завершения одного игрового цикла.

Векторные компоненты вычисляются с помощью тригонометрии. Полезные тригонометрические соотношения приведены на рис. 13.6.

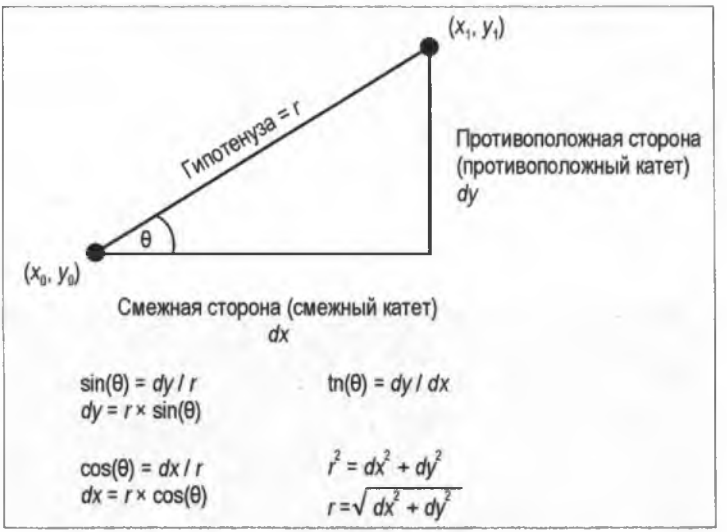


Рис. 13.6. Общеизвестные тригонометрические уравнения, используемые в играх

Для угла θ используется переменная `orient`. Атрибут `self.vel` приравнивается к *r*. Зная эти две компоненты, можно применить тригонометрическое соотношение для получения `self.dx` и `self.dy` ❹. Для получения `self.dx` умножьте `self.vel` на косинус ориентации `orient`, а для получения `self.dy` умножьте `self.vel` на синус ориентации `orient`. Обратите внимание, что вы должны сделать значение `self.dy` отрицательным, т. к. частицы выбрасываются вверх, а значения *y* в `pygame` увеличиваются вниз.

Обновление частиц и обработка граничных условий

Листинг 13.4 завершает класс `Particle`, определяя метод обновления частиц по мере их перемещения по экрану. Сюда входит применение силы гравитации, рисование линии для отслеживания траектории частицы и "уничтожение" частицы, когда она выдвигается за пределы экрана или под поверхность Ио.

Листинг 13.4. Определяет метод `update()` и завершает класс `Particle`. Файл tvashtar.py, часть 4

```
❶ def update(self):
      """Применить гравитацию, нарисовать траекторию и обработать
         граничные условия."""
   ❷ self.dy += Particle.GRAVITY
   ❸ pg.draw.line(self.background, self.color, (self.x, self.y),
                  (self.x + self.dx, self.y + self.dy))
   ❹ self.x += self.dx
     self.y += self.dy

   ❺ if self.x < 0 or self.x > self.screen.get_width():
        ❻ self.kill()
```

```
❼ if self.y < 0 or self.y > Particle.IO_SURFACE_Y:
      self.kill()
```

Определите метод `update()`, который в качестве аргумента принимает `self` ❼. Примените силу гравитации, добавляя атрибут класса `GRAVITY` в `self.dy` во время каждого игрового цикла ❷. Гравитация — это вектор силы, который работает только в вертикальном направлении, поэтому под ее влиянием оказываться только `self.dy`.

Для отрисовки траектории позади частицы используйте метод `draw.line()` пакета `pygame`, который в качестве аргументов принимает фоновой снимок Ио, цвет частицы и координаты предыдущего и текущего местоположений частицы ❸. Для получения текущего местоположения атрибуты `self.dx` и `self.dy` добавляются в `self.x` и `self.y`.

Затем обновите атрибуты `self.x` и `self.y`, добавив `self.dx` и `self.dy`, точно так же, как вы сделали в методе `draw.line()` ❹.

Теперь проверьте, не прошла ли частица левую или правую границу экрана ❺. Используйте `self.x`, равный нулю для левой стороны, и получите ширину атрибута `screen` для правой стороны. Если частица вышла за пределы одной из сторон экрана, то примените встроенный в пакет метод `kill()`, удалив ее из всех групп, которые ее содержат ❻. Как вы увидите позже, для управления спрайтами пакет `pygame` использует контейнеры, именуемые спрайтовыми группами, а удаление спрайта из группы выводит его из игры.

Повторите этот процесс для направления *y* ❼, но для максимального значения используйте константу `IO_SURFACE_Y` класса `Particle`, которая остановит частицу вблизи поверхности Ио, где остановится частица SO_2 (см. рис. 13.2 и 13.4).

Определение функции *main()*

Листинг 13.5 определяет первую часть функции `main()`, которая задает игровой экран, заголовок окна, легенду, спрайтовую группу и игровые часы.

Листинг 13.5. Определяет первую часть функции `main()`. Файл tvashtar.py, часть 5

```
def main():
    """Настроить и выполнить экран и цикл игры."""
❼   screen = pg.display.set_mode((639, 360))
❷   pg.display.set_caption('Симулятор вулкана на Ио')
❸   background = pg.image.load('tvashtar_plume.gif')

    # настроить легенду с цветовой кодировкой
❹   legend_font = pg.font.SysFont('None', 24)
❺   water_label = legend_font.render('--- H2O', True, WHITE, BLACK)
    h2s_label = legend_font.render('--- H2S', True, DK_GRAY, BLACK)
```

```
co2_label = legend_font.render('--- CO2', True, GRAY, BLACK)
so2_label = legend_font.render('--- SO2/S2', True, LT_GRAY, BLACK)

❻ particles = pg.sprite.Group()

❼ clock = pg.time.Clock()
```

Первым делом следует задать переменную `screen`, используя метод `display.set_mode()` пакета `pygame` ❶. Аргументами являются размеры в пикселах; в данном случае вы используете значения немного меньше, чем у снимка NASA. Это делается для обеспечения хорошей подгонки. Обратите внимание, что размеры должны быть предоставлены в виде кортежа, поэтому необходимо включить два набора скобок.

Далее объявите свое игровое окно с помощью метода `display.set_caption()` пакета `pygame` ❷, затем назначьте переменной `background` фотографию NASA шлейфа вулкана Тваштар ❸. Используйте метод `image.load()` пакета `pygame`, который создаст новый объект `Surface` из снимка. Пакет `pygame` поддерживает несколько форматов графических изображений, включая PNG, JPG и GIF. Возвращенный объект `Surface` наследует информацию о цвете и прозрачности из снимкового файла. Поскольку вы импортируете снимок в оттенках серого, выбор цвета будет ограничен.

Теперь добавьте код построения легенды, которая будет изображаться в левом верхнем углу экрана.

Назовите переменную `legend_font` и используйте метод `font.SysFont()` пакета `pygame` для выбора `None` при размере 24 ❹. Вы будете использовать его при визуализации текста. Модуль `font` пакета `pygame` позволяет отрисовывать новый набор шрифтов TrueType на новом объекте `Surface`. Если вы не хотите указывать шрифт, то `pygame` поставляется со встроенным стандартным шрифтом, к которому вы можете обратиться, передав значение `None` в качестве имени шрифта.

Разместите имена частиц в порядке возрастания их атомных масс, где самые легкие расположены на вершине. Для создания надписи вызовите функцию `render()` на созданном ранее объекте `legend_font`, создав новый объект `Surface` ❺. Передайте ему текст, затем `True` (для устранения контурных неровностей — антиалиасинг — что сделает текст визуально более гладким), а затем цвет описываемой частицы. Последний аргумент, `BLACK`, является необязательным и делает цвет фона надписи черным ради разборчивости текста над всеми нарисованными на экране траекториями частиц. Повторите этот процесс для трех оставшихся частиц и добавьте `S2` в `so2_label`, т. к. оба газа имеют одинаковую атомную массу и в симуляции будут вести себя одинаково.

Теперь создайте спрайтовую группу с именем `particles` ❻. Поскольку в играх обычно по экрану перемещается несколько спрайтов, то для управления ими пакет `pygame` использует контейнер — спрайтовую группу. В действительности же вы просто обязаны поместить спрайты в группу, иначе они ничего не сделают.

Завершите этот раздел, создав объект хронометрирования `Clock` для отслеживания и управления кадровой частотой симуляции ❼. "Часы" пакета `pygame` контролирует

скорость выполнения игры на основе числа кадров в секунду (frames per second, fps). Вы установите это значение в следующем разделе.

Завершение функции *main()*

Листинг 13.6 завершает функцию `main()` тем, что устанавливает скорость, с которой будет выполняться симуляция (в кадрах в секунду), и начинает цикл `while`, который выполняет эту симуляцию фактически. Он также обрабатывает события, которые происходят, когда пользователь контролирует программу с помощью мыши, джойстика или клавиатуры. Поскольку это не настоящая игра, а симуляция, управление со стороны пользователя ограничивается закрытием окна. Листинг заканчивается в глобальной области стандартным фрагментом кода для выполнения программы в виде модуля либо в автономном режиме.

Листинг 13.6. Запускает игровые часы и цикл и обрабатывает события в функции `main()`. Файл tvashtar.py, часть 6

```
❶ while True:
    ❷ clock.tick(25)
    ❸ particles.add(Particle(screen, background))
    ❹ for event in pg.event.get():
          if event.type == pg.QUIT:
              pg.quit()
              sys.exit()

    ❺ screen.blit(background, (0, 0))
      screen.blit(water_label, (40, 20))
      screen.blit(h2s_label, (40, 40))
      screen.blit(co2_label, (40, 60))
      screen.blit(so2_label, (40, 80))

    ❻ particles.update()
      particles.draw(screen)

    ❼ pg.display.flip()

❽ if __name__ == "__main__":
      main()
```

Начните цикл `while`, который выполняет симуляцию ❶. Затем примените метод `clock.tick()`, который установит в симуляции скоростной лимит ❷. Передайте ему `25`, в результате чего максимальная частоту кадров будет установлена равной 25 кадрам в секунду. Указанное значение можно свободно увеличить, если есть желание получить более энергичный вулкан.

Теперь настало время для того, чтобы на сцене появилась сама звезда шоу. Инстанцируйте частицу с помощью класса `Particle`, передав ему в качестве аргументов

переменные `screen` и `background`, и добавьте новую частицу в спрайтовую группу `particles` ❸. С каждым кадром из вулканического жерла будет случайно появляться и запускаться новая частица, производя приятный глазу брызг частиц (рис. 13.7).

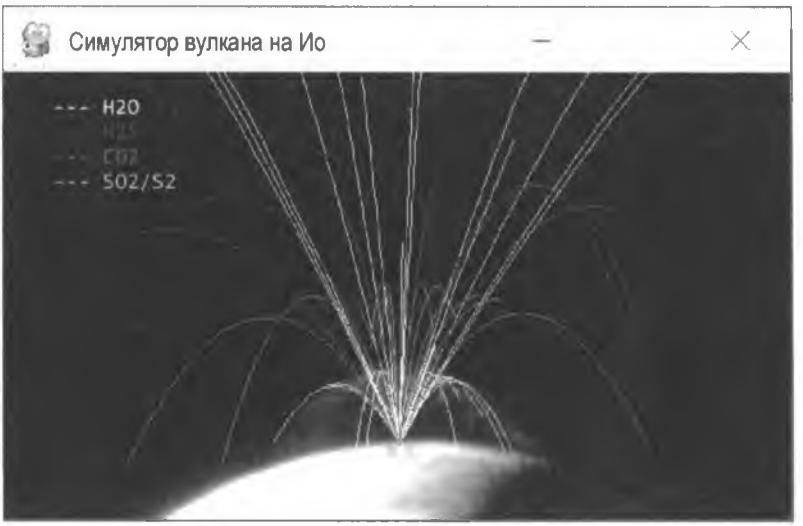


Рис. 13.7. Запуск симуляции, в которой случайные частицы генерируются со скоростью 25 кадров в секунду

Начните цикл `for` по обработке событий ❹. Все события, произошедшие во время текущего кадра, регистрируются и хранятся пакетом `pygame` в событийном буфере. Его метод `event.get()` создает список всех этих событий для их вычисления по очереди. Если происходит событие `QUIT` (когда пользователь закрывает окно игры), то вызываются метод `quit()` пакета `pygame` и системный метод `exit()`, которые завершают симуляцию.

Для визуализации игровых объектов и обновления визуального изображения `pygame` использует процесс под названием "блиттинг". Указанный процесс означает блочную передачу растровых данных, с помощью которой просто копируются пикселы одного прямоугольного объекта `Surface` в другой. При переносе фона на экран вы покрываете экран снимком Ио. С помощью блочной передачи можно взять тот же самый снимок и несколько раз отобразить его на экране в разных местах. Этот процесс может замедлиться, поэтому разработчики игр используют умные технические решения для устранения данного недостатка, такие как перенос растровых фрагментов в каждом игровом цикле только вокруг тех областей, которые в настоящее время обновляются, а не всего экрана.

Для переноса фона на экран вызовите метод `blit()` на объекте `screen` и передайте ему необходимые аргументы с источником и местом назначения ❺. В первом примере переменная фона `background` является источником, а местом назначения — координаты левого верхнего угла фона. Поскольку фон будет покрывать весь экран, используйте точку начала координат экрана, т. е. `(0, 0)`. Повторите это для надписей легенды, поместив их в левый верхний угол экрана.

Затем вызовите метод `update()` на группе частиц `particles` ❻. Этот метод не обновляет экран, но заставляет спрайты выполнять собственные методы `update()`. После этого вы используете метод `draw()` для переноса спрайтов на экран, основываясь на атрибутах `rect` каждого спрайта. Для этого метода нужна поверхность рисования, поэтому передайте ему объект `screen`.

Метод `draw()` позаботился о переносе спрайтов, поэтому сейчас для обновления фактической графики игры вам нужно лишь применить метод `flip()` ❼. Флиппинг, или отражение, — это вид двойной буферизации, где все переносится с объекта `screen` на фактический экран. Отражение обходит внутренне присущий медленный процесс передачи графики, способный вызывать мерцание на экране, выполняя работу на закулисном прямоугольнике и после этого используя версию метода `blit()` для окончательного копирования на экран.

Листинг заканчивается вне функции `main()` фрагментом кода, который позволяет программе работать как модуль либо в автономном режиме ❽.

Выполнение симуляции

На рис. 13.8 показан результат работы симулятора в течение примерно минуты. Шлейф водяного пара выходит за пределы верхней части окна. Следующий по высоте шлейф образуется сероводородом, за которым следует углекислый газ, а затем двуокись серы/сернистый газ (S_2), которые по конструкции идеально соответствуют шлейфу Тваштара.

Для выполнения симулятора только с SO_2 перейдите к методу `__init__` класса `Particle` и измените строки, в которых вы выбираете экземплярные атрибуты `gas` и `color`:

```
self.gas = 'SO2'
self.color = random.choice(list(Particle.gases_colors.values()))
```

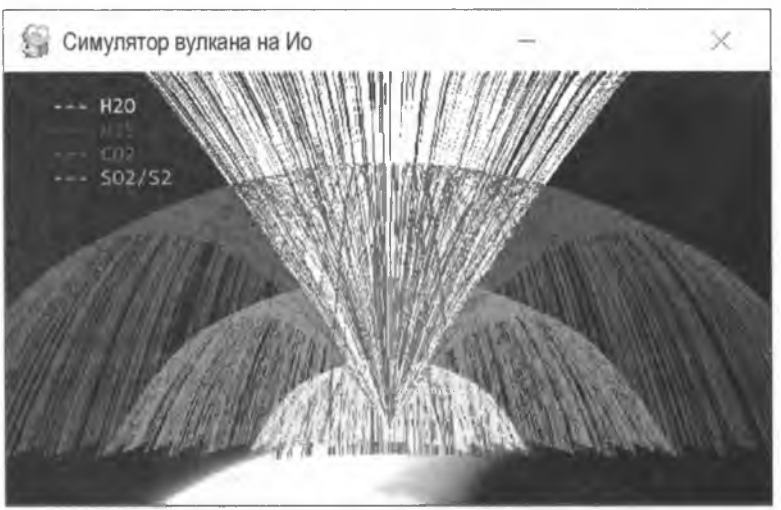


Рис. 13.8. Результаты выполнения программы tvashtar.py в течение одной минуты

Выбирая цвет наугад, вы сохраняете ощущение движения в шлейфе после того, как все возможные углы ориентации `self.orient` были исчерпаны. А если вы хотите ускорить или замедлить извержение, то перейдите к функции `main()` и поэкспериментируйте с параметром кадров в секунду метода `clock.tick()`.

В реальной жизни состав материала шлейфа был выведен с помощью спектроскопии, т. е. измерительного приема, который позволяет анализировать взаимодействие света с веществом. Сюда входят как видимые, так и невидимые длины волн, которые поглощаются, излучаются или рассеиваются. "Спектры выбросов" вместе с красками, наносимыми на поверхность, давали ключевое доказательство наличия шлейфов, богатых серой.

Резюме

В этой главе вы узнали, как использовать пакет `pygame` для симулирования гравитации и построения анимации извержения внеземного вулкана. В следующей главе вы воспользуетесь пакетом `pygame` для построения настоящей аркадной игры с взаимодействием игроков и условиями выигрыша-проигрыша.

Дальнейшее чтение

Книга Энди Хэрриса "Программирование игр: L-линия, экспресс-линия в обучении" (Harris A. Game programming: the L-line, the express line to learning. Wiley, 2007) представляет собой чрезвычайно полезное и подробное 570-страничное введение в `pygame`.

Книга Джонатана Харбора "Еще больше языка Python для абсолютного новичка" (Harbour J. More Python for the absolute beginner. Cengage Learning Course Technology, 2012) основана на предшествующей книге "Язык Python для абсолютного новичка" (Python for the Absolute Beginner), в которой используется игровой подход на основе `pygame`.

Книга Эла Свейгарта "Учим Python, делая крутые игры" (Sweigart A. Invent your own computer games with Python. 4th Edition. No Starch Press, 2016) является хорошим введением в Python и в проектирование игр для начинающих.

Онлайновое "руководство для новичков" по `pygame` доступно по адресу **https://www.pygame.org/docs/tut/newbieguide.html**, а соответствующую "шпаргалку" можно найти по адресу **http://www.cogsci.rpi.edu/~destem/gamedev/pygame.pdf**.

Статья Уильяма Дж. Макдониела и соавт. "Трехмерное моделирование газа и пыли в плюме вулкана Пеле на Ио" (McDoniel W. J. et al. Three-dimensional simulation of gas and dust in Io's Pele Plume) документирует симуляцию плюма вулкана Пеле на Ио с использованием прямой симуляции Монте-Карло и суперкомпьютеров в Техасском центре передовых вычислений в Университете Техаса. Статья доступна по адресу

http://cfpl.ae.utexas.edu/wp-content/uploads/2016/01/McDoniel_PeleDust.pdf.

Практический проект: весь путь до конца

Вы — один из лучников короля Генриха в битве при Азенкуре. Французы атакуют, и вы хотите выстрелить по ним как можно дальше. Под каким углом вы держите свой длинный лук?

Если у вас когда-либо были уроки физики, то вы, вероятно, знаете, что ответом будет 45°. Но можно ли доверять этому тонкошеему физику? Для проверки лучше выполнить быструю компьютерную симуляцию. Скопируйте и отредактируйте код tvashtar.py так, чтобы он случайно выстреливал частицы под углами 25, 35, 45, 55 и 65 градусов. Сделайте `self.color` белым для 45° и серым для всех других углов (рис. 13.9).

Рис. 13.9. Симулятор вулкана на Ио, модифицированный для углов выброса под углами 25, 35, 45, 55 и 65 градусов

Решение задачи, practice_45.py, можно найти в приложении к книге либо скачать с веб-сайта **https://www.rwstarch.com/impracticalpython/**. Сохраните его в той же папке, что и файл tvashtar_plume.gif.

Сложные проекты

Продолжите свои эксперименты с приведенными ниже сложными проектами. Никаких решений не предусмотрено.

Ударный купол

Считается, что видимость гигантских шлейфов Ио усиливается за счет конденсации газа в пыль в ударном куполе, т. е. точке, где частицы газа достигают своей вершины и начинают падать обратно на поверхность. Используйте атрибут `self.dy`, отредактировав цвет траектории в копии программы tvashtar.py. Траектории в вершине

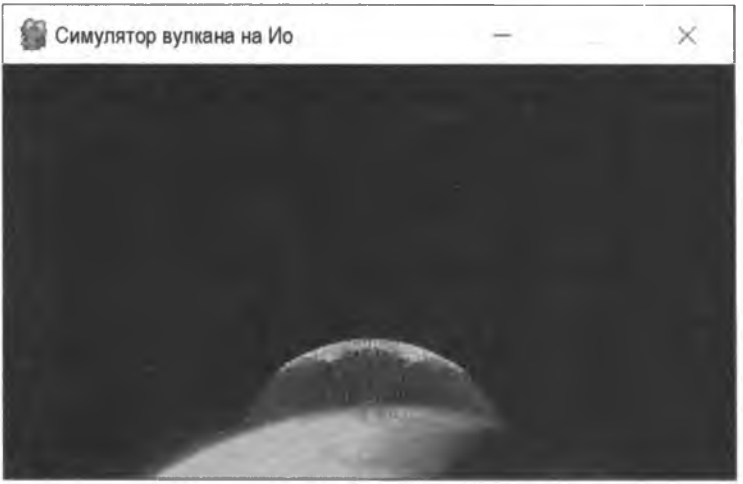


Рис. 13.10. Выделение ударного купола с помощью более светлых красок контура

шлейфа должны быть ярче, чем те, что расположены ниже (рис. 13.10). Как и во всех сложных проектах, никакого решения не предусмотрено.

Источник выбросов

Скопируйте и отредактируйте программу tvashtar.py так, чтобы симулировался только газ SO_2, и частицы были представлены маленькими белыми кружками без тянущихся позади траекторий (рис. 13.11).

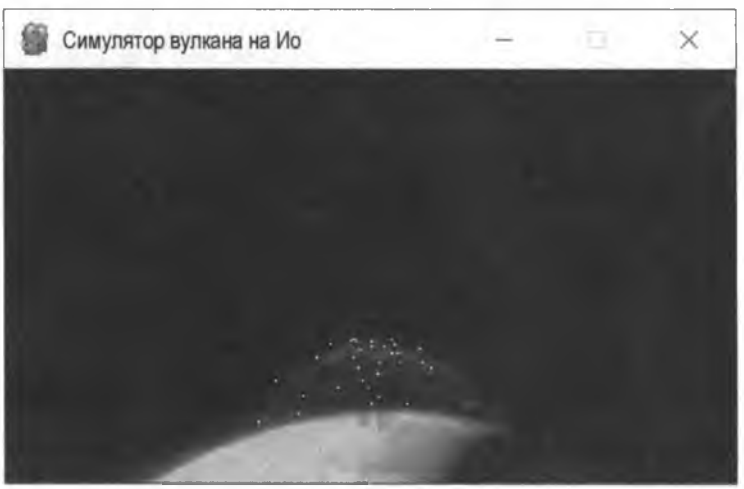


Рис. 13.11. Снимок симуляции SO_2, где кружки представляют отдельные частицы

Полет пули

Если выстрелить из пистолета прямо вверх на планете без атмосферы, то упадет ли пуля на поверхность с той же скоростью, с какой она вылетела из дула? Многие

отвечают на этот вопрос с трудом, но вы можете ответить на него с помощью языка Python. Скопируйте и отредактируйте программный код tvashtar.py так, чтобы он выбрасывал одну частицу SO_2 с ориентацией 90°. Напечатайте атрибут `self.y` частицы и абсолютное значение `self.dy` в координате точки запуска ($y = 300$). Сравните начальные и конечные значения скорости в этой точке, для того чтобы убедиться в их одинаковости или схожести.

ПРИМЕЧАНИЕ

Эпизод 50 американской научно-популярной телепередачи "Разрушители легенд (Myth Busters) посвящен мифу о том, что пули, выпущенные в воздух, сохраняют свою смертоносную способность, когда они в конечном итоге падают обратно. Они обнаружили, что пули, выпущенные совершенно вертикально на Земле, будут падать, замедляясь на обратном пути из-за сопротивления ветра. Если выстрелить немного не по вертикали, то пули сохранят свое вращение и баллистическую траекторию и вернутся на Землю со смертельной скоростью. Это был единственный миф, который когда-либо получал все три рейтинга (развенчан, правдоподобен и подтвержден)!

14

КАРТОГРАФИРОВАНИЕ МАРСА С ПОМОЩЬЮ ОРБИТАЛЬНОГО СПУТНИКА

Орбитальный спутник Марса Mars Orbiter был успешно выведен на марсианскую орбиту, но все оказалось не так хорошо. Орбита Марса является высокоэллиптической, и картографические задачи проекта требуют низкой высоты круговой орбиты. К счастью, для исправления ситуации на борту имеется достаточно топлива, если допустить, что яйцеголовые в Центре управления полетами обладают терпением и умением для того, чтобы его вытащить!

В данной главе вы спланируете и построите игру, основанную на этом скрипте. Вы снова будете использовать пакет `pygame` (обзор пакета `pygame` *см. в разд. "Кусочек пакета pygame" главы 13*) и внесете свой вклад в развитие междисциплинарного STEM-образования[1], сделав игру достаточно реалистичной для того, чтобы научить игроков основам орбитальной механики.

ПРИМЕЧАНИЕ

Хотя игровой спутник и индийский орбитальный спутник Mars Orbiter носят одинаковое название, упоминаемый в игре космический зонд Mars Orbiter не имеет прямого отношения к миссии зонда Mars Orbiter, запущенного Индийской организацией космических исследований (ISRO) в 2014 г. Игровой спутник создан по образцу беспилотной разведывательной станции Mars Global Surveyor, запущенной NASA в 1996 г.

[1] STEM (science, technology, engineering и mathematics) — наука, технология, инженерное дело и математика. — *Прим. перев.*

Астродинамика для геймеров

Поскольку вы хотите, чтобы ваша игра была максимально реалистичной, будет уместно привести быстрый обзор некоторых фундаментальных научных положений, лежащих в основе космического полета. Он будет кратким, сладким и приспособленным для разработки игры и для самой игры.

Закон универсальной гравитации (или всемирного тяготения)

Теория гравитации утверждает, что массивные объекты, такие как звезды и планеты, деформируют пространство и время вокруг себя, подобно тому как тяжелый шар для боулинга, помещенный на матрас, вызывает прогиб, который становится внезапным и резким рядом с шаром, но быстро выравнивается. Это поведение математически фиксируется законом всемирного тяготения Исаака Ньютона:

$$F = \frac{m_1 \cdot m_2}{d^2} G,$$

где F — это сила тяжести; m_1 — масса объекта 1; m_2 — масса объекта 2; d — расстояние между объектами; G — гравитационная постоянная ($6,674 \cdot 10^{-11}$ Н · м^2/кг^2).

Два объекта притягиваются друг к другу в соответствии с произведением их масс, деленным на квадрат расстояния между ними. Таким образом, гравитация намного сильнее, когда объекты находятся близко друг к другу, как глубокий прогиб матраса прямо под шаром для боулинга. В качестве иллюстрации 100-килограммовый человек будет весить на 250 грамм меньше на вершине горы Эверест, чем на уровне моря, где он будет на 8848 м ближе к центру Земли. (Предполагается, что масса планеты составляет $5,98 \cdot 10^{24}$ кг, а уровень моря — $6,37 \cdot 10^6$ м от центра.)

Сегодня мы обычно думаем о гравитации как о поле (как о матрасе в аналогии с шаром для боулинга), а не как о ньютоновской точке притяжения. Это поле попрежнему определяется законом Ньютона и приводит к ускорению, обычно выраженному в метрах на секунду в квадрате (м/с^2).

Согласно второму закону движения Ньютона, сила равна произведению массы и ускорения ($F = ma$). Сила, приложенная объектом 1 (m_1) к объекту 2 (m_2), может быть вычислена переписыванием уравнения гравитации следующим образом:

$$a = \frac{-G \cdot m_1}{d^2},$$

где a — ускорение; G — гравитационная постоянная; m_1 — масса одного из объектов; d — расстояние между объектами. Сила направлена от объекта 2 к центру масс объекта 1 (m_1).

Притяжение очень малых объектов к большим обычно игнорируется. Например, сила, приложенная спутником массой 1000 кг к Марсу, примерно в $1,6 \cdot 10^{-21}$ раз меньше силы, приложенной Марсом к спутнику! Поэтому в своей симуляции можно безопасно проигнорировать массу спутника.

ПРИМЕЧАНИЕ

Для упрощения в этом проекте расстояние вычисляется от центральных точек объектов. В реальной жизни орбитальный спутник испытывал бы тонкие изменения в гравитационном ускорении из-за изменений формы планеты, топографии, плотности коры и т. д. По данным Британской энциклопедии, эти изменения приводят к тому, что гравитационное ускорение на поверхности Земли варьируется примерно на 0,5%.

Законы движения планет Кеплера

В 1609 г. астроном Иоганн Кеплер обнаружил, что орбиты планет представляют собой эллипсы, что позволило ему объяснить и предсказать движение планет. Он также обнаружил, что отрезок линии, проведенный между Солнцем и вращающейся по орбите планетой, формирует собой равные площади за равные промежутки времени. Эта идея, известная как второй закон движения планет Кеплера, продемонстрирована на рис. 14.1, где планета показана в разных точках своей орбиты.

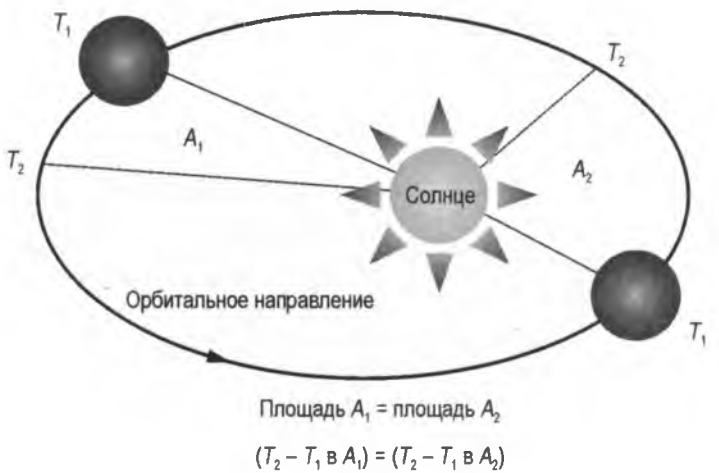


Рис. 14.1. Второй закон движения планет Кеплера: орбитальная скорость увеличивается по мере приближения планет к Солнцу

Этот закон применим ко всем небесным телам, и он означает, что орбитальный объект ускоряется по мере приближения к телу, вокруг которого он вращается, и замедляется, когда он удаляется.

Орбитальная механика

Вращение по орбите, в сущности, представляет собой вечное свободное падение. Вы падаете в ядро гравитационного колодца планеты — расположенного в ее буквальном ядре, — но ваша тангенциальная скорость настолько быстра, чтобы вы продолжаете не попадать по планете (рис. 14.2). До тех пор пока вы уравновешиваете свой импульс силой тяжести, орбитальное вращение никогда не закончится.

При вращении вокруг планеты в вакууме пространства могут происходить вещи, которые противоречат интуитивному пониманию. Без трения или сопротивления ветра космический корабль может вести себя неожиданным образом.

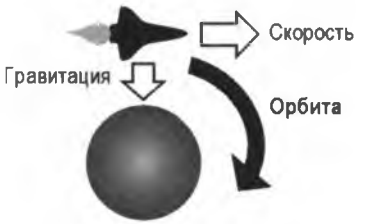


Рис. 14.2. Орбита достигается, когда скорость космического аппарата удерживает его "в свободном падении" вокруг небесного тела

Полет назад

Если вы когда-нибудь смотрели кинофильм "Звездный путь" (Star Trek), то вы наверняка заметили, как вращающийся по орбите космический корабль "Энтерпрайз", кажется, управляет своей траекторией вокруг планет, как автомобиль, едущий по трассе. Это, конечно, можно сделать — и определенно выглядит круто, но такое движение требует затрат драгоценного топлива. Если нет необходимости постоянно направлять определенную часть космического аппарата на планету, то нос космического аппарата всегда будет указывать в одном и том же направлении на протяжении всей его орбиты. В результате этого на каждой орбите будет время, когда будет казаться, что аппарат летит назад (рис. 14.3).

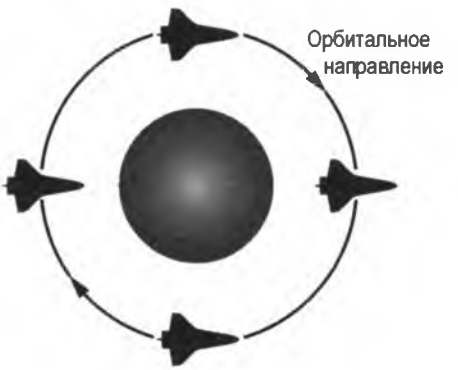


Рис. 14.3. Космические аппараты на орбите сохраняют прежнее положение, если только их не вынуждают к иному

Можно обвинить в этом Ньютона и его закон инерции, который гласит, что покоящийся объект остается в покое, а движущийся объект остается в движении с той же скоростью и в том же направлении, если только на него не действует несбалансированная сила.

Поднятие и опускание орбит

Тормоза в космосе не работают, трения нет, а инерция воспринимается очень серьезно. Для снижения орбиты космического корабля нужно запустить двигатели для уменьшения его скорости с тем, чтобы он продолжил падать в гравитационный колодец планеты. Для этого необходимо ретроградировать космический корабль так,

чтобы его нос был обращен в противоположную сторону от нынешнего вектора скорости — причудливый способ сказать, что вы должны лететь задом наперед. При этом, разумеется, предполагается, что основные двигатели находятся в задней части космического аппарата. И наоборот, если требуется поднять орбиту, то необходимо проградуировать космический аппарат, в результате чего его нос будет находиться в направлении, в котором вы путешествуете. Эти две идеи показаны на рис. 14.4.

Рис. 14.4. Прогрaдация и ретрогрaдация определяются ориентацией носовой части космического аппарата относительно направления его движения вокруг тела, которое он облетает

Занятие внутренней дорожки

Допустим, вы гонитесь за другим вращающимся по орбите космическим кораблем. Что нужно для того, чтобы его догнать: ускориться или замедлиться? Согласно второму закону Кеплера, нужно замедлиться. В результате этого ваша орбита будет снижена, что приведет к более высокой орбитальной скорости. Это подобно тому, как в скáчках вы хотите занять внутреннюю дорожку.

На левой стороне рис. 14.5 два космических челнока находятся бок о бок на одной и той же орбите, двигаясь с одинаковой скоростью.

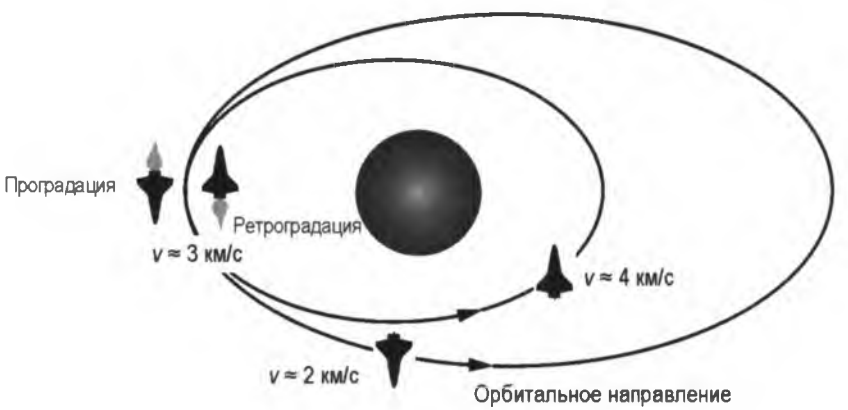


Рис. 14.5. Орбитальный парадокс: замедлиться, чтобы ускориться!

Ближайший к планете челнок поворачивается на 180° и включает ретроградную тягу, замедляя свою непосредственную скорость. Внешний челнок выполняет проградное движение, которое увеличивает его непосредственную скорость. Они одновременно останавливают тягу, и внутренний челнок падает на более низкую орбиту, тогда как внешний челнок переносится на более высокую орбиту. Через час или около того внутренний челнок уже движется гораздо быстрее из-за его большей сближенности с планетой и полным ходом продвигается к тому, чтобы догнать и перегнать внешний челнок.

Округлость эллиптической орбиты

Высокоэллиптические орбиты можно сделать круговыми, применяя импульсы двигателя в зависимости от ситуации в апоапсиде либо периапсиде. *Апоапсида* (именуемая апогеем, если объект вращается вокруг Земли) — это самая высокая точка на эллиптической орбите — точка, в которой объект находится дальше всего от тела, вокруг которого он вращается (рис. 14.6). *Периапсида* (перигей, если объект вращается вокруг Земли) — это самая низкая точка на орбите.

Для поднятия периапсиды космический аппарат выполняет проградную тягу в апоапсиде (см. левую часть рис. 14.7). Для снижения орбиты во время скругления кос-

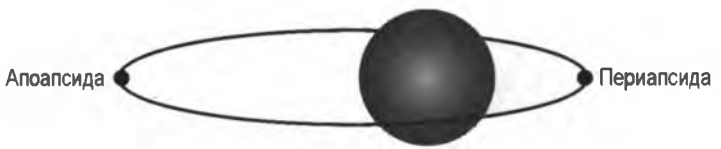


Рис. 14.6. Расположение апоапсиды и периапсиды на эллиптической орбите

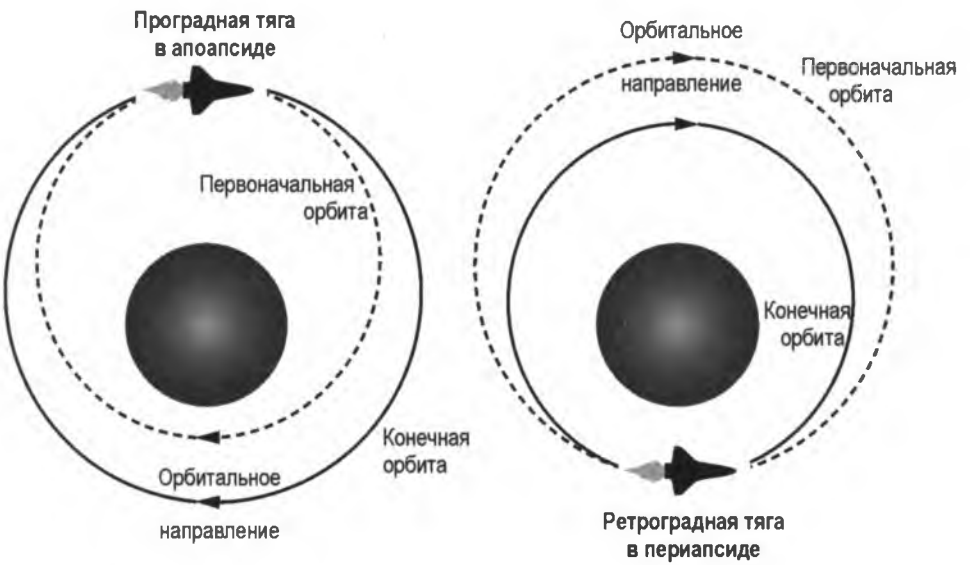


Рис. 14.7. Скругление и поднятие орбиты в апоапсиде (слева) и скругление и опускание орбиты в периапсиде (справа)

мический аппарат должен выполнить ретроградную тягу в периапсиде (см. правую часть рис. 14.7).

Часть этого маневра, которая несколько противоречит интуитивному пониманию, заключается в том, что первоначальная орбита (т. е. орбита, которая была бы) и конечная, или фактическая, орбита совпадут в точке приложения импульса двигателя.

Поднятие и опускание орбит с помощью гомановской траектории перехода

Орбита с переходом по эллипсу методом Гомана использует эллиптическую орбиту с целью перехода между двумя круговыми орбитами в одной плоскости (рис. 14.8). Орбита может быть поднята или опущена. Маневр происходит относительно медленно, но при этом расходуется минимально возможный объем топлива.

Рис. 14.8. Переход на нижнюю круговую орбиту с использованием гомановского перехода с орбиты на орбиту

Для перехода на орбиту с другой периапсидой и апоапсидой космическому аппарату требуются два импульса двигателя. Один импульс перемещает космический аппарат на переходную орбиту, а другой перемещает его на конечную, целевую орбиту. При поднятии орбиты космический аппарат применяет изменение скорости в направлении движения, а при опускании орбиты — изменение скорости, противоположное направлению движения. Как показано на рис. 14.8, изменения скоростей происходят на противоположных сторонах орбиты. Без второй тяги двигателя орбиты все равно пересекутся в точке первой тяги. Это показано на рис. 14.7 справа.

Поднятие и опускание орбит методом однотангенциального сжигания

Метод *однотангенциального сжигания*[2] переносит космический аппарат между орбитами быстрее, но менее эффективно, чем гомановский переход. Термин "сжи-

[2] Однотангенциальное сжигание (one-tangent burn) — это метод переноса, в котором переходная орбита является касательной по отношению к первоначальной орбите. См. **https://www.aerospacengineering.net/orbit-maneuvers/**. — *Прим. перев.*

гание" — это просто еще один вариант обозначения тяги или импульса двигателя. Как и в случае с переходом методом Гомана, орбиты могут быть подняты или опущены.

Для выполнения указанного маневра требуется два импульса двигателя, первый касательный (тангенциальный) к орбите, а второй — некасательный (рис. 14.9). Если первоначальная орбита является круговой, как на рисунке, то все точки вдоль нее представляют собой апоапсиду и периапсиду, и космический аппарат может применить свое первое сжигание в любое время.

Рис. 14.9. Переход на более высокую круговую орбиту методом однотангенциального сжигания

Так же как и при гомановском переходе с орбиты на орбиту, проградное сжигание поднимает орбиту, а ретроградное сжигание ее снижает. Если орбита является эллиптической, то первым сжиганием будет проградное сжигание в апоапсиде для поднятия орбиты либо ретроградным сжиганием в периапсиде для ее опускания.

Исполнение спиральной орбиты со спиральным переносом

Спиральный перенос использует непрерывное, низкоскоростное сжигание топлива с целью изменения размера орбиты. В игровом процессе его можно симулировать, используя как ретроградное, так и проградное сжигание, которые характерны кратковременностью и равномерностью распределения, как показано на рис. 14.10.

Для опускания орбиты все сжигания топлива должны быть ретроградными; для поднятия орбиты космический аппарат использует проградные сжигания.

Исполнение синхронных орбит

На *синхронной орбите* космическому аппарату для одного оборота вокруг планеты требуется столько же времени, сколько требуется планете на то, чтобы сделать один оборот вокруг своей оси. Если синхронная орбита является параллельной экватору, без наклона орбиты, то такая орбита является *стационарной*; для наблюдателя на небесном теле, вокруг которого осуществляется вращение, спутник кажется неподвижным, с фиксированным положением на небе. Спутники связи обычно ис-

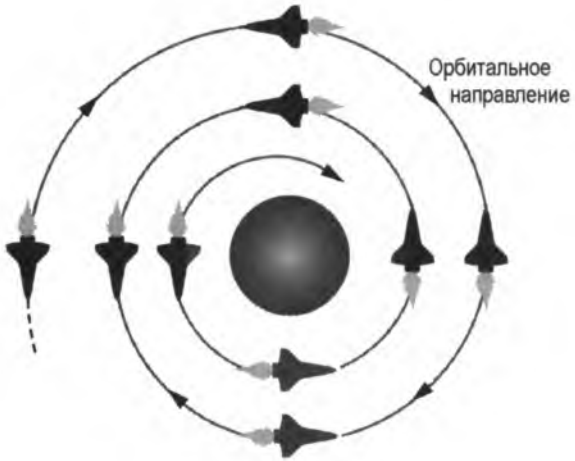


Рис. 14.10. Движение по спиральной орбите с использованием коротких ретроградных сжиганий топлива через регулярные промежутки времени

пользуют *геостационарные* орбиты, которые имеют высоту 22 236 миль над Землей. Аналогичная орбита будет называться *аэростационарной* вокруг Марса и *селеностационарной* вокруг Луны.

Проект 22: игра "Орбитальный спутник Марса"

В реальной жизни для точного исполнения орбитальных маневров применяется ряд уравнений. В игровом процессе вы будете использовать свою интуицию, терпение и рефлексы! Вам также в определенной степени нужно будет летать по приборам, используя в основном показания высоты космического аппарата и измерение округлости орбиты.

Цель

Применить пакет `pygame` для построения аркадной игры, которая учит основам орбитальной механики. Цель игры состоит в том, чтобы подтолкнуть спутник на круговую орбиту картографирования, не исчерпав топливо либо не сгорев в атмосфере.

Стратегия

Начните этап проектирования с эскиза игры, как вы делали в *главе 13*. Этот эскиз должен охватывать все основные моменты игры: как она будет выглядеть, как она будет звучать, какие предметы будут двигаться и как игра будет общаться с игроком (рис. 14.11).

Эскиз на рис. 14.11 описывает главный игровой процесс. Вам понадобится отдельный эскиз для описания условий выигрыша-проигрыша. Ключевыми моментами главного игрового процесса являются следующие:

- **Точка наблюдения** — это пункт управления полетом. Игровой экран должен напоминать монитор в Центре управления полетами, с которого игрок может управлять блуждающим космическим зондом.
- **Марс находится впереди и в центре.** Все любят красную планету, и поэтому она будет занимать центр угольно-черного экрана.
- **Марс анимирован.** Марсианский глобус будет медленно вращаться вокруг своей оси и отбрасывать тень. Спутник заметно потускнеет, когда он будет проходить через эту тень.
- **Первоначальная орбита спутника выбирается наугад.** Спутник появится при запуске с рандомизированной, но ограниченной ориентацией и скоростью. В редких случаях это может привести к мгновенному проигрышу в игре. Это все же лучше, чем реальные миссии, которые терпят неудачу в 47% случаев!
- **В проградации либо ретроградации спутника нет необходимости.** Постоянное вращение космического зонда перед запуском его двигателей значительно уменьшает игровой процесс. Будем считать, что направляющие двигатели расположены массивом вокруг фюзеляжа, и для выбора запускаемого двигателя вы будете использовать клавиши со стрелками.

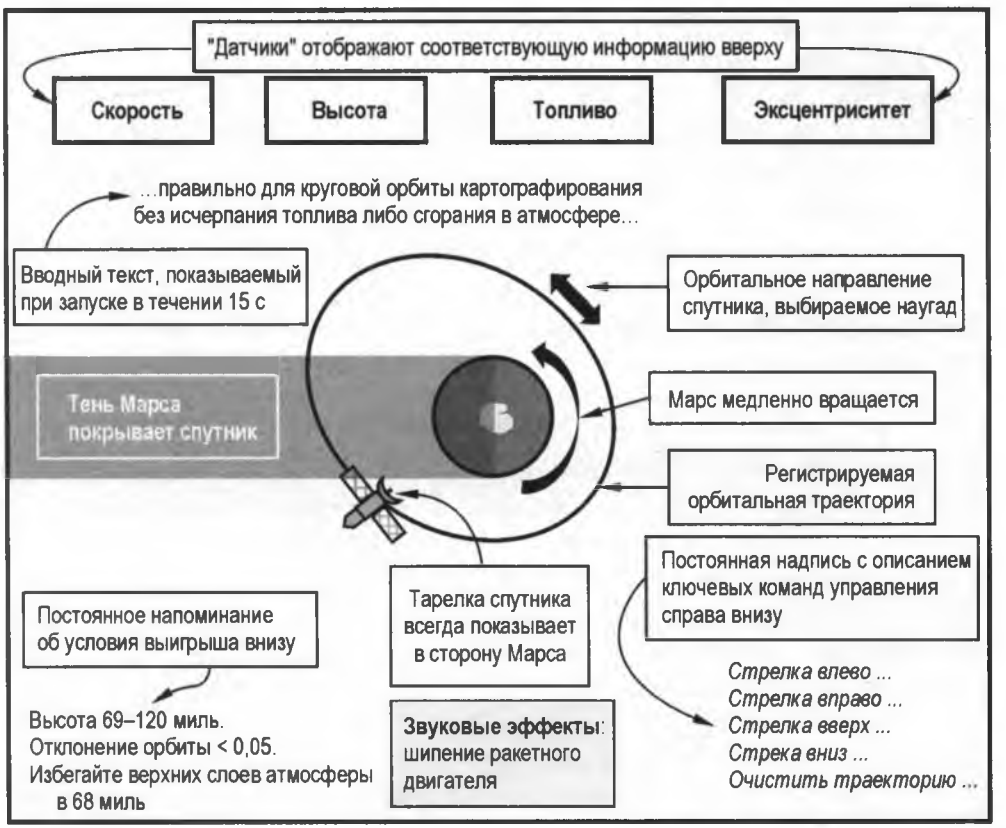


Рис. 14.11. Эскиз главного игрового процесса игры "Орбитальный спутник Марса"

- **Включение двигателей вызывает слышимое шипение.** Несмотря на то что в космосе звука нет, предоставьте игроку удовольствие в том, чтобы он слышал приятное на слух шипение, когда запускает двигатели.
- **Спутниковая тарелка всегда направлена на Марс.** Спутник будет медленно и автоматически вращаться так, чтобы его тарелка дистанционного зондирования всегда была направлена на Марс.
- **Видна орбитальная траектория спутника.** Тонкая белая линия будет выходить из-за спутника и сохраняться до тех, пока игрок не очистит ее, нажав клавишу <Пробел>.
- **Индикация данных размещается вверху экрана.** Вы будете показывать информацию, полезную для процесса игры в прямоугольниках вверху окна. Ключевыми данными являются скорость космического зонда, его высота, топливо и эксцентриситет орбиты (мера округлости орбиты).
- **Краткий инструктаж показывается при запуске.** Знакомящий с игрой текст появится в центре экрана, когда игра начнется, и будет оставаться примерно в течение 15 секунд. Указанный текст не будет нарушать игровой процесс, поэтому игрок может сразу же начать манипулировать спутником.
- **Условия выигрыша и ключевые элементы управления показываются в постоянных надписях.** Критическая информация, такая как цели полета и клавиши управления, будет постоянно изображена в левом и правом нижних углах экрана.

Эскиз игры на рис. 14.12 описывает, что происходит в случаях успеха и неудачи. Игроку нужно вознаграждение, когда он выигрывает, и интересный результат, когда проигрывает.

Для выигрышного и проигрышного результатов ключевыми моментами являются следующие:

- **Изменение изображения спутника для аварии и сгорания.** Если спутник снижается на высоту менее 68 миль, то он сгорает в атмосфере. Движущееся изображение спутника будет заменено его светящейся красной версией, которая прилипнет к поверхности Марса; это похоже на то, что можно увидеть на реальном дисплее управления полетами.
- **Спутник теряется в космосе, если у него заканчивается топливо.** Хотя это нереально, спутник улетает за пределы экрана и в космические глубины, если у него закончится топливо. Это будет зудеть над ухом игрока как назойливая муха!
- **Условия выигрыша разблокируют приз.** Если спутник достигает круговой орбиты в пределах целевого диапазона высот, то новый текст будет побуждать игрока нажать клавишу <M>.
- **Нажатие клавиши <M> изменяет изображение Марса.** Когда клавиша <M> разблокирована, ее нажатие приводит к изменению снимка Марса на радужное изображение, где холодные цвета представляют участки с высокой влажностью почвы, а теплые цвета — более сухие участки.

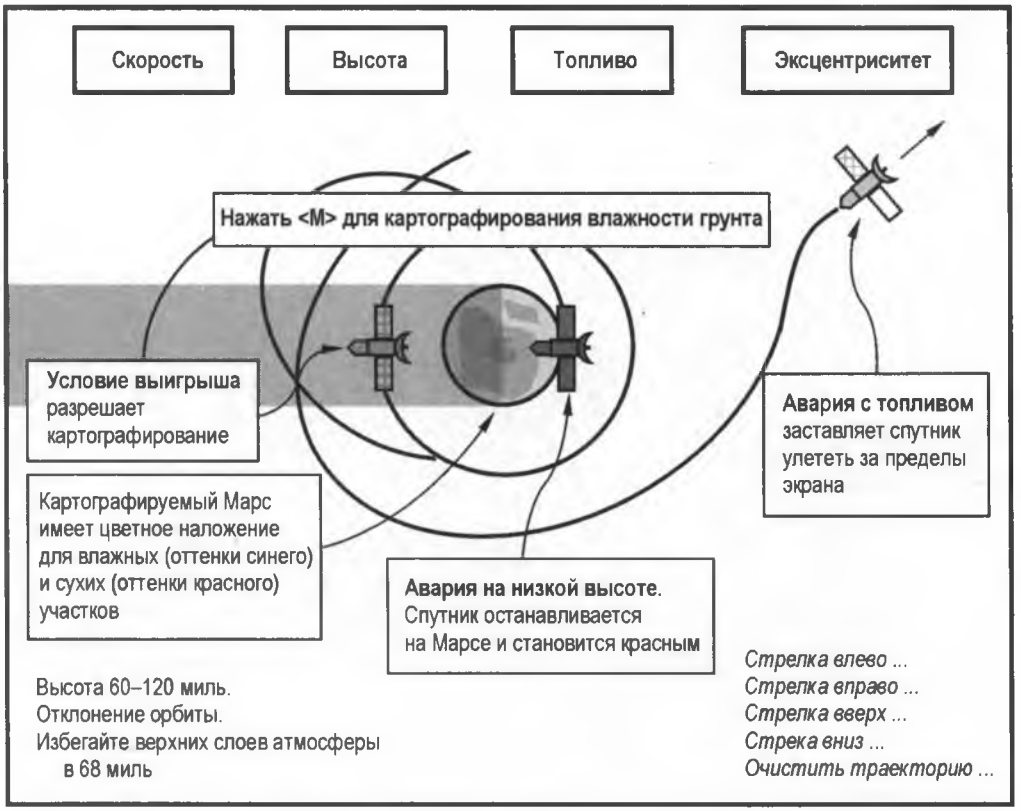


Рис. 14.12. Игровой эскиз выигрышного и проигрышного результатов в игре "Орбитальный спутник Марса"

Размер спутника и его орбитальная скорость не будут реалистичными для процесса игры, но общее поведение будет правильным. Вы должны суметь правильно выполнить все орбитальные маневры, описанные в *разд. "Аэродинамика для геймеров" ранее в этой главе*.

Активы игры

Для игры "Орбитальный спутник Марса" вам понадобятся активы. Это два изображения спутника, два снимка планеты и звуковой файл. Их можно заготовить сразу в начале процесса либо собрать, когда они вам понадобятся. Последний подход позволяет вам устраивать эпизодические перерывы в написании программного кода, которые отдельные разработчики предпочитают делать.

Поиск хороших, без авторских прав, графических и звуковых файлов может стать проблемой. Подходящие активы можно отыскать в Интернете бесплатно либо за отдельную плату, но лучше всего, когда это возможно, изготовить свои собственные. Это позволяет избежать каких-либо юридических проблем в будущем.

Спрайты (двухмерные значки или изображения), которые я использовал для этого проекта, показаны на рис. 14.13. Вам нужен спутник, красная "сожженная" версия

спутника, вид Марса с полярной шапкой по центру и тот же вид с красочным наложением, который будет представлять картографированные градации влажности почвы. Спрайт спутника я нашел на веб-сайте бесплатных иконок AHA-SOFT (**http://www.aha-soft.com/**), а затем скопировал его и перекрасил, для того чтобы сделать разбитую версию. Оба спрайта Марса — это модифицированные для игры снимки NASA.

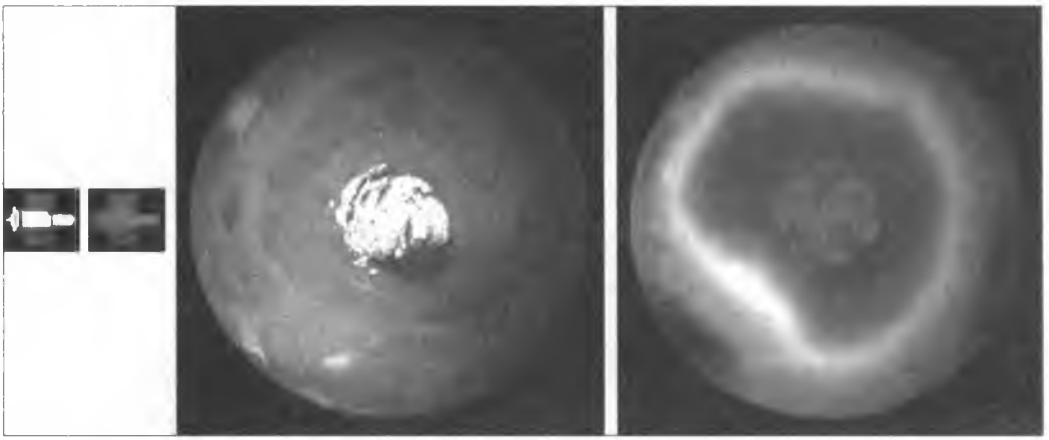


Рис. 14.13. Спутник, разбившийся спутник, снимок Марса и наложение на Марс, используемые в качестве игровых спрайтов

Я изготовил звуковой файл для ситуации, когда спутник запускает свои двигатели. Для этого я использовал генератор белого шума в программе с открытым исходным кодом Audacity. Бесплатную копию программы Audacity можно скачать по адресу **https://www.audacityteam.org/**. Я сохранил файл в формате Ogg Vorbis, стандартном формате сжатия звука с открытым исходным кодом, который является бесплатным и хорошо работает с Python и `pygame`. С пакетом `pygame` можно использовать и другие форматы, такие как MP3 и WAV, но некоторые из них имеют документированные проблемы либо имеют нативные компоненты, которые могут вызвать юридические проблемы, в случае если вы попытаетесь коммерциализировать свою игру.

Эти файлы можно скачать с веб-сайта книги по адресу **https://www.nostarch.com/impracticalpython/** как файлы satellite.png, satellite_crash_40x33.png, mars.png, mars_water.png и thrust_audio.ogg. Скачайте их, сохранив в ту же папку, что и код.

Код

Рис. 14.14 является примером финального игрового экрана, который вы будете строить. К этому рисунку можно регулярно возвращаться, для того чтобы иметь представление о том, что конкретно делает программный код.

Полную программу (mars_orbiter.py) можно скачать с **https://www.nostarch.com/impracticalpython/**.

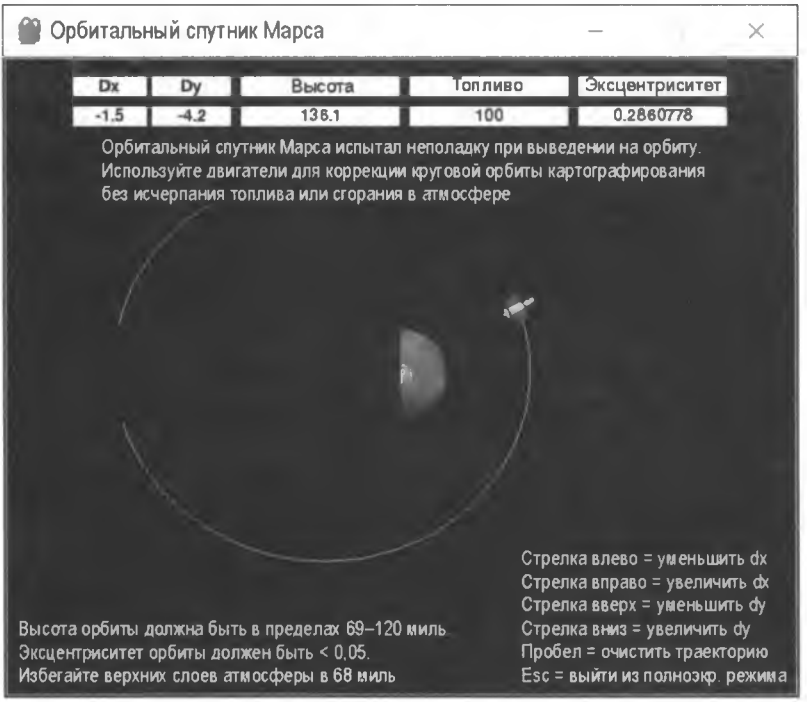


Рис. 14.14. Пример экрана при запуске игры в окончательной версии программы mars_orbiter.py

Импорт и построение цветовой таблицы

Листинг 14.1 импортирует необходимые модули и создает цветовую таблицу.

Листинг 14.1. Импортирует модули и строит цветовую таблицу. Файл mars_orbiter.py, часть 1

```
❶ import os
  import math
  import random
  import pygame as pg

❷ WHITE = (255, 255, 255)
  BLACK = (0, 0, 0)
  RED = (255, 0, 0)
  GREEN = (0, 255, 0)
  LT_BLUE = (173, 216, 230)
```

Сначала импортируйте модуль операционной системы, обозначаемый как `os` ❶. Игра запустится в полноэкранном режиме, но у игрока будет возможность из него выйти. Этот модуль позволит вам управлять расположением игрового окна после нажатия игроком клавиши <Esc>.

Модуль `math` будет использоваться для гравитационных и тригонометрических расчетов и случайного запуска спутника с произвольным местоположением и скоростью. Импортируйте пакет `pygame`, как вы сделали в *главе 13*, используя вместо `pygame` псевдоним `pg` с целью уменьшить печатание с клавиатуры.

Завершите построение цветовой таблицы RGB ❶, как это было сделано в *главе 13*. Это позволит вводить имена цветов вместо кортежей значений RGB, когда вам потребуется задавать один из этих цветов.

Определение метода инициализации класса *Satellite*

Листинг 14.2 определяет класс `Satellite` и его метод инициализации, который будет использоваться для инстанцирования объекта-спутника в игре. Поскольку это определение метода является длинным, оно разделено на два листинга.

Листинг 14.2. Определяет первую часть метода инициализации класса `Satellite`. Файл mars_orbiter.py, часть 2

```
❶ class Satellite(pg.sprite.Sprite):
      """Объект Satellite, который вращается повернутым
         к планете и падает и сгорает."""

   ❷ def __init__(self, background):
      ❸ super().__init__()
      ❹ self.background = background
      ❺ self.image_sat = pg.image.load("satellite.png").convert()
        self.image_crash = pg.image.load("satellite_crash_40x33.png").
                           convert()
      ❻ self.image = self.image_sat
      ❼ self.rect = self.image.get_rect()
      ❽ self.image.set_colorkey(BLACK) # устанавливает прозрачный цвет
```

Определите класс для объекта-спутника ❶; если вам необходимо обновить знания по ОПП, то прочитайте *главу 11*. Передайте ему класс `Sprite` пакета `pygame`, т. к. объекты, инстанцированные из класса `Satellite`, будут спрайтами. Как описано в *главе 13*, `Sprite` — это встроенный класс, который служит шаблоном для создания спрайтов. Ваш новый класс унаследует из этого базового класса функции, которые понадобятся вашим спрайтам. К ним относятся такие важные атрибуты, как `rect` и `image`, с которые вы вскоре будете работать.

Затем определите метод `__init__()` для объекта `Satellite` ❷ и передайте ему ключевое слово `self`, по соглашению являющееся специальным именем в определении класса, которое ссылается на текущий объект. Также необходимо передать методу фоновый объект `background`. На этом объекте будет нарисована траектория спутника.

Внутри метода `__init__()` сразу же вызовите метод инициализации для встроенного класса `Sprite` с использованием метода `super` ❸. Благодаря этому инициализируется спрайт и устанавливаются необходимые атрибуты объектов `rect` и `image`. Используя `super`, вам не нужно явно ссылаться на базовый класс (`Sprite`). Дополнительные сведения о методе `super` см. в листинге 11.5 или в документации на вебстранице **https://docs.python.org/3/library/functions.html?highlight=super#super**.

Далее назначьте `background` переменной `self` в качестве атрибута объекта ❹. Затем примените метод `image.load()` пакета `pygame` для загрузки двух ваших изображений спутника — один рабочий и другой аварийный — и на том же шаге выполните на них метод `convert()` ❺. Он сконвертирует объект в графический формат, который `pygame` может эффективно использовать после запуска игрового цикла. Без этого шага игра может заметно замедлиться, поскольку формат `png` конвертируется на лету более 30 раз в секунду.

Вы будете использовать только одно из двух изображений спутника за один раз, в зависимости от того, сгорел или нет игрок в атмосфере, поэтому для хранения загруженного и конвертированного изображения используйте обобщенный атрибут `self.image` ❻. Изображение несгоревшего спутника будет использоваться по умолчанию; оно будет заменено красным изображением разбившегося спутника, если объект-спутник окажется слишком близко к Марсу.

Теперь получите для изображения информацию о прямоугольнике ❼. Напомним, что пакет `pygame` помещает спрайты на объекты прямоугольной поверхности, и во время игры он должен знать размеры и расположение этих прямоугольников.

Наконец, сделайте черные части изображения спутника невидимыми ❽. Значок спутника находится на черном поле (см. рис. 14.13), и нужно, чтобы разбитое и сгоревшее изображение частично попало на изображение Марса, поэтому с методом `colorkey()` объекта-изображения используйте константу `BLACK`, для того чтобы сделать фон значка прозрачным. В противном случае вы увидите черный прямоугольник с наложением красного спутника на красную планету. Обратите внимание, что если вы хотите набрать эквивалент RGB для черного цвета, то вам нужно ввести его как кортеж: `(0, 0, 0)`.

Задание первоначального положения спутника, его скорости, топлива и звука

Листинг 14.3 завершает определение метода инициализации класса `Satellite`. Первоначальное положение и скорость объекта-спутника выбираются случайно из ограниченного диапазона вариантов; инициализируется ориентация тарелки дистанционного зондирования, заправляется топливный бак, и добавляются звуковые эффекты.

Листинг 14.3. Завершает метод инициализации класса `Satellite` путем инициализации параметров. Файл mars_orbiter.py, часть 3

```
❶ self.x = random.randrange(315, 425)
  self.y = random.randrange(70, 180)
```

```
❷ self.dx = random.choice([-3, 3])
❸ self.dy = 0
❹ self.heading = 0  # инициализирует ориентацию тарелки
❺ self.fuel = 100
  self.mass = 1
  self.distance = 0 # инициализирует расстояние между спутником
                    # и планетой
❻ self.thrust = pg.mixer.Sound('thrust_audio.ogg')
❼ self.thrust.set_volume(0.07) # допустимые значения: 0-1
```

При запуске игры спутник появится в случайной точке в верхней части экрана. Вы выберете точное местоположение из диапазона значений *x* и *y* ❶.

Вы также выберите наугад скорость спутника, но она будет достаточно медленной для того, чтобы спутник не смог уйти с орбиты. Случайно установите скорость, равной –3 или 3. Отрицательные значения приводят к вращению против часовой стрелки, и наоборот. Используйте только атрибут `dx` (дельта *x*) ❷ и дайте гравитации самой позаботиться о `dy`. Как обсуждалось в *главе 13*, пакет `pygame` перемещает спрайты по экрану, используя инкрементные изменения в местоположении *x* (именуемые дельта *x* или *dx*) и инкрементные изменения в местоположении *y* (именуемые дельта *y* или *dy*). Эти векторные компоненты вычисляются и добавляются к текущей позиции спрайта (`self.x`, `self.y`) с каждым игровым циклом.

Далее установите атрибут `dy` равным `0` ❸. Позже метод `gravity()` установит первоначальное значение `dy`, когда он будет ускорять только что инстанцированный спутник вниз экрана в сторону планеты.

Задайте атрибут направления полета (`heading`) спутника ❹. Тарелка дистанционного зондирования, которая будет считывать влажность почвы на поверхности планеты, всегда должна быть направлена на Марс, и если вы помните из рис. 14.3, то это не произойдет, если вы не преодолеете инерцию. Вы будете использовать метод, который будет фактически поворачивать спутник, поэтому на данный момент просто инициализируйте атрибут `heading` значением 0.

Теперь долейте в топливный бак 100 единиц топлива ❺. Если вы хотите связать эту процедуру с реальной жизнью, то она, вероятно, будет представлять собой заливку 100 кг гидразина, подобного использовавшемуся в зонде Магеллан, который занимался картографированием Венеры.

Далее установите массу объекта, равной 1. Она, в сущности, означает, что в уравнении гравитации вы будете использовать только массу Марса, потому что вы перемножаете массы двух объектов между собой. Как было сказано ранее, притяжение спутника на Марсе является несущественным, поэтому его вычислять не требуется. Атрибут `mass` спутника включен для полноты и в качестве заполнителя в случае, если позже вы захотите поэкспериментировать с другими значениями.

Следующий атрибут `distance` хранит расстояние между спутником и телом, вокруг которого он вращается. Фактическое значение будет рассчитываться методом, который вы определите позже.

Пришло время добавить звуковые эффекты. Звуковой микшер пакета `pygame` инициализируется в функции `main()`, но пока объявите атрибут `thrust` для звукового эффекта при создании тяги двигателями ❻. Передайте классу `Sound` микшера короткий клип белого шума в формате Ogg Vorbis (.ogg). Наконец, установите громкость воспроизведения, используя значения от 0 до 1 ❼. Возможно, на вашем компьютере потребуется его откалибровать. В идеале вам нужно значение, которое каждый игрок сможет, по крайней мере, услышать, а затем точно настроить с помощью регулятора громкости своего компьютера.

Запуск двигателей и проверка вводимых игроком значений

В листинге 14.4 определены методы `thruster()` и `check_keys()` класса `Satellite`. Первый определяет действия, предпринимаемые в случае срабатывания одного из двигателей спутника. Второй проверяет, не взаимодействовал ли игрок с двигателями, нажимая клавишу со стрелкой.

Листинг 14.4. Определяет методы `thruster()` и `check_keys()` для класса `Satellite`. Файл mars_orbiter.py, часть 4

```
❶ def thruster(self, dx, dy):
      """Исполнить действия, ассоциированные с запуском двигателей."""
   ❷ self.dx += dx
      self.dy += dy
   ❸ self.fuel -= 2
   ❹ self.thrust.play()

❺ def check_keys(self):
      """Проверить нажатие пользователем клавиш со стрелкой
         и вызвать метод thruster()."""
   ❻ keys = pg.key.get_pressed()
      # запустить двигатели
   ❼ if keys[pg.K_RIGHT]:
      ❽ self.thruster(dx=0.05, dy=0)
      elif keys[pg.K_LEFT]:
          self.thruster(dx=-0.05, dy=0)
      elif keys[pg.K_UP]:
          self.thruster(dx=0, dy=-0.05)
      elif keys[pg.K_DOWN]:
          self.thruster(dx=0, dy=0.05)
```

Метод `thruster()` в качестве аргументов принимает `self`, `dx` и `dy` ❶. Последние два аргумента, которые могут быть положительными либо отрицательными, немедленно добавляются в компоненты `self.dx` и `self.dy` скорости спутника ❷. Далее уровень топлива снижается на две единицы ❸. Изменение этого значения является

одним из способов усложнить либо упростить игру. Завершите метод вызовом метода `play()` на аудиоатрибуте `thrust`, который создаст шипящий звук ❹. Обратите внимание, что вместо возврата значений методы ООП обновляют существующие объектные атрибуты.

Метод `check_keys()` в качестве аргумента принимает ключевое слово `self` ❺. Сначала вы используете модуль `key` пакета `pygame`. Это делается для того, чтобы определить, нажал или нет игрок клавишу ❻. Метод `get_pressed()` возвращает кортеж булевых значений — 1 для `True` и 0 для `False`, которые представляют текущее состояние каждой клавиши на клавиатуре. `True` означает, что клавиша нажата. Указанный кортеж может индексироваться с помощью констант клавиш. Список всех констант клавиатуры можно найти по адресу **https://www.pygame.org/docs/ref/key.html**.

Например, клавиша со стрелкой вправо (<→>) равна `K_RIGHT`. Если эта клавиша была нажата ❼, то вызовите метод `thruster()` и передайте ему значения `dx` и `dy` ❽. В пакете `pygame` значения x увеличиваются в правой части экрана, а значения y — в нижней. Поэтому если пользователь нажимает клавишу со стрелкой влево (<←>), то вычтите из `dx`; схожим образом, если нажата клавишу со стрелкой вверх (<↑>), то уменьшите значение `dy`. Стрелка вправо увеличит `dx`, а стрелка вниз (<↓>) увеличит `dy`. Показания в верхней части экрана помогут игроку соотнести движения спутника с базовыми значениями `dx` и `dy` (см. рис. 14.14).

Локализация спутника

По-прежнему находясь в классе `Satellite`, листинг 14.5 определяет метод `locate()`. Указанный метод вычисляет расстояние спутника от планеты и определяет направление полета для наведения антенны на планету. Атрибут `distance` будет использоваться позже при расчете силы гравитации и эксцентриситета орбиты. Эксцентриситет — это мера отклонения орбиты от идеальной окружности.

Листинг 14.5. Определяет метод `locate()` для класса `Satellite`. Файл mars_orbiter.py, часть 5

```
❶ def locate(self, planet):
      """Вычислить расстояние и направленность к планете."""
   ❷ px, py = planet.x, planet.y
   ❸ dist_x = self.x - px
      dist_y = self.y - py
      # получить направление к планете, для того чтобы направить тарелку
   ❹ planet_dir_radians = math.atan2(dist_x, dist_y)
   ❺ self.heading = planet_dir_radians * 180 / math.pi
   ❻ self.heading -= 90 # sprite is traveling tail-first
   ❼ self.distance = math.hypot(dist_x, dist_y)
```

Для локализации спутника вам нужно передать методу `locate()` объекты `Satellite` (`self`) и `planet` ❶. Сначала определите расстояние между объектами в пространст-

ве *x*-*y*. Получите атрибуты *x* и *y* планеты ❷; затем вычтите их из атрибутов *x* и *y* спутника ❸.

Теперь используйте эти новые переменные расстояния для расчета угла между направлением полета спутника и планетой с тем, чтобы можно было повернуть спутниковую антенну к планете. Модуль `math` использует радианы, поэтому задайте локальную переменную `planet_dir_radians` для хранения направления в радианах и передайте `dist_x` и `dist_y` в функцию `math.atan2()`, которая вычислит арктангенс ❹. Поскольку `pygame` использует градусы (увы), конвертируйте угол из радиан в градусы, используя стандартную формулу; для этого в качестве альтернативы можно применить модуль `math`, но иногда неплохо видеть человека за занавесом ❺. Этот атрибут объекта-спутника должен быть совместным, и поэтому назовите его `self.heading`.

В `pygame` фронт спрайта по умолчанию находится на востоке, т. е. спрайт-спутник вращается хвостом вперед (см. значок спутника на рис. 14.13). Для того чтобы установить тарелку так, чтобы она была направлена на Марс, вам нужно вычесть 90° из направления полета, потому что в пакете `pygame` отрицательные углы приводят к вращению по часовой стрелке ❻. Этот маневр не будет задействовать ни один из топливных ресурсов игрока.

Наконец, получите евклидово расстояние между спутником и Марсом, используя модуль `math` для вычисления гипотенузы из компонент *x* и *y* ❼. Вы должны сделать его атрибутом объекта-спутника, т. к. оно будет использоваться позже в других функциях.

ПРИМЕЧАНИЕ

В реальной жизни удерживать тарелку спутника направленной к планете можно несколькими способами, не затрачивая большого объема топлива. Эти способы включают в себя медленное кувыркание или вращение спутника, делая конец тарелки тяжелее, чем противоположный конец, используя магнитный момент или внутренние маховики — так называемые реактивные колеса, или импульсные колеса. Маховики задействуют электродвигатели, которые могут работать от солнечных батарей, устраняя необходимость в тяжелом и токсичном жидком топливе.

Вращение спутника и вычерчивание его орбиты

Листинг 14.6 продолжает класс `Satellite`, определяя методы поворота спутниковой антенны к планете и нанесения его траектории позади. Позже, в функции `main()`, вы добавите программный код, который позволяет игроку стирать и перезапустить траекторию, нажимая клавишу <Пробел>.

Листинг 14.6. Определяет методы `rotate()` и `path()` класса `Satellite`. Файл mars_orbiter.py, часть 6

```
❶ def rotate(self):
      """Вращать спутник, используя градусы так, чтобы тарелка
         была обращена к планете."""
   ❷ self.image = pg.transform.rotate(self.image_sat, self.heading)
   ❸ self.rect = self.image.get_rect()
```

```
❹ def path(self):
      """Обновить позицию спутника и нанести линию
         для прослеживания орбитальной траектории."""
    ❺ last_center = (self.x, self.y)
    ❻ self.x += self.dx
      self.y += self.dy
    ❼ pg.draw.line(self.background, WHITE, last_center, (self.x, self.y))
```

Метод `rotate()` будет использовать атрибут `heading`, который вы вычисляете в методе `locate()` для поворота спутниковой антенны в сторону Марса. Передайте ключевое слово `self` в методе `rotate()` ❶. Это означает, что метод `rotate()` автоматически примет имя объекта-спутника в качестве аргумента, когда он будет вызван позже.

Теперь поверните изображение спутника, используя метод `transform.rotate()` пакета `pygame` ❷. Передайте ему исходное изображение и затем атрибут `heading`; назначьте результат атрибуту `self.image`, для того чтобы не ухудшить исходное главное изображение. С каждым игровым циклом вам нужно будет преобразовывать изображение, а преобразование изображения быстро ухудшает его качество. Поэтому всегда держите главное изображение про запас и создавайте новую копию всякий раз, когда выполняете преобразование.

Завершите функцию, получив объект `rect` преобразованного изображения ❼. Далее следует определить метод с именем `path()` и передать ему ключевое слово `self` ❹. Он нарисует линию, отмечающую траекторию спутника, и поскольку для нанесения линии вам нужны две точки, задайте переменную, регистрирующую местоположение центра спутника в виде кортежа до его перемещения ❺. Затем увеличьте местоположения *x* и *y* на величину атрибутов `dx` и `dy` ❻. Завершите метод, используя метод `draw.line()` пакета `pygame` ❼. Этот метод нуждается в объекте-рисунке, поэтому передайте ему атрибут `background`, а затем цвет линии и кортежи предыдущего и текущего местоположений *x*-*y*.

Обновление объекта *Satellite*

Листинг 14.7 обновляет объект-спутник и завершает определение класса. Объекты класса `Sprite` почти всегда имеют метод `update()`, который при выполнении игры вызывается один раз за кадр. Все, что происходит со спрайтом, например движение, изменение цвета, взаимодействие с пользователем и т. д., включается в этот метод. Во избежание излишней загроможденности методы `update()` в основном вызывают другие методы.

Листинг 14.7. Определяет метод `update()` для класса `Satellite`. Файл mars_orbiter.py, часть 7

```
❼ def update(self):
      """Обновлять объект-спутник во время игры."""
    ❷ self.check_keys()
```

```
❸ self.rotate()
❹ self.path()
❺ self.rect.center = (self.x, self.y)
   # изменить изображение на огненно-красное, если спутник
   # находится в атмосфере
❻ if self.dx == 0 and self.dy == 0:
       self.image = self.image_crash
       self.image.set_colorkey(BLACK)
```

Начните с определения метода `update()` и передачи ему объекта, или ключевого слова `self` ❶. Затем вызовите определенные ранее методы. Первый из них выполняет проверку наличия взаимодействия со стороны игрока, осуществляемую через клавиатуру ❷. Второй вращает объект-спутник так, чтобы его тарелка продолжала указывать на планету ❸. Последний метод обновляет местоположение *x-y* спутника и наносит позади него траекторию с целью визуализации его орбиты ❺.

Программа должна отслеживать местоположение спрайта спутника, когда он вращается вокруг Марса, поэтому задайте атрибут `rect.center` и установите его равным текущему местоположению *x-y* спутника ❺.

Последний фрагмент кода изменяет изображение спутника в случае, если игрок падает и сгорает в атмосфере ❻. Верхние слои марсианской атмосферы находятся примерно в 68 милях над ее поверхностью. По причинам, которые я объясню позже, будем считать, что значение высоты 68 миль (которое измеряется в пикселах от центра планеты) приравнивается к верхним слоям атмосферы. Если во время игры спутник опускается ниже этой высоты, то функция `main()` установит его скорость, представленную значениями `dx` и `dy`, равной 0. Проверьте, что эти значения равны 0, и если это так, то измените изображение на `image_crash` и установите его фон прозрачным (как вы делали ранее для главного изображения спутника).

Определение метода инициализации класса *Planet*

В листинге 14.8 определен класс `Planet`, который будет использоваться для инстанцирования объекта `planet`.

Листинг 14.8. Начинает определение класса `Planet`. Файл mars_orbiter.py, часть 8

```
❶ class Planet(pg.sprite.Sprite):
      """Объект-планета, который вращается и проецирует
         гравитационное поле."""

   ❷ def __init__(self):
         super().__init__()
      ❸ self.image_mars = pg.image.load("mars.png").convert()
         self.image_water = pg.image.load("mars_water.png").convert()
      ❹ self.image_copy = pg.transform.scale(self.image_mars, (100, 100))
```

```
❺ self.image_copy.set_colorkey(BLACK)
❺ self.rect = self.image_copy.get_rect()
  self.image = self.image_copy
❼ self.mass = 2000
❽ self.x = 400
  self.y = 320
  self.rect.center = (self.x, self.y)
❺ self.angle = math.degrees(0)
  self.rotate_by = math.degrees(0.01)
```

Вы, вероятно, уже хорошо знакомы с первыми шагами по созданию класса `Planet`. Прежде всего, вы именуете класс с большой буквы, а затем передаете ему класс `Sprite` с тем, чтобы он удобно наследовал функции у этого встроенного в пакет `pygame` класса ❶. Далее вы определяете метод `__init__()`, или метод инициализации вашего объекта `planet` ❷. Затем, как и для класса `Satellite`, вы вызываете метод инициализации `super()`.

Загрузите снимки в качестве атрибутов и одновременно конвертируйте их в графический формат пакета `pygame` ❸. Вам нужен как обычный снимок Марса, так и снимок картографированной влажности грунта. Вы могли бы использовать спрайт спутника в его естественных размерах, но снимок Марса является слишком крупным. Отмасштабируйте снимок в 100×100 пикселов ❹ и назначьте масштабированный снимок новому атрибуту, с тем чтобы повторные преобразования не ухудшали главный снимок.

Теперь сделайте прозрачный цвет преобразованного снимка черным, как вы делали ранее с изображением спутника ❺. Все спрайты в `pygame` "монтируются" на прямоугольных поверхностях, и если вы не сделаете черный цвет невидимым, то углы поверхности планеты могут накладываться и покрывать белую орбитальную траекторию, наносимую спутником (рис. 14.15).

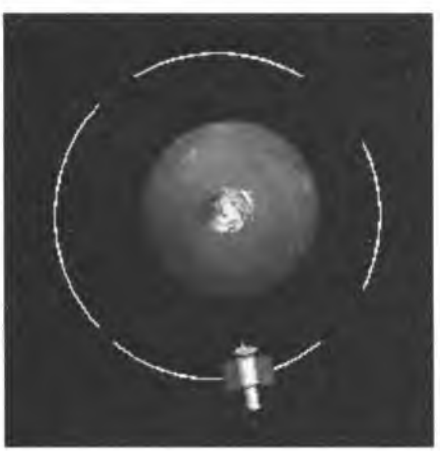


Рис. 14.15. Углы прямоугольника `rect` Марса, накладывающиеся на орбитальную траекторию

Как всегда, получите объект `rect` спрайта ❻. Впереди будет еще одно преобразование, поэтому скопируйте атрибут снимка еще раз и назначьте ему логическое имя `self.image`.

Для того чтобы применить силу гравитации, планете нужна масса, поэтому объявите атрибут `mass` и назначьте ему значение `2000` ❼. Ранее вы назначили спутнику массу `1`; это означает, что Марс лишь в 2000 раз массивнее спутника! Это нормально, потому что вы не работаете в реальных единицах измерения, и шкала времени, и расстояния отличаются от реальности. Если расстояния шкалируются так, что спутник находится всего в нескольких сотнях пикселов от Марса, то также должна шкалироваться и гравитация. Невзирая на это, спутник все равно будет вести себя реалистично по отношению к гравитации.

Значение массы планеты было определено экспериментально. Для шкалирования силы гравитации можно либо изменить величину массы, либо позже применить величину гравитационной постоянной (`G`).

Установите атрибуты `x` и `y` объекта `planet` равными центральной точке экрана — вы задействуете размер экрана 800×645 в функции `main()` — и назначьте эти значения центру объекта `rect` ❽.

Наконец, задайте атрибуты, которые вам понадобятся для медленного вращения Марса вокруг своей оси ❾. Вы будете использовать тот же метод `transform.rotate()`, который вы применяли для поворота спутника, поэтому вам нужно создать атрибут угла `angle`. Затем используйте атрибут `rotate_by` для назначения приращения в градусах, на которое этот угол поворота изменяется с каждым игровым циклом.

Вращение планеты

Листинг 14.9 продолжает класс `Planet`, определяя его метод `rotate()`. Этот метод вращает планету вокруг своей оси, внося небольшие изменения с каждым игровым циклом.

Листинг 14.9. Определяет метод вращения планеты вокруг своей оси. Файл mars_orbiter.py, часть 9

```
❶ def rotate(self):
      """Вращать снимок планеты с каждым игровым циклом."""
   ❷ last_center = self.rect.center
   ❸ self.image = pg.transform.rotate(self.image_copy, self.angle)
     self.rect = self.image.get_rect()
   ❹ self.rect.center = last_center
   ❺ self.angle += self.rotate_by
```

Метод `rotate()` в качестве аргумента также принимает объект ❶. Поскольку квадратный снимок Марса вращается, объект `rect` (ограничительный прямоугольник) остается стационарным и должен расширяться для размещения новой конфигура-

ции (рис. 14.16). Это изменение размера может повлиять на центральную точку объекта `rect`, поэтому задайте переменную предыдущего центра `last_center` и установите ее равной текущей центральной точке планеты ❷. Если вы этого не сделаете, то во время игры Марс будет колебаться вокруг своей оси.

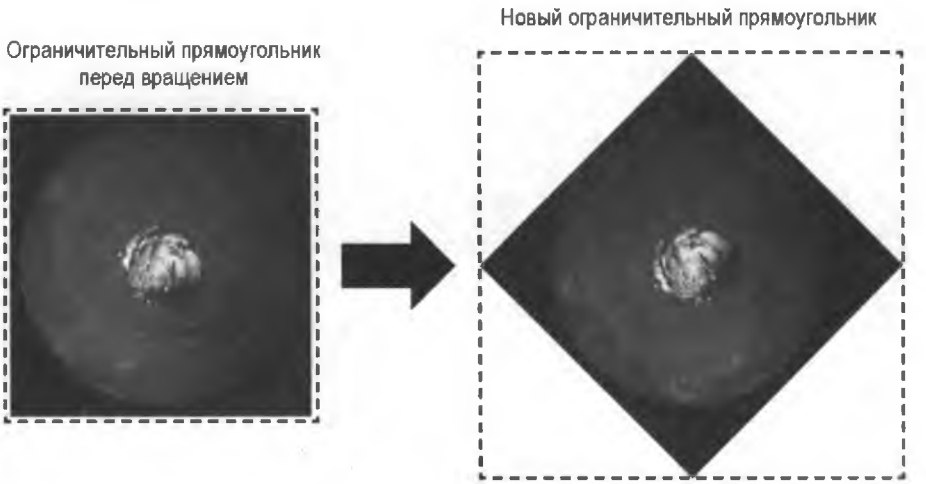


Рис. 14.16. Ограничительный прямоугольник изменяет размер для размещения вращающихся снимков

Затем поверните скопированный снимок с помощью метода `transform.rotate()` пакета `pygame` и назначьте его атрибуту `self.image` ❸; указанному методу необходимо передать скопированный снимок и атрибут `angle`. Сразу после поворота сбросьте атрибут `rect` снимка и переместите его центральное местоположение обратно в `last_center`, для того чтобы смягчить любое смещение `rect`, которое произошло во время поворота ❹.

Когда инстанцируется объект `planet`, атрибут угла будет начинаться с 0°, а затем увеличиваться на величину, равную `0.1`, назначаемую в атрибуте `rotate_by`, с каждым кадром ❺.

Определение методов *gravity()* и *update()*

Листинг 14.10 завершает класс `Planet`, определяя методы `gravity()` и `update()`. В *главе 13* вы рассматривали гравитацию как константу, приложенную в направлении *y*. Применяемый здесь метод несколько сложнее, поскольку учитывает расстояние между двумя объектами.

Листинг 14.10. Определяет методы `gravity()` и `update()` класса `Planet`. Файл mars_orbiter.py, часть 10

```
❶ def gravity(self, satellite):
      """Вычислить воздействие гравитации на спутник."""
   ❷ G = 1.0    # гравитационная постоянная для игры
```

```
        dist_x = self.x - satellite.x
        dist_y = self.y - satellite.y
        distance = math.hypot(dist_x, dist_y)
        # нормализовать в единичный вектор
        dist_x /= distance
        dist_y /= distance
        # приложить гравитацию
        force = G * (satellite.mass * self.mass) / (math.pow(distance, 2))
        satellite.dx += (dist_x * force)
        satellite.dy += (dist_y * force)

    def update(self):
        """вызвать метод rotate."""
        self.rotate()
```

Определите метод `gravity()` и передайте ему ключевое слово `self` и объект-спутник ❶. Вы по-прежнему находитесь в классе `Planet`, поэтому ключевое слово `self` здесь представляет Марс.

Начните с объявления локальной переменной `G`; заглавная `G` — это универсальная гравитационная постоянная, также именуемая константой пропорциональности ❷. В реальной жизни это очень малое, эмпирически полученное число, которое, по сути дела, представляет собой конверсионное число для того, чтобы единицы измерения работали правильно. В игре реальные единицы измерения не используются, поэтому установите ее равной 1; благодаря этому она не повлияет на уравнение гравитации. Во время разработки игры вы можете скорректировать эту константу вверх или вниз в целях точной настройки гравитации и ее влияния на орбитальные объекты.

Вам нужно знать, как далеко друг от друга находятся два объекта, поэтому получите их расстояние в направлении *x* и направлении *y* ❸. Затем примените метод `hypot()` модуля `math` для получения евклидова расстояния. В уравнении гравитации оно будет представлять *r*.

Поскольку вы собираетесь обратиться к магнитуде расстояния между спутником и Марсом в уравнении гравитации напрямую, то от вектора расстояния вам нужно получить только направление. Поэтому разделите `dist_x` и `dist_y` на расстояние `distance`, для того чтобы "нормализовать" вектор в единичный вектор с магнитудой, равной 1 ❹. Вы, в сущности, делите длину каждой стороны прямоугольного треугольника на его гипотенузу. Это оставляет нетронутым направление вектора, представленное относительными разницами в `dist_x` и `dist_y`, но устанавливает его магнитуду равной 1. Обратите внимание, что если не выполнить этот шаг нормализации, то результаты будут нереалистичными, но интересными (рис. 14.17).

Вычислите силу гравитации с помощью уравнения Ньютона, которое я описал в *разд. "Закон универсальной гравитации (или закон всемирного тяготения)" ранее в этой главе* ❺. Закончите умножением нормализованных расстояний на силу

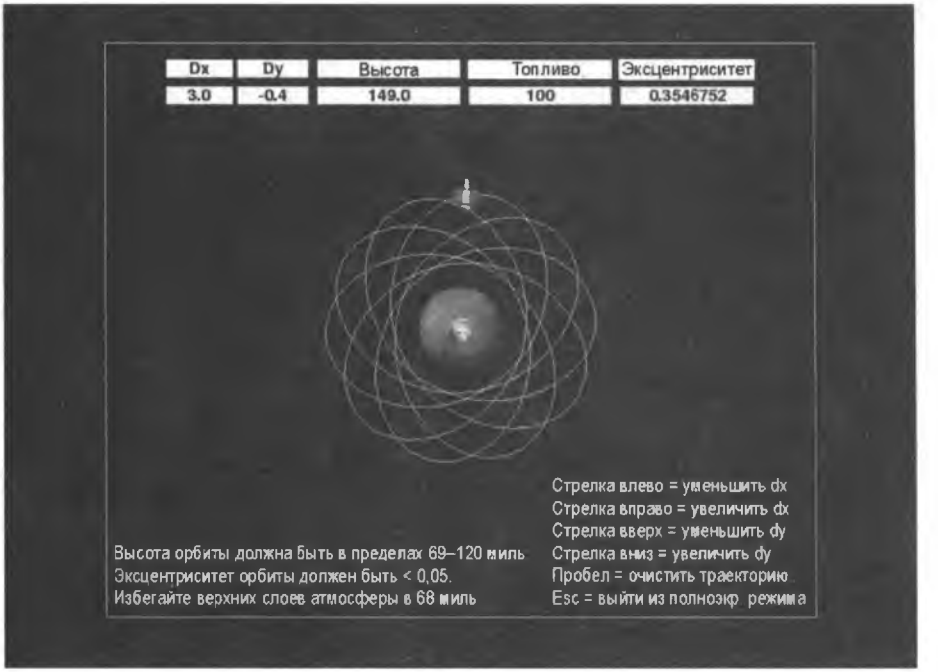


Рис. 14.17. "Спирографическая" орбита, полученная в результате использования ненормированного вектора расстояния

`force` (для того чтобы вычислить, насколько ускорение изменяет скорость на каждом шаге) и добавьте эти величины в атрибуты `dx` и `dy` объекта-спутника ❻.

Обратите внимание, что большинство этих переменных не задаются в качестве атрибутов параметра `self`. Они просто представляют промежуточные шаги внутри метода, которые не нужно использовать совместно с другими методами, и их можно рассматривать как локальные переменные процедурного программирования.

Наконец, определите метод, который будет вызываться каждый игровой цикл для обновления объекта `planet` ❼. Используйте его для вызова метода `rotate()`.

Расчет эксцентриситета

Вы закончили определение классов. Теперь самое время определить несколько функций, которые помогут с выполнением игры. В листинге 14.11 определена функция для вычисления эксцентриситета орбиты спутника. Игроку нужно добиться круговой орбиты в определенном диапазоне высот, и эта функция обеспечивает измерение округлости.

Листинг 14.11. Определяет функцию, измеряющую эксцентриситет орбиты. Файл mars_orbiter.py, часть 11

```
❶ def calc_eccentricity(dist_list):
      """Вычислить и вернуть эксцентриситет из списка радиусов."""
   ❷ apoapsis = max(dist_list)
```

```
    periapsis = min(dist_list)
❸ eccentricity = (apoapsis - periapsis) / (apoapsis + periapsis)
    return eccentricity
```

Определите функцию `calc_eccentricity()` и передайте ей список расстояний ❷. В функции `main()` во время каждого игрового цикла вы будете добавлять атрибут `sat.distance` — который регистрирует высоту спутника — в этот список. Для вычисления эксцентриситета нужно знать как апоапсиду, так и периапсиду орбиты. Получите их, найдя в этом списке максимальное и минимальное значения ❷. Затем вычислите эксцентриситет ❸. Позже, в функции `main()`, вы будете выводить это число на экран до восьми знаков после точки, просто чтобы показания датчиков выглядели круто и точно.

Обратите внимание, что круговые орбиты будут иметь одинаковое значение как для апоапсиды, так и для периапсиды, и поэтому для идеального круга указанный расчет даст 0. Завершите функцию, вернув переменную `eccentricity`.

Определение функций для создания надписей

В игре потребуется изрядное количество текста для инструктажа и показаний телеметрии. Вывод этого текста на экран по одной строке за раз может привести к избыточности программного кода, поэтому в листинге 14.12 будут определены две функции — одна для публикации инструктажа, а другая для потоков данных о скорости, высоте, топливе и эксцентриситете, которыми вы должны будете делиться с игроком.

Листинг 14.12. Определяет функции для создания надписей с инструктажем и показаниями датчиков. Файл mars_orbiter.py, часть 12

```
❶ def instruct_label(screen, text, color, x, y):
      """Принять экран, список символьных цепочек, цвет и
         начало координат и визуализировать текст на экране."""
   ❷ instruct_font = pg.font.SysFont(None, 25)
   ❸ line_spacing = 22
   ❹ for index, line in enumerate(text):
         label = instruct_font.render(line, True, color, BLACK)
         screen.blit(label, (x, y + index * line_spacing))

❺ def box_label(screen, text, dimensions):
      """Сделать надпись фиксированного размера из параметров
         экрана, текста и левой верхней позиции, ширины, высоты."""
      readout_font = pg.font.SysFont(None, 27)
   ❻ base = pg.Rect(dimensions)
   ❼ pg.draw.rect(screen, WHITE, base, 0)
   ❼ label = readout_font.render(text, True, BLACK)
   ❾ label_rect = label.get_rect(center=base.center)
   ❹ screen.blit(label, label_rect)
```

Определите функцию `instruct_label()` для изображения инструктажа на игровом экране ❶. Передайте ему экран, список, содержащий текст, цвет текста и координаты левого верхнего угла объекта `surface` пакета `pygame`, который будет содержать текст.

Далее сообщите пакету `pygame`, какой шрифт использовать ❷. Аргументами метода `font.SysFont()` являются шрифт и размер. Использование для шрифта `None` вызывает встроенный в `pygame` стандартный шрифт, который должен работать на многочисленных платформах. Обратите внимание, что указанный метод принимает как `None`, так и `'None'`.

Вводный текст и текст инструктажа занимают несколько строк (см. пример на рис. 14.14). Вам нужно будет указать межстрочный интервал в пикселах между строками текста, поэтому объявите для этого переменную и установите ее равной `22` ❸.

Теперь начните перебирать список символьных цепочек с текстом в цикле ❹. Используйте встроенную функцию `enumerate()`. Она будет возвращать индекс, который вы будете использовать с переменной `line_spacing` для размещения строк в правильных местах. Текст нужно будет разместить на поверхности. Назовите эту поверхность `label`, передайте методу `font.render()` строку текста, которую вы хотите показать. Для получения более гладкого текста установите сглаживание равным `True`, раскрасьте текст и сделайте цвет фона черным. Завершите функцию, перенеся поверхность на экран. Передайте методу переменную `label` и координаты левого верхнего угла, при этом `y` определяется как `y + index * line_spacing`.

Далее определите функцию `box_label()` для надписей с показаниями. Указанные показания будут отображаться в виде индикаторов датчиков в верхней части экрана (рис. 14.18) ❺. Параметрами для этой функции является экран, немного текста и кортеж, содержащий размеры прямоугольной поверхности, которая образует датчик.

Dx	Dy	Высота	Топливо	Эксцентриситет
0.1	-3.4	158.7	100	0.20803277

Рис. 14.18. Надписи с показаниями датчиков в верхней части игрового окна (заголовочная надпись выше и индикаторная надпись ниже)

Поверхности, созданные функцией `instruct_label()`, будут автоматически изменять размер, списывая объем выводимого на экран текста. Это прекрасно работает для статических дисплеев, но показания будут постоянно меняться, расширяя и сжимая ваши индикаторы датчиков во время их корректировки по размеру текста внутри них. Для смягчения этого эффекта вы будете использовать автономный объект `rect` заданного размера, формируя базу для вашего текстового объекта.

Начните функцию с установки шрифта, как вы сделали в пункте ❷. Назначьте переменную `base` объекту `rect` пакета `pygame`; для размера используйте аргумент `dimensions` ❻. Этот аргумент позволяет точно определять положение прямоугольника, указав левую и верхнюю координаты прямоугольника, а затем его ширину и

высоту. Результирующий прямоугольник должен быть широким настолько, чтобы обрабатывать максимально длинное показание датчика, которое игра будет производить для выводимого на экран типа данных.

Теперь нарисуйте `base` с помощью метода `draw_rect()` ❼. Его аргументами являются поверхность рисунка, цвет заливки, имя прямоугольника `rect` и ширина `0`, которая заполняет прямоугольник, но не рисует границу. Вы разместите свой текстовый объект поверх этого белого прямоугольника.

Повторите код, который выводит текст ❽, затем получите прямоугольник для надписи ❾. Обратите внимание, что в методе `get_rect()` есть параметр, который устанавливает центр равным центру `base`. Это позволяет разместить текстовую надпись поверх белого базового прямоугольника. Наконец, перенесите все на экран, указав прямоугольники источника и цели ❿.

Картографирование влажности грунта

В листинге 14.13 определены функции, позволяющие игроку "картографировать" Марс, в случае если соблюдены условия выигрыша в игре. Когда игрок нажимает клавишу <M>, эти функции будут вызваны функцией `main()`, и снимок планеты будет заменен красочным наложением, которое, мы притворимся, представляет содержание влаги в грунте. Когда игрок отпустит клавишу, вернется нормальный вид Марса. Проверки нажатия данной клавиши также будут выполняться в функции `main()`.

Листинг 14.13. Определяет функции, позволяющие игроку картографировать влажность грунта на Марсе. Файл mars_orbiter.py, часть 13

```
❶ def mapping_on(planet):
      """Показать снимок влажности грунта планеты."""
   ❷ last_center = planet.rect.center
   ❸ planet.image_copy = pg.transform.scale(planet.image_water, (100, 100))
   ❹ planet.image_copy.set_colorkey(BLACK)
     planet.rect = planet.image_copy.get_rect()
     planet.rect.center = last_center

❺ def mapping_off(planet):
      """Восстановить нормальный снимок планеты."""
   ❻ planet.image_copy = pg.transform.scale(planet.image_mars, (100, 100))
     planet.image_copy.set_colorkey(BLACK)
```

Начните с определения функции, которая в качестве аргумента принимает объект `planet` ❶. Создайте переменную `last_center`, как это было сделано в листинге 14.9; она будет использоваться для предотвращения колебаний планеты на своей оси ❷.

Затем прошкалируйте водный снимок Марса в тот же размер, что и нормальный снимок, и назначьте его атрибуту `image_copy` планеты, поскольку многократные

преобразования снимка ухудшают его качество ❸. Сделайте фон снимка прозрачным ❹, получите его прямоугольник `rect` и установите центр прямоугольника равным переменной `last_center`; благодаря этому Марс будет оставаться в центре экрана.

Теперь определите еще одну функцию для случая, когда игрок перестает активно картографировать Марс ❺. В качестве аргумента она также принимает объект `planet`. Вам нужно только сбросить снимок планеты в исходную версию ❻. Поскольку вы по-прежнему используете атрибут `image_copy`, то снова получать объект `rect` не требуется, но при этом следует сделать цвет прозрачным.

Отбрасывание тени

Листинг 14.14 определяет функцию, которая придает Марсу "темную сторону" и отбрасывает тень за планетой. Тень будет представлять собой черный полупрозрачный прямоугольник, правый край которого совпадает с центром спрайта планеты (рис. 14.19). При этом будем считать, что Солнце находится справа от экрана и что на Марсе сейчас весеннее или осеннее равноденствие.

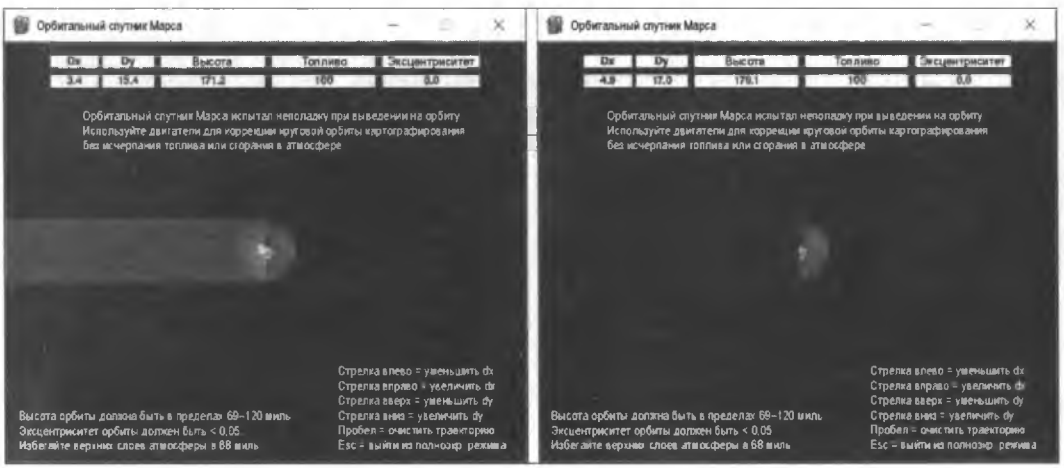


Рис. 14.19. Теневой прямоугольник в полупрозрачном белом (слева) и конечном полупрозрачном черном (справа)

Листинг 14.14. Определяет функцию, которая дает Марсу темную сторону и позволяет ему отбрасывать тень. Файл mars_orbiter.py, часть 14

```
❶ def cast_shadow(screen):
      """Добавить на экран необязательную границу света и тени
         и тень за планетой."""
   ❷ shadow = pg.Surface((400, 100),
                         flags=pg.SRCALPHA) # кортеж равен w,h
   ❸ shadow.fill((0, 0, 0, 210))      # последнее число устанавливает
                                      # прозрачность
     screen.blit(shadow, (0, 270))    # кортеж равен левым верхним координатам
```

Функция `cast_shadow()` принимает в качестве аргумента объект `screen` ❶. Назначьте поверхность пакета `pygame` размером 400×100 пикселов объекту с именем `shadow` ❷. Используйте флаг `SRCALPHA` — обозначающий альфа-канал источника — указав, что вы будете использовать попиксельную альфа (прозрачность). Заполните объект черным цветом и установите альфа-канал — представленный последним числом, равным 210 ❸. Альфа является частью цветовой системы RGBA и имеет допустимые значения от 0 до 255, поэтому результирующий цвет будет очень темным, но не полностью непрозрачным. Наконец, перенесите поверхность на экран вместе с координатами ее левого верхнего угла. Для отключения тени просто закомментируйте вызов указанной функции в `main()` или установите значение альфа-канала равным 0.

Определение функции *main()*

Листинг 14.15 начинает определение функции `main()`, которая запускает игру. В ней инициализируются пакет `pygame` и звуковой микшер, настраивается игровой экран, а инструкции для игрока сохраняются в виде списков.

Листинг 14.15. Начинает функцию `main()` с инициализации пакета `pygame` и звукового микшера, а также настройки экрана игры и определения инструкции. Файл mars_orbiter.py, часть 15

```
def main():
    """Задать надписи и инструкции, создать объекты и запустить
       игровой цикл."""
❶   pg.init()    # инициализировать пакет pygame

    # настроить экран:
❷   os.environ['SDL_VIDEO_WINDOW_POS'] = '700, 100' # установить начальные
                                                    # координаты окна
❸   screen = pg.display.set_mode((800, 645), pg.FULLSCREEN)
❹   pg.display.set_caption("Орбитальный спутник Марса")
❺   background = pg.Surface(screen.get_size())

❻   pg.mixer.init() # для звуковых эффектов

❼   intro_text = [
        'Орбитальный спутник Марса испытал неполадку ' \
        'при выведении на орбиту.',
        'Используйте двигатели для коррекции круговой орбиты ' \
        'картографирования ',
        'без исчерпания топлива или сгорания в атмосфере'
        ]

    instruct_text1 = [
        'Высота орбиты должна быть в пределах 69-120 миль',
```

```
        'Эксцентриситет орбиты должен быть < 0,05',
        'Избегайте верхних слоев атмосферы в 68 миль'
        ]

    instruct_text2 = [
        'Стрелка влево = уменьшить dx',
        'Стрелка вправо = увеличить dx',
        'Стрелка вверх = уменьшить dy',
        'Стрелка вниз = увеличить dy',
        'Пробел = очистить траекторию',
        'Escape = выйти из полноэкр. режима'
        ]
```

Начните функцию `main()` с инициализации пакета `pygame` ❶. Затем используйте метод `environ()` модуля `os` для назначения координат левого верхнего угла игрового окна ❷. Этот шаг не является строго необходимым, но этим я хотел продемонстрировать, что у вас есть контроль над тем, где на рабочем столе появляются окна.

Далее задайте переменную, которая будет содержать объект `screen`, и установите режим изображения в полноэкранный ❸. Используйте кортеж `(800, 645)`, указав размер используемого экрана, если игрок выйдет из полноэкранного режима.

Теперь примените метод `display.set_caption()` пакета `pygame` для назначения окну игры названия "Орбитальный спутник Марса" ❹. Затем примените класс `Surface` пакета `pygame` для создания фонового объекта, который имеет тот же размер, что и экран ❺.

Инициализируйте звуковой микшер пакета `pygame`, который позволит воспроизводить звуковой эффект двигателя ❻. Вы определили этот звук ранее в методе инициализации спутника.

Игра начнется с короткого введения, которое исчезнет через 15 секунд. Постоянные надписи, описывающие управление с клавиатуры и условия выигрыша, занимают нижние углы экрана. Введите текст для них в виде списков ❼. Позже вы передадите эти списки в функцию `instruct_label()`, написанную в листинге 14.12. Каждый элемент списка, выделенный запятой, появится в окне игры отдельной строкой (см. рис. 14.19).

Инстанцирование объектов, настройка верификации орбиты, картографирования и хронометража

Листинг 14.16, по-прежнему находящийся в функции `main()`, инстанцирует объекты `planet` и `satellite`, задает несколько полезных переменных для определения эксцентриситета орбиты, подготавливает игровые часы внутри функции и задает переменную для отслеживания состояния функционала картографирования.

Листинг 14.16. Инстанцирует объекты и задает полезные переменные в `main()`. Файл mars_orbiter.py, часть 16

```
    # инстанцировать объекты planet и satellite
❶ planet = Planet()
❷ planet_sprite = pg.sprite.Group(planet)
❸ sat = Satellite(background)
❹ sat_sprite = pg.sprite.Group(sat)

    # для верификации круговой орбиты
❺ dist_list = []
❻ eccentricity = 1
❼ eccentricity_calc_interval = 5 # оптимизировано для высоты 120 миль

    # хронометрирование
❽ clock = pg.time.Clock()
    fps = 30
    tick_count = 0

    # для функционала картографирования влажности грунта
❾ mapping_enabled = False
```

Продолжите функцию `main()`, создав объект `planet` из класса `Planet` ❶, а затем поместите его в спрайтовую группу ❷. Как вы помните из *главы 13*, `pygame` управляет спрайтами с помощью контейнеров, именуемых страйтовыми группами.

Далее инстанцируйте объект-спутник, передав методу инициализации класса `Satellite` объект `background` ❸. Спутник нуждается в фоне `background` для отрисовки его траектории.

После создания спутника поместите его в собственную спрайтовую группу ❹. Как правило, в своих контейнерах вам придется хранить радикально разные типы спрайтов. Это позволяет легко управлять такими вещами, как порядок вывода на экран и обработка столкновений.

Теперь задайте несколько переменных, которые помогут с вычислением эксцентриситета. Создайте пустой список для хранения значений расстояния, вычисляемых в каждом игровом цикле ❺, затем назначьте переменной `eccentricity` значение-заполнитель, равное `1` ❻, обозначающее некруговую начальную орбиту.

Вы захотите регулярно обновлять переменную `eccentricity`, оценивая любые изменения, которые игрок вносит в орбиту. Помните, что для вычисления эксцентриситета вам нужны апоапсида и периапсида орбиты, а для больших эллиптических орбит может потребоваться некоторое время на то, чтобы фактически их опробовать. Хорошая новость заключается в том, что необходимо рассматривать только "выигрышные" орбиты между 69 и 120 милями. Поэтому вы можете оптимизировать частоту отбора проб для орбит ниже 120 миль, что обычно у спрайта спутника

занимает менее 6 секунд. Используйте 5 секунд и присвойте это значение переменной `excentricity_calc_interval` ❼. Это означает, что для орбит с высотами выше 120 миль расчетный эксцентриситет технически может быть неверным, но он будет достаточно хорошим, учитывая тот факт, что орбита не удовлетворяет условиям выигрыша на этой высоте.

Далее обратитесь к хронометражу. Используйте переменную `clock`. Она будет хранить игровые часы `pygame`, которые будут контролировать скорость игры в кадрах в секунду ❽. Каждый кадр будет представлять один тик часов. Назначьте переменной `fps` значение `30`. Иными словами, игра будет обновляться 30 раз в секунду. Далее задайте переменную `tick_count`, которая будет использоваться для определения времени очистки вводного текста и вызова функции `calc_eccentricity()`.

Завершите этот раздел, объявив переменную, которая будет активировать функционал картографирования, и установите для нее значение, равное `False` ❾. Если игрок достигнет условий выигрыша, то вы измените его на `True`.

Запуск игрового цикла и воспроизведение звуков

Листинг 14.17 содержит код, все еще находящийся в функции `main()`, и запускает игровые часы и цикл `while`, также именуемый *игровым циклом*. Он также получает события, в частности когда игрок запускает двигатели с помощью клавиш со стрелками. Если игрок запускает двигатели, то воспроизводится аудиофайл Ogg Vorbis, и игрок слышит удовлетворительное шипение.

Листинг 14.17. Запускает игровой цикл, получает события и воспроизводит звуки в `main()`. Файл mars_orbiter.py, часть 17

```
❶ running = True
  while running:
    ❽ clock.tick(fps)
      tick_count += 1
    ❸ dist_list.append(sat.distance)

      # получить данные с клавиатуры
    ❶ for event in pg.event.get():
      ❺ if event.type == pg.QUIT:  # закрыть окно
            running = False
      ❽ elif event.type == pg.KEYDOWN and event.key == pg.K_ESCAPE:
            screen = pg.display.set_mode((800, 645)) # выйти
                                                # из полноэкранного режима
      ❼ elif event.type == pg.KEYDOWN and event.key == pg.K_SPACE:
            background.fill(BLACK) # очистить траекторию
      ❽ elif event.type == pg.KEYUP:
          ❾ sat.thrust.stop()     # остановить звучание
            mapping_off(planet)   # выключить просмотр карты влажности
```

```
⑩ elif mapping_enabled:
        if event.type == pg.KEYDOWN and event.key == pg.K_m:
            mapping_on(planet)
```

Сначала задайте переменную `running` для использования с циклом `while`, который выполняет игру ❶, а затем начните цикл. Установите скорость игры с помощью метода `tick()` часов и передайте ему переменную `fps`, которую вы объявили в предыдущем листинге ❷. Если игра кажется вам медленной, увеличьте скорость до 40 кадров в секунду. Для каждого цикла — или кадра — увеличьте на 1 основанный на часах счетчик.

Далее добавьте значение `sat.distance` объекта-спутника в `dist_list` ❸. Это значение является расстоянием между спутником и планетой, рассчитываемое для каждого игрового цикла методом `locate()` спутника.

Теперь соберите данные, введенные игроком через клавиатуру ❹. Как описано в предыдущей главе, `pygame` записывает каждое взаимодействие со стороны пользователя — именуемое *событием* — в событийный буфер. Метод `event.get()` создает список этих событий, которые можно оценить в данном случае с помощью инструкций `if`. Начните с проверки того, закрыл ли игрок окно, для того чтобы выйти из игры ❺. Если это правда, то установите `running` равным `False` для завершения игрового цикла.

Если игрок нажимает клавишу <Esc>, то он выходит из полноэкранного режима, поэтому сбросьте размер экрана в 800×645 пикселов с помощью метода `display.set_mode()`, который вы вызвали в самом начале функции `main()` ❻. Если игрок нажимает клавишу <Пробел>, то заполните фон черным цветом, который сотрет белую орбитальную траекторию спутника ❼.

Когда игрок нажимает клавишу со стрелкой, объект-спутник воспроизводит шипящий звук, но ничто в его методе `check_keys()` не говорит ему остановиться. Поэтому передайте пакету `pygame` любые события `KEYUP` ❽; когда `pygame` засечет, что игрок отпустил клавишу со стрелкой, то вызовите метод `stop()` на двигателе `thrust` для остановки воспроизведения звука ❾.

Для картографирования Марса игрок должен будет удерживать клавишу <M>, поэтому используйте то же событие `KEYUP` для вызова функции `mapping_off()`. Это приведет к сбросу снимка планеты в его нормальное, не картографированное состояние.

Наконец, проверьте, является ли переменная `mapping_enabled` истинной, что означает, что игрок достиг условий выигрыша и готов к картографированию Марса ❿. Если он нажмет клавишу <M>, то вызовите функцию `mapping_on()`, для того чтобы вместо обычного вида планеты показать снимок-наложение влажности грунта.

Применение гравитации, расчет эксцентриситета и обработка аварий

В листинге 14.18 продолжается цикл `while` функции `main()`, воздействуя на спутник гравитацией, а затем вычисляя эксцентриситет его орбиты. Значение эксцентриси-

тета будет обнаруживать, является ли орбита круговой, т. е. одним из условий выигрыша в игре. Указанный листинг также размывает фон и реагирует на аварийные условия исчерпания топлива или сгорания в атмосфере.

Листинг 14.18. Применение гравитации, расчет эксцентриситета и устранение условий для аварий. Файл mars_orbiter.py, часть 18

```
    # получить направление полета и расстояние до планеты
    # и применить гравитацию
❶   sat.locate(planet)
    planet.gravity(sat)

    # вычислить эксцентриситет орбиты
❷   if tick_count % (eccentricity_calc_interval * fps) == 0:
        eccentricity = calc_eccentricity(dist_list)
❸       dist_list = []

    # повторно перенести фон для команды рисования - предотвращает
    # очистку траектории
❹   screen.blit(background, (0, 0))

    # условия аварии, связанной с топливом/высотой
❺   if sat.fuel <= 0:
❻       instruct_label(screen, ['Топливо исчерпано!'], RED, 340, 195)
        sat.fuel = 0
        sat.dx = 2
❼   elif sat.distance <= 68:
        instruct_label(screen, ['Вход в атмосферу!'], RED, 320, 195)
        sat.dx = 0
        sat.dy = 0
```

Вызовите метод `locate()` спутника и передайте ему в качестве аргумента объект `planet` ❶. Этот метод вычисляет направление полета и расстояние до Марса, которые вы используете, для того чтобы направить тарелку, вычислить эксцентриситет орбиты и применить гравитацию. Затем, для того чтобы применить гравитацию, вызовите метод `gravity()` планеты и передайте ему объект-спутник.

Если остаток от деления `tick_count` и `excentricity_calc_interval * fps` равен `0` ❷, то вызовите функцию, вычисляющую эксцентриситет, и передайте ей переменную `dist_list`. Затем сбросьте переменную `dist_list` в `0` для перезапуска выборки образцов расстояния ❸.

Далее вызовите метод `blit()` экрана и передайте ему фон и координаты левого верхнего угла ❹. Размещение этой инструкции имеет большое значение. Например, если вы переместите ее после фрагмента кода, который обновляет спрайты, то на экране игры вы не увидите спутник или Марс.

Теперь рассмотрим случай, когда у игрока заканчивается топливо до достижения им круговой орбиты. Сначала получите текущий уровень топлива из атрибута `fuel`

объекта-спутника ❺. Если уровень равен или ниже `0`, то примените функцию `instruct_label()`, объявив о том, что топливо израсходовано ❻, а затем установите атрибут `dx` спутника равным 2. Это приведет к тому, что спрайт спутника быстро улетит за пределы экрана в космические глубины, а показания высоты будут становиться все больше и больше. Хотя это нереально, зато обеспечивает осведомленность игрока о том, что его попытка оказалась безуспешной!

Последний случай аварии — сгорание игрока в атмосфере. Если атрибут `distance` спутника меньше или равен 68 ❼, поместите надпись в центре экрана с информацией о том, что игрок вошел в атмосферу, а затем установите атрибуты скорости спутника равными `0`. Это приведет к тому, что гравитация зафиксирует спрайт на планете (рис. 14.20). Кроме того, когда `dx` и `dy` равны `0`, метод `update()` спутника (см. листинг 14.7) переключит изображение спутника на его красную аварийную версию.

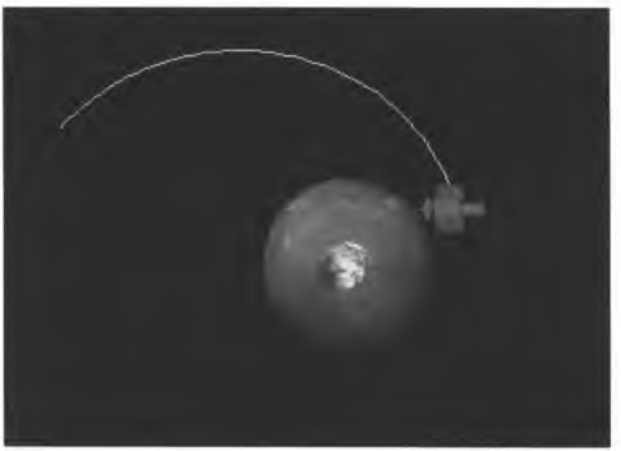


Рис. 14.20. Спутник в аварийной конфигурации

Параметр высоты немного обманывает тем, что высота приравнивается к атрибуту `distance`, который измеряется от центров спрайтов планеты и спутника, а не от поверхности планеты к спутнику. Все сводится к шкале. Планетарные атмосферы представляют собой тонкую грань — по шкале игры марсианская атмосфера была бы менее 2 пикселов в толщину! Особенность разработанной игры состоит в том, что, когда кончик спутниковой тарелки царапает планету, спутник сгорает, но, поскольку размер спрайта спутника нереально велик, центральная точка спрайта должна быть выдвинута на 68 миль дальше.

Вознаграждение за успех и обновление и отрисовка спрайтов

Листинг 14.19, по-прежнему продолжающий цикл `while` функции `main()`, вознаграждает победителя, активируя функционал, который позволяет ему картографировать содержание влаги в марсианском грунте. В реальной жизни это можно достигнуть с помощью радара либо микроволновых резонаторов, которые могут удаленно

измерять влажность в голом грунте на глубину до нескольких дюймов. Указанный листинг также обновляет спрайты планеты и спутника и рисует их на экране.

Листинг 14.19. Активирует функционал картографирования и обновляет спрайты в цикле игры. Файл mars_orbiter.py, часть 19

```
    # активировать функционал картографирования
❶ if eccentricity < 0.05 and sat.distance >= 69 and sat.distance <= 120:
    ❷ map_instruct = ['Нажмите и удерживайте M для картографирования ' \
                      'влажности грунта']
       instruct_label(screen, map_instruct, LT_BLUE, 250, 175)
    ❷ mapping_enabled = True
    else:
       mapping_enabled = False

❹ planet_sprite.update()
❺ planet_sprite.draw(screen)
   sat_sprite.update()
   sat_sprite.draw(screen)
```

Если орбита является круговой и соответствует требованиям высоты ❷, то покажите на экране сообщение с инструкцией игроку о том, что для картографирования влажности грунта нужно нажать клавишу <M> ❷. Поместите текст в скобках, т. к. функция `instruct_label()` ожидает список. Сделайте цвет текста светло-синим и поместите его в центре экрана.

Далее установите переменную активации картографирования `mapping_enabled` равной `True` ❸; в противном случае, если орбита выходит за пределы целевых параметров, установите для нее значение `False`.

Завершите вызовом метода `update()` спрайта планеты через спрайтовую группу ❹, а затем фактически нарисуйте его на экране ❺. Аргументом метода `draw()` является `screen`, т. е. объект, на котором будут отрисованы спрайты. Повторите эти шаги для спрайта спутника.

Вывод на экран инструкций и телеметрии и отбрасывание тени

Листинг 14.20 завершает цикл `while` и функцию `main()`, выводя на экран инструкции, показания датчиков и тень планеты. Текст введения в игру будет показываться при запуске в течение лишь короткого времени.

Листинг 14.20. Выводит на экран текст и тень планеты и вызывает функцию `main()`. Файл mars_orbiter.py, часть 20

```
    # показывать вводный текст в течение 15 секунд
❶ if pg.time.get_ticks() <= 15000: # время в милисекундах
       instruct_label(screen, intro_text, GREEN, 145, 100)
```

```
        # вывести на экран телеметрию и инструкции
    ❷ box_label(screen, 'Dx', (70, 20, 75, 20))
      box_label(screen, 'Dy', (150, 20, 80, 20))
      box_label(screen, 'Высота', (240, 20, 160, 20))
      box_label(screen, 'Топливо', (410, 20, 160, 20))
      box_label(screen, 'Эксцентриситет', (580, 20, 150, 20))

    ❸ box_label(screen, '{:.1f}'.format(sat.dx), (70, 50, 75, 20))
      box_label(screen, '{:.1f}'.format(sat.dy), (150, 50, 80, 20))
      box_label(screen, '{:.1f}'.format(sat.distance),
                (240, 50, 160, 20))
      box_label(screen, '{}'.format(sat.fuel), (410, 50, 160, 20))
      box_label(screen, '{:.8f}'.format(eccentricity),
                (580, 50, 150, 20))
    ❹ instruct_label(screen, instruct_text1, WHITE, 10, 575)
      instruct_label(screen, instruct_text2, WHITE, 570, 510)

      # добавить границу света и тени на поверхности планеты
    ❺ cast_shadow(screen)
    ❻ pg.draw.rect(screen, WHITE, (1, 1, 798, 643), 1)
    ❼ pg.display.flip()

❽ if __name__ == "__main__":
      main()
```

Текст с обобщением игры должен парить ближе к середине экрана достаточно долго, для того чтобы его можно было прочитать, а затем должен исчезнуть. Контролируйте это, используя инструкцию `if` и метод `tick.get_ticks()` пакета `pygame`, который возвращает число миллисекунд, прошедших с момента запуска игры ❶. Если прошло менее 15 секунд, то примените функцию `instruct_label()`, выведя на экран список строк текста из листинга 14.15 зеленым цветом.

Далее изготовьте индикаторы датчиков для вывода показаний, начиная с полей, содержащих заголовки. Используйте функцию `box_label()` и вызывайте ее для каждого из пяти датчиков ❷. Повторите это для показаний датчиков ❸. Обратите внимание, что при передаче текста указанной функции можно использовать метод `format` форматирования символьных цепочек.

Используйте функцию `instruct_label()` для размещения инструктажа, приведенного в листинге 14.15, в нижних углах экрана ❹. Если вы хотите различать текст с описанием условий выигрыша от текста с определением функций клавиш, то вы можете свободно поменять цвет текста.

Теперь вызовите функцию, которая изображает тень планеты ❺, а затем в качестве последнего штриха добавьте границу с помощью метода `draw.rect()` пакета `pygame` ❻. Передайте ему объект `screen`, цвет границы, угловые координаты и ширину линии.

Завершите функцию `main()` и ее игровой цикл, перенеся изображение ❼. Как описано в предыдущей главе, метод `flip()` переносит все из объекта `screen` в визуальное изображение.

Наконец, вызовите функцию `main()` в глобальном пространстве, используя стандартный синтаксис для ее запуска автономно либо в виде модуля ❽.

Резюме

В этой главе вы использовали пакет `pygame` для создания двухмерной игры в аркадном стиле со спрайтами изображений, звуковыми эффектами и клавиатурными элементами управления. Вы также создали забавный эвристический метод изучения орбитальной механики. В игре должны работать все технические приемы, показанные в *разд. "Аэродинамика для геймеров" ранее в этой главе*. В следующем разделе, *"Сложные проекты"*, вы сможете продолжить совершенствовать как игру, так и игровой опыт.

Сложные проекты

Доработайте игру "Орбитальный спутник Марса" и сделайте ее неповторимой, усовершенствовав ее и добавив новые задачи, основанные на следующих предложениях. Как всегда, никаких решений для сложных проектов не предусмотрено.

Титульный экран игры

Скопируйте и отредактируйте программу mars_orbiter.py так, чтобы перед главным игровым экраном на короткое время появлялась заставка. Пусть на ней изображается наклейка типа той, которая используется в NASA для миссии разведывательной станции Mars Global Surveyor (рис. 14.21), но сделайте ее уникальной для игры "Орбитальный спутник Марса". Несколько других наклеек NASA можно увидеть в цвете по адресу **https://space.jpl.nasa.gov/art/patches.html**.

Рис. 14.21. Наклейка миссии исследовательской станции Mars Global Surveyor

Умные датчики

Скопируйте и отредактируйте программу mars_orbiter.py так, чтобы показания высоты и эксцентриситета использовали либо красный фон, либо красный цвет текста, когда их значения находятся за пределами целевых диапазонов. Но будьте осторожны: значение кругового эксцентриситета должно оставаться красным до тех пор, пока значение высоты не окажется в диапазоне!

Временное прекращение радиосвязи

Скопируйте и отредактируйте программу mars_orbiter.py так, чтобы клавиатурные элементы управления блокировались, когда спутник находится в пределах прямоугольника `shadow`.

Подсчет баллов

Скопируйте и отредактируйте программу mars_orbiter.py так, что она выставляла игроку баллы и сохраняла лучшие результаты в выводимом на экран списке рекордов. Самые высокие баллы получают те, кто достигает самой низкой допустимой орбиты при использовании наименьшего объема топлива за наименьшее время. Например, топливной компонентой балла может быть объем оставшегося топлива; орбитальной компонентой — максимально допустимая высота (120) минус высота круговой орбиты, а временной компонентой — обратное значение времени, затраченного на достижение круговой орбиты, умноженное на 1000. Для получения окончательного счета сложите все три компоненты вместе.

Руководство по стратегии игры

Скопируйте и отредактируйте программу mars_orbiter.py так, чтобы она включала всплывающее руководство по стратегии или файл справки, путем встраивания некоторых рисунков из *разд. "Астродинамика для геймеров" ранее в этой главе*. Например, добавьте в инструктаж строку, в которой игрок для получения справки должен нажать и удерживать клавишу <H>. В результате этого будут вызваны изображения различных орбитальных маневров, таких как гомановский переход или однотангенциальное сжигание с циклическим просмотром рисунков. Обязательно включите комментарии о сильных и слабых сторонах каждого технического приема и приостановите игру до тех пор, пока руководство не будет открыто.

Аэродинамическое торможение

Аэродинамическое торможение — это способ экономии топлива, который для замедления космического аппарата использует атмосферное трение (рис. 14.22). Скопируйте и отредактируйте программу mars_orbiter.py с целью включения в нее аэродинамического торможения. В функции `main()` установите самую низкую выигрышную высоту равной 70 милям и самую низкую безопасную высоту равной

Рис. 14.22. Скругление орбиты с использованием атмосферы вместо ретроградного сжигания топлива

60 милям. Если высота спутника составляет от 60 до 70 миль, уменьшите его скорость на небольшую величину.

На рис. 14.23 показан пример использования аэродинамического торможения в игре для скругления эллиптической орбиты. Верхние слои атмосферы были установлены равными высоте 80 миль. Аэродинамическое торможение служит той же цели, что и ретроградное сжигание топлива в периапсиде, но вы должны быть

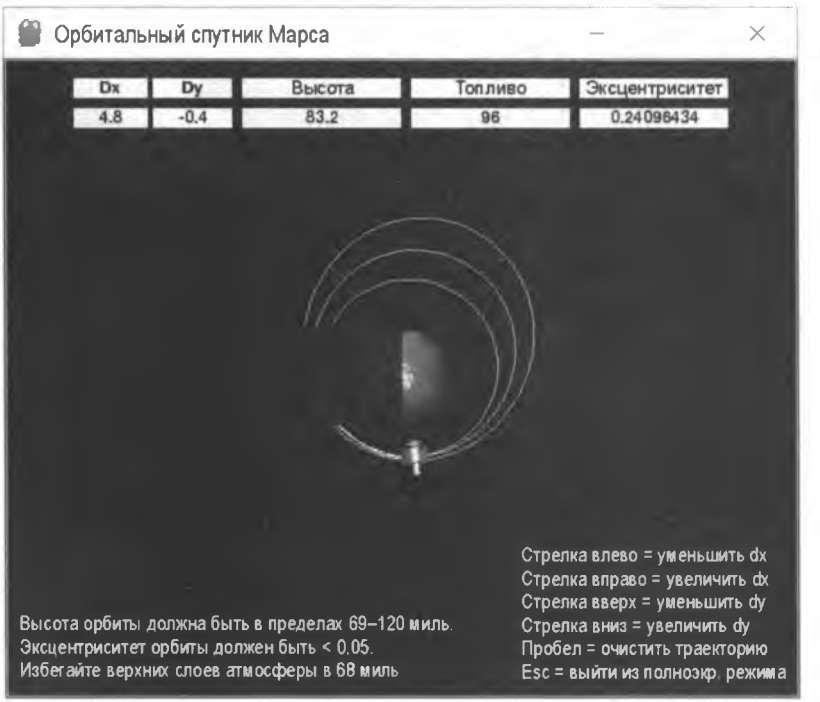


Рис. 14.23. Скругление орбиты с использованием аэродинамического торможения
Обратите внимание на низкий расход топлива

осторожны и терпеливы и поднять орбиту за пределы атмосферы, прежде чем она станет круговой.

Агентство NASA использовало аналогичный прием для перемещения исследовательской станции Mars Global Surveyor с эллиптической орбиты захвата на ее конечную картографическую орбиту. Этот процесс занял много месяцев, потому что необходимо было защитить космический аппарат от перегрева в атмосфере.

Сигнал о вторжении!

Скопируйте и отредактируйте программу mars_orbiter.py так, чтобы инстанцировался и пролетал новый объект `planet`, нарушая орбиту спутника своей гравитацией. Создайте новый спрайт, представляющий комету или астероид, и запускайте его через случайные промежутки времени (но не слишком часто!). Не применяйте к указанному объекту метод `gravity()` Марса, для того чтобы он не выходил на марсианскую орбиту. Наоборот, примените метод `gravity()` нового объекта к спутнику. Поиграйте с массой нового объекта так, чтобы он заметно возмущал орбиту спутника с расстояния 100 пикселов или около того. Дайте объекту пройти мимо Марса или спутника без столкновения.

Скольжение по верхам

Орбитальный спутник Марса в настоящее время использует экваториальную орбиту. Это было сделано для легкости написания программного кода, т. к. приходится поворачивать лишь один снимок Марса. Но истинные картографические орбиты используют полярные орбиты, ориентированные перпендикулярно экваториальным орбитам — и проходят над полюсами планеты (рис. 14.24). Когда планета вращается под орбитой, спутник способен наносить на карту всю ее поверхность. При экваториальных орбитах высокие широты из-за искривления поверхности планеты практически не изменяются (см. пунктирную линию на рис. 14.24).

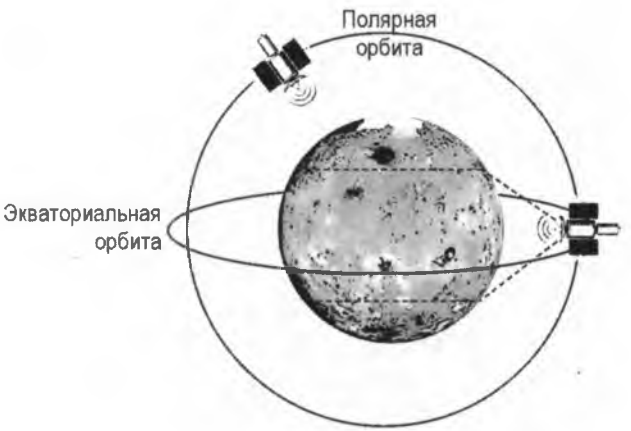


Рис. 14.24. Полярные и экваториальные орбиты; условные северные и южные границы картографирования для экваториальной орбиты представлены пунктирной линией

Скопируйте и отредактируйте программу mars_orbiter.py так, чтобы спутник следовал по полярной орбите. Все это сопряжено с изменением снимка Марса. Но вы больше не сможете использовать один снимок "сверху вниз"; вид должен быть перпендикулярен оси вращения планеты. Пример видео см. по адресу **https://youtu.be/IP2SDbhFbXk**; анимированный GIF-файл Марса см. по адресу **http://gph.is/2caBKKS**. Анимированные GIF-файлы можно использовать непосредственно в `pygame`, но вы можете выделить и использовать отдельные кадры. Инструменты для выделения кадров можно найти в Интернете, и в следующей главе вы примените один из таких продуктов для извлечения изображений из видеоролика.

15

УЛУЧШЕНИЕ АСТРОФОТОСЪЕМКИ С ПОМОЩЬЮ НАЛОЖЕНИЯ СНИМКОВ ПЛАНЕТЫ

Если вы когда-либо смотрели в телескоп на Юпитер, Марс или Сатурн, то наверняка были немного разочарованы. Планеты казались маленькими и безликими. Вам хотелось увеличить масштаб, причем существенно, но это не работало. Все, что имеет более 200-кратное увеличение, как правило, является размытым.

Проблема заключается в турбулентности воздуха, или в том, что астрономы называют *видимостью*. Даже в ясную ночь воздух постоянно находится в движении, с восходящими и нисходящими тепловыми потоками, которые могут легко размыть точки света, представляющие небесные объекты. Но с коммерциализацией *устройств с зарядовой связью* (charge-coupled device, CCD) в 1980-х годах астрономы нашли способ преодолевать турбулентность. Цифровая фотография позволяет использовать метод, именуемый *наложением снимков* (stacking — укладка в стопку), в котором многие фотографии — некоторые хорошие, некоторые плохие — усредняются или складываются в одно изображение. При достаточно большом числе фотографий постоянные неизменные признаки (такие, как поверхность планеты) доминируют над транзитными признаками (такими, как рассеянное облако). Это позволяет астрофотографам увеличивать пределы увеличения, а также компенсировать менее оптимальные условия наблюдения.

В этой главе вы будете использовать сторонний модуль Python под названием `pillow` для наложения сотен снимков Юпитера. Результатом будет один снимок с более высоким соотношением сигнал/шум, чем любой из отдельных кадров. Вы также будете работать с файлами в разных папках, помимо каталога, в котором находится ваш код на Python, и управлять файлами и папками с помощью модуля

операционной системы (`os`) и модуля утилит командной оболочки (`shutil`) языка Python.

Проект 23: наложение фотоснимков Юпитера

Большой, яркий и красочный газовый гигант Юпитер является излюбленной мишенью астрофотографов. Даже любительские телескопы могут разглядеть его оранжевую полосу, вызванную линейными полосами облаков, и Большое красное пятно — овальную бурю, настолько большую, что она может поглотить Землю (рис. 15.1).

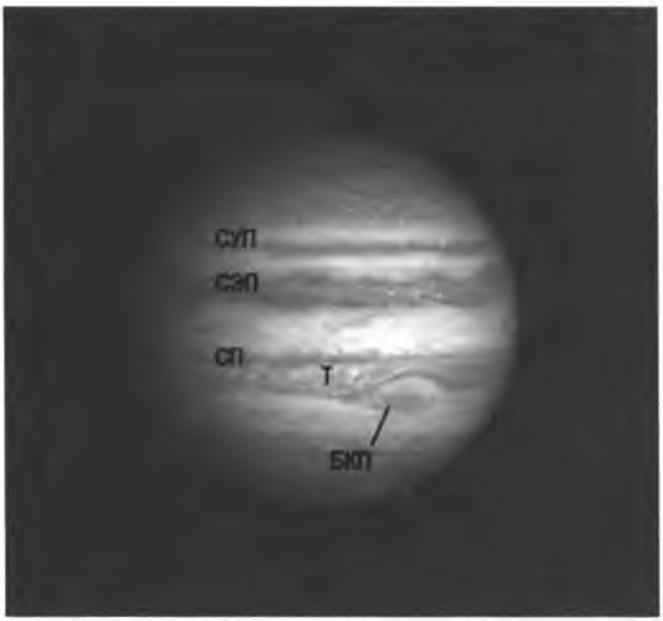


Рис. 15.1. Юпитер, сфотографированный космическим аппаратом "Кассини"
СУП — северный умеренный пояс; СЭП — северный экваториальный пояс; СП — субэкваториальный пояс; БКП — Большое красное пятно; Т — турбулентность с подветренной стороны БКП

Юпитер — отличный предмет для обучения наложению снимков. Его линейные облачные полосы и Большое красное пятно обеспечивают глаз точками калибровки для оценивания улучшений в определении краев и четкости, а его относительно большой размер позволяет легко обнаруживать шум.

Шум проявляется как "зернистость". Каждая цветовая полоса имеет свои артефакты, в результате чего на изображении появляются цветные крапинки. Основными источниками шума являются камера (шум электронного считывания и тепловой сигнал) и фотонный шум от самого света, т. к. переменное число фотонов с течением времени поражает датчик. Шумовые артефакты, к счастью, по своей природе являются случайными и в значительной степени могут быть аннулированы за счет наложения снимков.

Цель

Написать программы, которые обрезают, масштабируют, накладывают поверх друг друга и улучшают снимки, создавая более четкую фотографию Юпитера.

Модуль *pillow*

Для работы с изображениями вам понадобится бесплатный сторонний модуль Python под названием `pillow`. Этот проект является преемником библиотеки обработки изображений Python Imaging Library (PIL), разработка которой была прекращена в 2011 г. Модуль `pillow` стал "веткой" репозитория PIL и был обновлен до Python 3.

Указанный модуль можно использовать в Windows, macOS и Linux, и он поддерживает большое число форматов изображений, включая PNG, JPEG, GIF, BMP и TIFF. Он предлагает стандартные процедуры обработки изображений, такие как изменение отдельных пикселов, маскирование, обработка прозрачности, фильтрация и улучшение, а также добавление текста. Но реальная сила данного модуля состоит в его способности с легкостью редактировать многочисленные изображения.

Установить модуль `pillow` можно без труда с помощью инструмента pip (подробнее о `pip` *см. в разд. "Управление документами Word с помощью python-docx" главы 6*). В командной строке введите команду

```
pip install pillow
```

Большинство основных дистрибутивов Linux включают `pillow` в пакеты, которые ранее содержали PIL, поэтому в вашей системе `pillow` уже может иметься. Если независимо от вашей платформы библиотека PIL уже установлена, то перед установкой `pillow` вам нужно ее удалить. Дополнительные инструкции см. на вебстранице **http://pillow.readthedocs.io/en/latest/installation.html**.

Работа с файлами и папками

Во всех предыдущих проектах этой книги вы содержали файлы и модули в той же папке, что и ваш код Python. Это было удобно в простых проектах, но такой подход не используется в реальных приложениях и, разумеется, не годится, если вы имеете дело с сотнями файлов изображений, которые вы создадите в этом проекте. К счастью, Python поставляется с несколькими модулями, которые способны в этом помочь, в частности `os` и `shutil`. Но сначала я кратко рассмотрю каталожные пути.

Каталожные пути

Каталожный путь — это адрес файла или папки. Он начинается с корневого каталога, который в Windows обозначается буквой (например, C:\), а в системах на базе UNIX косой чертой (/). Дополнительные диски в Windows получают другие, отличные от C буквы, диски в macOS помещаются под /volume, а диски в UNIX под /mnt (mnt обозначает "монтирование").

ПРИМЕЧАНИЕ

В примерах этой главы я использую операционную систему Windows, но вы можете достичь того же результата в macOS и других системах. И как это обычно делается, термины "каталог" и "папка" — здесь синонимы.

Имена путей выглядят по-разному в зависимости от операционной системы. Windows разделяет папки обратной косой чертой (\), тогда как системы macOS и UNIX используют прямую косую черту (/). Кроме того, в UNIX имена папок и файлов чувствительны к регистру.

Если вы пишете свою программу в Windows и вводите имена путей с обратными косыми чертами, то другие платформы эти пути не распознают. К счастью, метод `os.path.join()` автоматически обеспечивает, чтобы ваше имя пути подходило для любой операционной системы, в которой работает Python. Давайте рассмотрим этот и другие примеры в листинге 15.1.

Листинг 15.1. Работа с путями Windows с помощью модуля `os`

```
❶ >>> import os
❷ >>> os.getcwd()
  'C:\\Python35\\Lib\\idlelib'
❸ >>> os.chdir('C:\\Python35\\Материалы по Python 3')
  >>> os.getcwd()
  'C:\\Python35\\Материалы по Python 3'
❹ >>> os.chdir(r'C:\Python35\Материалы по Python 3\Наложение снимков планеты')
  >>> os.getcwd()
❺ 'C:\\Python35\\Материалы по Python 3\\Наложение снимков планеты'
❻ >>> os.path.join('Наложение снимков планеты', 'stack_8', '8file262.jpg')
  'Наложение снимков планеты\\stack_8\\8file262.jpg'
❼  >>> os.path.normpath('C:/Python35/Материалы по Python 3')
  'C:\\Python35\\Материалы по Python 3'
❽ >>> os.chdir('C:/Python35')
  >>> os.getcwd()
  'C:\\Python35'
```

После импорта модуля `os` для доступа к функционалу, зависящему от операционной системы ❶, получите *текущий рабочий каталог*, или cwd ❷. Текущий рабочий каталог назначается процессу при его запуске, т. е. при запуске скрипта из командной оболочки указанный каталог оболочки и скрипта будет одним и тем же. Для программы на Python текущий рабочий каталог — это папка, содержащая программу. Когда вы получите текущий рабочий каталог, вам будет показан полный путь. Обратите внимание, что в целях экранирования символов обратной косой черты, применяемых в качестве разделителей имен файлов, необходимо использовать дополнительные обратные косые черты.

Далее вам потребуется изменить текущий рабочий каталог посредством метода `os.chdir()` ❸, передав ему полный путь в кавычках, указав двойные обратные ко-

сые черты. Затем вы снова получаете текущий рабочий каталог, который покажет новый путь.

Если вы не хотите использовать двойную обратную косую черту, то перед символьным аргументом с именем пути можно ввести r, преобразовав его в сырую (необработанную) *символьную цепочку* ❹. В сырых символьных цепочках используются различные правила для последовательностей с экранированием обратной косой черты, но даже сырая символьная цепочка не может заканчиваться одной обратной косой чертой. Путь по-прежнему будет показываться с двойной обратной косой чертой ❺.

Если вы хотите, чтобы ваша программа была совместима со всеми операционными системами, то примените метод `os.path.join()` и передайте ему имена папок и файлов без символа-разделителя ❻. Методы `os.path` осведомлены об используемой вами системе и возвращают правильные разделители. Это позволяет осуществлять независимую от платформы манипуляцию именами файлов и папок.

Метод `os.path.normpath()` корректирует разделители с учетом используемой вами системы ❼. В приведенном выше примере с Windows неправильные UNIX-подобные разделители заменяются обратными косыми чертами. Привычная нам ОС Windows также поддерживает использование прямой косой черты и автоматически сделает преобразование ❽.

Полный путь к каталогу — от корня вниз — называется *абсолютным путем*. Для упрощения работы с каталогами можно пользоваться сокращениями, именуемые *относительными путями*. Относительные пути интерпретируются с точки зрения текущего рабочего каталога. Если абсолютные пути начинаются с прямой косой черты или метки диска, то относительные пути записываются иначе. В следующем ниже фрагменте кода можно поменять каталоги без ввода абсолютного пути — Python осведомлен о новом местоположении, потому что он находится *внутри* текущего рабочего каталога. За кулисами относительный путь присоединяется к пути, ведущему к текущему рабочему каталогу, формируя полный абсолютный путь.

```
>>> os.getcwd()
'C:\\Python35\\Материалы по Python 3'
>>> os.chdir('Наложение снимков планеты')
>>> os.getcwd()
'C:\\Python35\\Материалы по Python 3\\Наложение снимков планеты'
```

Идентифицировать папки и еще больше сэкономить на вводе можно с помощью точки (.) и двойной точки (..). Например, в Windows .\ ссылается на текущий рабочий каталог, а ..\ указывает на родительский каталог, который содержит текущий рабочий каталог. Точка также может использоваться для получения абсолютного пути к вашему текущему рабочему каталогу:

```
>>> os.path.abspath('.')
'C:\\Python35\\Материалы по Python 3\\Наложение снимков планеты\\для_книги'
```

"Точечные" папки могут использоваться в Windows, macOS и Linux. Дополнительные сведения о модуле `os` см. на веб-странице **https://docs.python.org/3/library/os.html**.

Модуль утилит командной оболочки

Модуль `shutil` утилит командной оболочки обеспечивает высокоуровневые функции для работы с файлами и папками, такие как копирование, перемещение, переименование и удаление. Поскольку он является частью стандартной библиотеки Python, загрузить `shutil` можно, просто введя `import shutil`. Примеры использования указанного модуля приведены в разделах кода этой главы. Между тем документацию по данному модулю можно найти по адресу **https://docs.python.org/3.7/library/shutil.html**.

Видеоролик

Брукс Кларк записал используемый в данном проекте цветной видеоролик Юпитера в ветреную ночь в Хьюстоне, штат Техас. Это mov-файл объемом 101 Мбайт длительностью около 16 секунд.

Длительность видеоролика является намеренно короткой. Период вращения Юпитера составляет около 10 часов. Это означает, что неподвижные фотографии могут размываться с продолжительностью экспозиции всего в минуту, и признаки, которые вы хотите усилить за счет наложения видеокадров, могут менять положение, значительно усложняя данный процесс.

Для конвертации видеокадров в отдельные снимки я использовал бесплатный набор мультимедийных программ Free Studio, разработанный в DVDVideoSoft. Инструмент конвертации формата Free Studio в JPG (Video to JPG Converter) позволяет захватывать изображения с постоянными временны*ы*ми или кадровыми интервалами. Я установил интервал для выборки кадров по всей длине видео. Это было сделано для того, чтобы улучшить шансы захвата некоторых снимков, когда воздух был неподвижен и видимость была хорошей.

Несколько сотен снимков должно быть достаточно для того, чтобы продемонстрировать очевидное улучшение за счет наложения. В этом случае я захватил 256 кадров.

Папку со снимками под названием video_frames можно найти онлайн вместе с ресурсами книги по адресу **https://www.nostarch.com/impracticalpython/**. Скачайте эту папку и сохраните ее.

Пример кадра из видео в оттенках серого показан на рис. 15.2. Видно, что полосы облаков Юпитера слабые и нечеткие, Большое красное пятно не различимо, и снимок страдает от низкой контрастности, являющейся распространенным побочным эффектом увеличения. Шумовые артефакты также придают Юпитеру зернистый вид.

В дополнение к этим проблемам ветер тряс камеру, и неточное отслеживание заставило планету дрейфовать в сторону к левой стороне кадра. Пример *бокового дрейфа* можно увидеть на рис. 15.3, в котором я наложил пять случайно выбранных кадров с черным фоном, сделанным прозрачным.

Рис. 15.2. Пример кадра из видеоролика с Юпитером

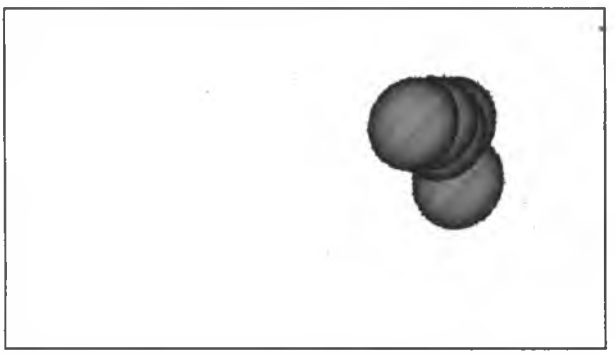


Рис. 15.3. Пример тряски и дрейфа на видеоролике с Юпитером, основанный на пяти случайно выбранных кадрах

Движение не обязательно имеет плохие последствия, потому что подергивания снимка могут сгладить дефекты, ассоциированные с поверхностью CCD-датчика, пылью на объективе или датчике и т. д. Но ключевое допущение относительно наложения снимков заключается в том, что в результате снимки идеально выравниваются, благодаря чему по мере усреднения снимков постоянные признаки, такие как полосы облаков Юпитера, усиливают друг друга.

Для того чтобы соотношение сигнал/шум было высоким, снимки должны быть зарегистрированы. *Регистрация изображений*[1] — это процесс преобразования данных в одну и ту же систему координат с целью их сравнения и интеграции. Регистрация, пожалуй, является самой сложной частью наложения снимков. Астрономы обычно используют коммерческое программное обеспечение, такое как RegiStax, RegiStar, Deep Sky Stacker или CCDStack, которое помогает им выравнивать и накладывать свои астрофотоснимки поверх друг друга. Тем не менее вы попробуете это на практике и проделаете это самостоятельно с помощью языка Python.

---

[1] Регистрация снимков (image registration) — это процесс координатного совмещения двух или более снимков одной и той же сцены, сделанных в разное время, с разных точек зрения и/или разными датчиками. По сути дела, она геометрически выравнивает два снимка — эталонный и датчиковый. См. **https://www.sciencedirect.com/science/article/pii/S0262885603001379**. — *Прим. перев.*

Стратегия

Приведем необходимые для наложения снимков шаги (первый из них уже завершен):

1. Извлечь снимки из видеозаписи.
2. Обрезать изображения вокруг Юпитера.
3. Прошкалировать обрезанные снимки в тот же размер.
4. Совместить снимки в один снимок.
5. Улучшить и отфильтровать итоговый снимок.

Код

Все шаги можно включить в одну программу, но я решил распределить их по трем отдельным программам. Это связано с тем, что обычно по пути хочется остановиться и проверить результаты, плюс может возникнуть желание выполнить более поздние процедуры, такие как улучшение, без необходимости полностью перезапускать весь рабочий поток. Первая программа будет обрезать снимки и масштабировать их, вторая будет их накладывать, а третья — улучшать.

Код обрезки и масштабирования снимков

Сначала нужно выполнить регистрацию снимков. В случае больших ярких объектов, таких как Луна и Юпитер, одним из подходов в астрофотографии является обрезка каждого снимка так, чтобы его четыре границы были касательными к поверхности тела. В результате этого удаляется значительная часть неба и смягчаются любые проблемы, связанные с дрожанием и дрейфом. Масштабирование обрезанных снимков обеспечивает их одинаковый размер и будет немного их сглаживать, уменьшая шум.

Программу crop_n_scale_images.py можно скачать с **https://www.nostarch.com/impracticalpython/**. Сохраните ее в каталоге, который содержит папку с захваченными видеокадрами.

Импорт модулей и определение функции *main()*

Листинг 15.2 импортирует модули и определяет функцию `main()`, которая выполняет программу crop_n_scale_images.py.

Листинг 15.2. Импортирует модули и определяет функцию `main()`. Файл crop_n_scale_images.py, часть 1

```
❶ import os
  import sys
❷ import shutil
❸ from PIL import Image, ImageOps
```

```
def main():
    """Взять начальную папку, скопировать папку,
    выполнить функцию обрезки и очистить папку."""

    # получить имя папки в текущем рабочем каталоге
    # с оригинальными видеоснимками
❹   frames_folder = 'video_frames'

    # подготовить файлы и папки
❺   del_folders('cropped')
❻   shutil.copytree(frames_folder, 'cropped')

    # run cropping function
    print("начать обрезание и шкалирование...")
❼   os.chdir('cropped')
    crop_images()
❽   clean_folder(prefix_to_save='cropped') # удалить
                                           # необрезанные оригиналы
    print("Готово! \n")
```

Начните с импортирования модулей операционной системы (`os`) и системных модулей (`sys`) ❶. Импорт модуля `os` уже включает в себя импорт модуля `sys`, но этот функционал в будущем может выйти из употребления, поэтому лучше всего импортировать `sys` вручную. Модуль `shutil` содержит описанные ранее утилиты командной оболочки ❷. Из библиотеки обработки изображений вы будете использовать модуль `Image` для загрузки, обрезки, конвертации и фильтрации снимков; также вам понадобится модуль `ImageOps` для шкалирования снимков ❸. Обратите внимание, что в инструкции `import` нужно использовать библиотеку PIL, а не модуль `pillow`.

Начните функцию `main()` с назначения переменной `frames_folder` имени начальной папки ❹. Эта папка содержит все исходные снимки, захваченные из видеоролика.

Вы сохраните обрезанные снимки в новой папке с именем cropped, но утилиты командной оболочки не будут создавать эту папку, если она уже существует, поэтому вызовите функцию `del_folders()`, которую вы напишете через мгновение ❺. В том виде, как она написана, эта функция не выдаст ошибку, если папка не существует, поэтому ее можно безопасно выполнять в любое время.

Всегда нужно работать с копией оригинальных снимков, поэтому используйте метод `shutil.copytree()` копирования папки, содержащей оригиналы, в новую папку с именем cropped ❻. Теперь перейдите в эту папку ❼ и вызовите функцию `crop_images()`, которая будет обрезать и масштабировать снимки. Вслед за этим выполните функцию `clean_folder()`, которая удаляет исходные видеокадры, которые были скопированы в папку cropped и все еще болтаются там без надобности ❽.

Обратите внимание на использование имени параметра при передаче аргумента функции `clean_folder()`, поскольку это делает назначение функции очевиднее.

Выведите сообщение "Готово!", уведомив пользователя о том, что программа завершила свою работу.

Удаление и очистка папок

Листинг 15.3 определяет вспомогательные функции удаления файлов и папок программы crop_n_scale_images.py. Модуль `shutil` откажется создавать новую папку, если в целевом каталоге уже существует папка с таким же именем. Если вы хотите выполнить эту программу более одного раза, то сначала необходимо удалить или переименовать существующие папки. Указанная программа также переименует снимки после их обрезки, и вы наверняка захотите удалить исходные снимки, прежде чем начать их накладывать. Поскольку там будут сотни снимковых файлов, эти функции будут автоматизировать задачу, которая в противном случае окажется трудоемкой.

Листинг 15.3. Определяет функции для удаления папок и файлов. Файл crop_n_scale_images.py, часть 2

```
❶ def del_folders(name):
      """Если в каталоге существует папка с именованным префиксом,
         то удалить ее."""
   ❷ contents = os.listdir()
   ❸ for item in contents:
      ❹ if os.path.isdir(item) and item.startswith(name):
         ❺ shutil.rmtree(item)

❻ def clean_folder(prefix_to_save):
      """Удалить все файлы в папке за исключением тех,
         которые имеют именованный префикс."""
   ❼ files = os.listdir()
      for file in files:
      ❽ if not file.startswith(prefix_to_save):
         ❾ os.remove(file)
```

Определите функцию `del_folders()`, которая удаляет папки ❶. Единственным ее аргументом будет имя папки, которую вы хотите удалить.

Далее перечислите содержимое папки ❷, а затем начните перебирать содержимое в цикле ❸. Если указанная функция обнаруживает элемент, который начинается с имени папки и также является каталогом ❹, то примените метод `shutil.rmtree()` для удаления папки ❺. Как вскоре будет видно, для удаления папки используется другой метод, нежели для удаления файла.

ПРИМЕЧАНИЕ

Всегда будьте осторожны, когда используете метод `rmtree()`, т. к. он удаляет папки и их содержимое безвозвратно. Можно стереть бóльшую часть вашей системы, потерять важные документы, не связанные с проектами Python, и даже сломать ваш компьютер!

Теперь определите вспомогательную функцию "очистки" папки и передайте ей префикс файлов, которые вы *не хотите удалять* ❻. Сначала это немного противоречит здравому смыслу, но, поскольку вы хотите сохранить только последнюю партию обработанных вами снимков, вам не нужно беспокоиться о явном перечислении любых других находящихся в папке файлов. Если файлы не начинаются с указанного префикса, например cropped, то они автоматически удаляются.

Процесс аналогичен предыдущей функции. Перечислите содержимое папки и начните перебирать список в цикле ❼. Если файл не начинается с указанного вами префикса ❽, то используйте метод `os.remove()` для его удаления ❾.

Обрезка, масштабирование и сохранение снимков

Листинг 15.4 регистрирует захваченные из видео кадры путем укладки рамки вокруг Юпитера и обрезки снимка по рамке (рис. 15.4). Этот метод хорошо работает с яркими снимками на черном поле (еще один пример см. в *разд. "Дальнейшее чтение" далее в этой главе*).

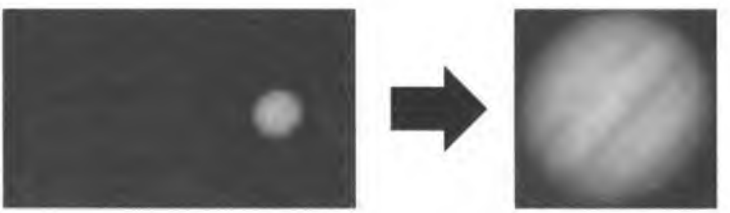


Рис. 15.4. Обрезка оригинального видеокадра до размера Юпитера для выравнивания снимков

Обрезая снимки плотно вокруг Юпитера, вы решаете все проблемы дрожания и дрейфа.

Каждый обрезанный снимок также масштабируется в более крупный и единообразный размер и слегка сглаживается ради уменьшения шума. Обрезанные и отмасштабированные снимки будут храниться в собственной папке, которую позже создаст функция `main()`.

Листинг 15.4. Обрезает исходные видеокадры в прямоугольник вокруг Юпитера и масштабирует их. Файл crop_n_scale_images.py, часть 3

```
❺ def crop_images():
      """Обрезать и отмасштабировать снимки планеты в прямоугольник
         вокруг планеты."""
   ❷ files = os.listdir()
   ❸ for file_num, file in enumerate(files, start=1):
      ❹ with Image.open(file) as img:
         ❺ gray = img.convert('L')
         ❻ bw = gray.point(lambda x: 0 if x < 90 else 255)
```

```
            ❼ box = bw.getbbox()
              padded_box = (box[0]-20, box[1]-20, box[2]+20, box[3]+20)
            ❽ cropped = img.crop(padded_box)
              scaled = ImageOps.fit(cropped, (860, 860),
                                    Image.LANCZOS, 0, (0.5, 0.5))
              file_name = 'cropped_{}.jpg'.format(file_num)
            ❾ scaled.save(file_name, "JPEG")

if __name__ == '__main__':
    main()
```

Функция `crop_images()` не принимает никаких аргументов ❶, но в конечном итоге будет работать с копией папки — с именем cropped, содержащей оригинальные видеокадры. Вы сделали эту копию в функции `main()` перед вызовом указанной функции.

Начните функцию с создания списка содержимого текущей папки (cropped) ❷. Программа будет последовательно нумеровать каждый снимок, поэтому используйте функцию `enumerate()` с циклом `for` и установите параметр `start` равным `1` ❷. Если вы не использовали функцию `enumerate()` раньше, то знайте, что она представляет собой встроенную вспомогательную функцию, которая действует как автоматический генератор счетчика; счетчик будет назначен переменной `file_num`.

Далее объявите переменную `img`, которая будет содержать снимок, и используйте метод `open()` для открытия файла ❹.

Для того чтобы вписать границы ограничительной рамки в Юпитер, вам нужно, чтобы все части снимка, отличные от Юпитера, были черными — `(0, 0, 0)`. К сожалению, позади Юпитера есть блуждающие, связанные с шумом, не черные пикселы, а край планеты является диффузным и градационным. Эти проблемы приводят к рамкам неоднородной формы (рис. 15.5). К счастью, эти проблемы можно легко решить, конвертировав снимок в черно-белый. Затем этот конвертированный снимок можно использовать для определения правильных размеров рамки для каждой цветной фотографии.

Для устранения шумовых эффектов, которые нарушают технику выявления ограничительной рамки, конвертируйте загруженный снимок в режим "L", состоящий из 8-битных черно-белых пикселов, и объявите переменную `gray`, обозначающую шкалу оттенков серого ❺. В этом режиме имеется всего один канал (в отличие от трех каналов RGB для цветных изображений), поэтому при пороговой обработке вам нужно выбрать только одно-единственное значение, т. е. установить предел, выше или ниже которого происходит действие.

Задайте новую переменную с именем `bw` для хранения истинного черно-белого снимка ❻. Примените метод `point()`, используемый для изменения значений пикселов, и лямбда-функцию, которая устанавливает любое значение ниже 90 равным черному (`0`), а все остальные значения равными белому (`255`). Пороговое значение было определено методом проб и ошибок. Метод `point()` теперь возвращает чистое изображение для вписывания ограничительной рамки (рис. 15.6).

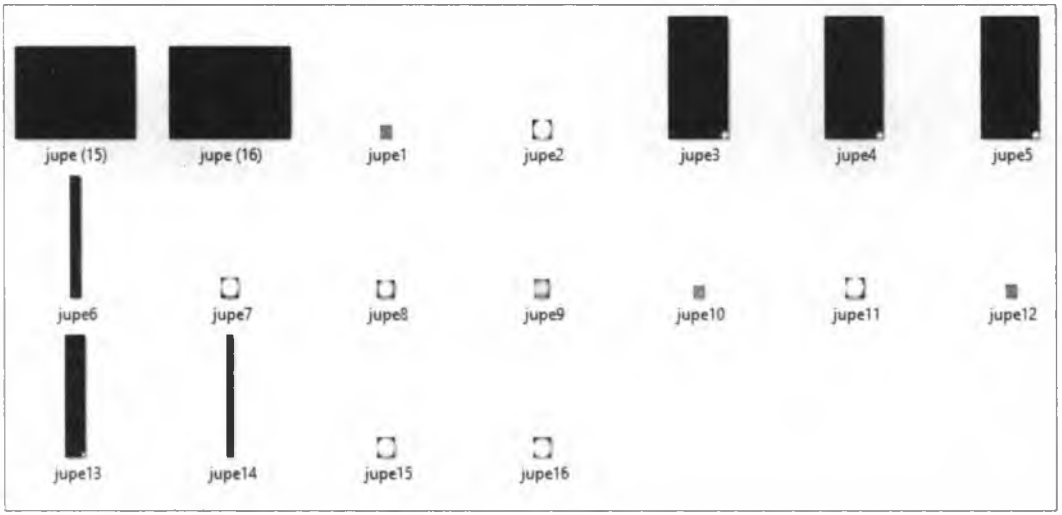


Рис. 15.5. Обрезанные снимки нерегулярного размера из-за проблем с определением размеров ограничительной рамки

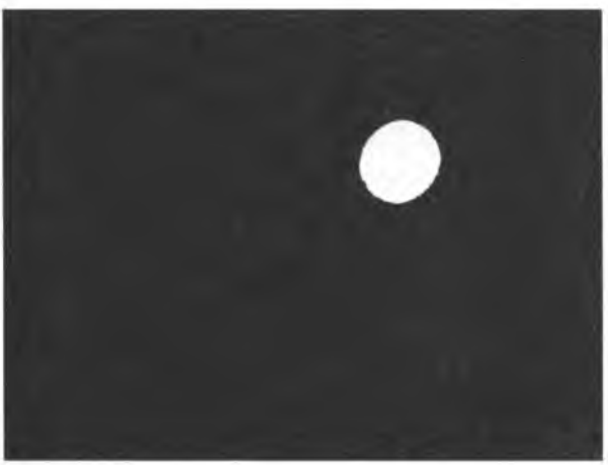


Рис. 15.6. Снимок экрана одного из оригинальных видеокадров, преобразованных в чистый черно-белый цвет

Теперь вызовите метод `getbox()` модуля `Image` на `bw` ❼. Этот метод обрезает черные границы, вписывая ограничительную рамку в ненулевые области снимка. Он возвращает кортеж с левой, верхней, правой и нижней пиксельными координатами рамки.

Если вы примените `box` для обрезки видеокадров, то получите снимок с границами, касательными к поверхности Юпитера (см. средний снимок на рис. 15.7). Это целевой снимок, но он не эстетичный. Поэтому добавьте немного черного заполнения, назначив новую переменную с рамкой, именуемую `padded_box`, чьи края расширены на 20 пикселов во всех четырех направлениях (см. крайний правый снимок на рис. 15.7). Поскольку указанное заполнение является единообразным и применяется ко всем снимкам, оно не ухудшит результаты обрезки.

Рис. 15.7. Начальный кадр, касательный к поверхности Юпитера (рамка), и конечный кадр с заполнением (padded_box)

Продолжите обрезку каждого снимка с помощью метода `crop()` ❽. Этот метод принимает рамку `padded_box` в качестве аргумента.

Для шкалирования снимка примените метод `ImageOps.fit()`. Он берет снимок, размер как кортеж из ширины и высоты в пикселах, метод повторной дискретизации, границу (`0` означает без границы) и даже обрезку от центра, обозначаемую кортежем `(0.5, 0.5)`. Модуль `pillow` имеет несколько вариантов алгоритмов изменения размера изображения, но я выбрал популярный фильтр Ланцоша (Lanczos). Увеличение снимка тяготеет к уменьшению его резкости, но фильтр Ланцоша может создавать артефакты ложного сигнала вдоль сильных краев; это помогает увеличивать воспринимаемую резкость. Указанное непреднамеренное усиление краев помогает взгляду сосредоточиться на вызывающих интерес признаках, которые слабы и размыты в оригинальных видеокадрах.

После шкалирования задайте переменную `file_name`. Каждый из 256 обрезанных снимков будет начинаться с префикса cropped_ и оканчиваться номером снимка, который вы передаете в поле подстановки метода `format()`. Завершите работу функции, сохранив файл ❾.

Вернувшись в глобальную область, добавьте код, позволяющий программе работать как модуль либо в автономном режиме.

ПРИМЕЧАНИЕ

Я храню файлы в формате JPEG, потому что он универсально читается, хорошо обрабатывает градации в цвете и занимает очень мало памяти. Однако JPEG использует сжатие с потерями, которое вызывает легкое ухудшение качества изображения при каждом сохранении файла на диск; степень сжатия можно настроить за счет размера хранилища. В большинстве случаев при работе с астрофотографиями вы захотите использовать один из многих доступных форматов без потерь, таких как TIFF.

В этом месте рабочего потока вы обрезали оригинальные видеокадры в рамку вокруг Юпитера; затем вы отмасштабировали обрезанные снимки в более крупный, единообразный размер (рис. 15.8).

В следующем разделе вы напишете код, который накладывает обрезанные и масштабированные снимки.

Оригинальный видеокадр Обрезанный снимок Масштабированный снимок

Рис. 15.8. Относительные размеры снимков после обрезки и шкалирования

Код наложения снимков

Код stack_images.py берет снимки, созданные предыдущей программой, и усредняет их так, чтобы получился один наложенный снимок. Указанный код можно скачать с веб-страницы книги по адресу **https://www.nostarch.com/impracticalpython/**. Сохраните его в той же папке, что и программу crop_n_scale_images.py.

Листинг 15.5 импортирует модули, загружает снимки, создает списки цветовых каналов (красный, синий, зеленый), усредняет и рекомбинирует каналы, а также создает и сохраняет итоговый наложенный снимок. Он прост настолько, что мы не будем беспокоиться о функции `main()`.

Листинг 15.5. Разбивает и усредняет цветовые каналы, а затем рекомбинирует их в один-единственный снимок. Файл stack_images.py

```
❶ import os
  from PIL import Image

  print("\nначато наложение снимков...")

  # перечислить снимки в каталоге
❷ os.chdir('cropped')
  images = os.listdir()

  # перебрать снимки в цикле и извлечь RGB-каналы как отдельные списки
❸ red_data = []
  green_data = []
  blue_data = []

❹ for image in images:
      with Image.open(image) as img:
          if image == images[0]:  # получить размер первого
                                  # обрезанного снимка
              img_size = img.size # кортеж ширины/высоты
                                  # для использования позже
        ❺ red_data.append(list(img.getdata(0)))
          green_data.append(list(img.getdata(1)))
          blue_data.append(list(img.getdata(2)))
```

```
❺ ave_red = [round(sum(x) / len(red_data)) for x in zip(*red_data)]
  ave_blue = [round(sum(x) / len(blue_data)) for x in zip(*blue_data)]
  ave_green = [round(sum(x) / len(green_data)) for x in zip(*green_data)]

❼ merged_data = [(x) for x in zip(ave_red, ave_green, ave_blue)]
❽ stacked = Image.new('RGB', (img_size))
❾ stacked.putdata(merged_data)
  stacked.show()

❿ os.chdir('..')
  stacked.save('jupiter_stacked.tif', 'TIFF')
```

Начните с повторения нескольких инструкций импорта, которые вы использовали в предыдущей программе ❶. Далее измените текущий каталог на папку cropped, которая содержит обрезанные и отмасштабированные снимки Юпитера ❷, и сразу же в указанной папке сформируйте список снимков, используя для этого метод `os.listdir()`.

С помощью модуля `pillow` можно управлять отдельными пикселами или группами пикселов, и это можно делать для отдельных цветовых каналов, таких как красный, синий и зеленый. В целях демонстрации при наложении снимков вы будете работать с отдельными цветовыми каналами.

Создайте три пустых списка для хранения пиксельных RGB-данных ❸, затем начните обход списка снимков в цикле ❹. Сначала откройте снимок. Затем получите ширину и высоту первого снимка в пикселах как кортеж. Напомним, что в предыдущей программе вы отмасштабировали все малые обрезанные снимки в более крупный размер. Вам понадобятся эти размеры позже для создания нового наложенного снимка, и `size` извлекает эту информацию для вас автоматически.

Теперь примените метод `getdata()`, который получает пиксельные данные выбранного снимка ❺. Указанному методу передайте индекс нужного цветового канала: 0 для красного, 1 для зеленого и 2 для синего. Добавьте результаты в список данных, если это необходимо. Данные из каждого снимка сформируют отдельный список в списках данных.

Для усреднения значений в каждом списке используйте операцию включения в список, которая суммирует пикселы во всех снимках и делит на общее число снимков ❻. Обратите внимание, что вы используете встроенную функцию `zip` с оператором "звездочка" (`*`). Например, список `red_data` является списком списков, причем каждый вложенный список представляет один из 256 снимковых файлов. Использование функции `zip` с оператором `*` распаковывает содержимое списков так, что первый пиксел в image1 суммируется с первым пикселом в image2, и т. д.

Для слияния усредненных цветовых каналов используйте операцию включения в список вместе с функцией `zip` ❼. Далее создайте новый снимок, именуемый `stacked`, используя метод `Image.new()` ❽. Передайте этому методу цветовой режим

('RGB') и кортеж img_size, содержащий нужную ширину и высоту снимка в пикселах, которые были получены ранее из одного из обрезанных снимков.

Заполните новый снимок stacked с помощью метода putdata() и передайте ему список merged_data ❾. Этот метод копирует данные из объекта-последовательности в снимок, начиная с левого верхнего угла (0, 0). Выведите конечный снимок на экран с помощью метода show(). Завершите, поменяв папку на родительский каталог и сохранив снимок в виде файла TIFF с именем stacked ❿.

Если вы сравните один из исходных видеокадров с конечным наложенным снимком (jupiter_stacked.tif), как на рис. 15.9, то увидите явное улучшение в определении границ и соотношении сигнал/шум. Это лучше всего оценить в цвете, поэтому если вы еще не выполнили эту программу, то найдите время на то, чтобы скачать файл Figure 15-9.pdf с веб-сайта. Когда снимок просматривается в цвете, то преимущества наложения проявляются в более гладких, "кремовых" полосах, лучшей определенности красных полос и большей очевидности Большого красного пятна. Однако возможности для улучшения по-прежнему остаются, поэтому далее вы напишете программу для улучшения конечного наложенного снимка.

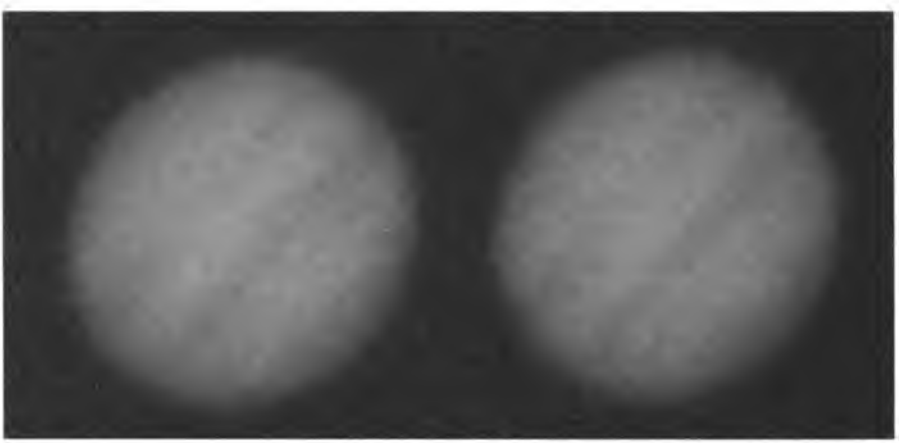


Рис. 15.9. Оригинальный видеокадр в сопоставлении с конечным наложенным снимком (jupiter_stacked.tif)

ПРИМЕЧАНИЕ

Если Большое красное пятно в наложенном снимке кажется вам розоватым, знайте, что так оно и есть! Время от времени оно затухает, и многие опубликованные фотографии Юпитера имеют преувеличенные цвета из-за обработки, поэтому данная едва различимая расцветка теряется. И это, вероятно, к лучшему, т. к. "Большое розовое пятно" звучит как-то странно.

Код улучшения качества снимков

Вы успешно совместили все видеокадры, но Юпитер все еще "кривой", и его признаки не отчетливы. Наложенный снимок можно дополнительно улучшить с помощью фильтров, усилителей и преобразований, имеющихся в pillow. По мере улуч-

шения снимков вы будете уходить все дальше и дальше от "эмпирической истины" сырых данных. По этой причине я решил изолировать процесс улучшения в отдельной программе.

В общем случае первые шаги после наложения состоят в улучшении деталей с помощью фильтров высоких частот или алгоритма нерезкой маски, а затем точной настройки яркости, контрастности и цвета. Данный код будет использовать функционал улучшения изображений модуля `pillow`, применяя эти шаги — хотя и в другом порядке. Указанный программный код можно скачать как enhance_image.py с веб-сайта **https://nostarch.com/impracticalpython/**. Сохраните его в той же папке, что и предыдущие программы Python.

ПРИМЕЧАНИЕ

Обработка астрономических снимков бывает довольно сложной, и на эту тему были написаны целые книги. В данном рабочем потоке некоторые стандартные шаги были опущены. Например, оригинальное видео не было откалибровано, а эффекты искажения из-за турбулентности не были исправлены. Передовое программное обеспечение, такое как RegiStax или AviStack, может предотвратить размытие, деформируя отдельные снимки так, чтобы искаженные объекты, такие как края облачных полос, накладывались надлежаще во всех снимках.

Листинг 15.6 импортирует классы модуля `pillow`, а затем открывает, улучшает и сохраняет наложенный снимок, созданный предыдущим кодом. Поскольку существует много возможных вариантов улучшения снимков, я решил подразделить эту программу на модули, несмотря на ее небольшой размер.

Листинг 15.6. Открывает снимок, улучшает его и сохраняет под новым именем. Файл enhance_image.py

```
❶ from PIL import Image, ImageFilter, ImageEnhance
❺ def main():
      """Получить снимок, улучшить его, показать и сохранить."""
   ❸ in_file = 'jupiter_stacked.tif'
      img = Image.open(in_file)
   ❹ img_enh = enhance_image(img)
      img_enh.show()
      img_enh.save('enhanced.tif', 'TIFF')

❺ def enhance_image(image):
      """Улучшить снимок, используя фильтры и преобразования
         модуля pillow."""
   ❻ enhancer = ImageEnhance.Brightness(image)
   ❼ img_enh = enhancer.enhance(0.75)    # 0.75 выглядит неплохо

   ❽ enhancer = ImageEnhance.Contrast(img_enh)
      img_enh = enhancer.enhance(1.6)
      enhancer = ImageEnhance.Color(img_enh)
      img_enh = enhancer.enhance(1.7)
```

```
    ❾ img_enh = img_enh.rotate(angle=133, expand=True)

    ❿ img_enh = img_enh.filter(ImageFilter.SHARPEN)

      return img_enh

if __name__ == '__main__':
    main()
```

Инструкции импорта уже знакомы, за исключением двух последних ❹. Это новые модули, `ImageFilter` и `ImageEnhance`, которые содержат предопределенные фильтры и классы, которые применяют для изменения изображений с помощью размытия, резкости, осветления, сглаживания и т. д (полный список того, что находится в каждом модуле, см. на веб-странице **https://pillow.readthedocs.io/en/5.1.x/**).

Начните с определения функции `main()` ❷. Назначьте наложенный снимок переменной с именем `in_file`, а затем передайте ее методу `Image.open()`, для того чтобы открыть файл ❸. Далее вызовите функцию `enhance_image()` и передайте ей переменную `image` ❹. Покажите улучшенный снимок, а затем сохраните его в виде файла TIFF, в результате чего качество снимка не будет ухудшено.

Теперь определите функцию улучшения, `enhance_image()`, которая в качестве аргумента принимает изображение ❺. Перефразируя документацию по `pillow`, можно сказать, что все классы улучшения реализуют общий интерфейс, содержащий единый метод `enhance(factor)`, который возвращает улучшенное изображение. Параметр `factor` является значением с плавающей точкой, которое управляет улучшением. Значение, равное `1.0`, возвращает копию оригинала; более низкие значения уменьшают цвет, яркость, контрастность и т. д.; а более высокие его значения указанные качества усиливают.

Для изменения яркости изображения необходимо сначала создать экземпляр класса `Brightness` модуля `ImageEnhance`, передав ему оригинальное изображение ❹. Сымитируйте документацию по `pillow` и назовите этот объект `enhancer`. Для создания окончательного улучшенного снимка вызовите метод `enhance()` объекта и передайте ему аргумент `factor` ❷. В данном случае вы уменьшаете яркость на 0,25. Комментарий `# 0.75` в конце строки кода является полезным способом поэкспериментировать с разными факторами. Используйте этот комментарий для хранения понравившихся значений; благодаря этому вы можете запомнить и восстановить их, если другие тестовые значения не дают приемлемых результатов.

Продолжайте улучшать качество снимка, перейдя к контрасту ❽. Если вы не хотите корректировать контраст вручную, то можете воспользоваться методом автоматической настройки контрастности модуля `pillow`. Сначала импортируйте `ImageOps` из библиотеки PIL. Затем поменяйте две строки, начинающиеся с шага ❷, на одну строку: `img_enh = ImageOps.autocontrast(img_enh)`.

Далее оживите цвет. Это поможет сделать Большое красное пятно заметнее.

Никому неохота смотреть на наклоненный Юпитер, поэтому преобразуйте изображение, повернув его так, чтобы планета приняла более "обычный" вид, где полосы облаков проходят горизонтально, а Большое красное пятно находится справа внизу. Вызовите метод `rotate()` модуля `Image` на снимке и передайте ему угол, измеренный против часовой стрелки в градусах, и он автоматически расширит выходной снимок, сделав его достаточно большим, для того чтобы вместить весь повернутый снимок ❾.

Теперь сделайте снимок резче. Бывает, что даже высококачественные изображения требуют повышения резкости с целью смягчения интерполяционных эффектов преобразования данных, изменения размера, поворота изображений и т. д. Хотя некоторые астрофотографические ресурсы рекомендуют выполнять этот шаг первым, в большинстве рабочих потоков обработки изображений он идет последним. Это связано с тем, что он зависит от конечного размера изображения (расстояния просмотра), а также используемого носителя. Повышение резкости также может увеличить шумовые артефакты и является операцией с потерями, которая может удалить данные, — и все эти шаги вы вряд ли захотите сделать до других изменений.

Повышение резкости немного отличается от предыдущих улучшений качества, т. к. вы используете класс `ImageFilter`. Промежуточный шаг не требуется; новое изображение можно построить одной строкой кода, вызвав метод `filter()` на объекте-изображении и передав ему предопределенный фильтр `SHARPEN` ❿. Модуль `pillow` имеет другие фильтры, помогающие определять края, такие как `UnsharpMask` и `EDGE_ENHANCE`, но для этого снимка результаты из `SHARPEN` неразличимы.

Завершите работу, вернув снимок и применив код выполнения программы в виде модуля либо в автономном режиме.

Окончательный улучшенный снимок сравнивается со случайным видеокадром и конечным наложенным снимком на рис. 15.10. Для удобства сравнения все снимки были повернуты.

Улучшение качества заметнее всего на цветном снимке. Если вы хотите перед выполнением программы посмотреть на цветную версию, обратитесь к файлу Figure 15-10.pdf, который можно скачать с веб-сайта книги.

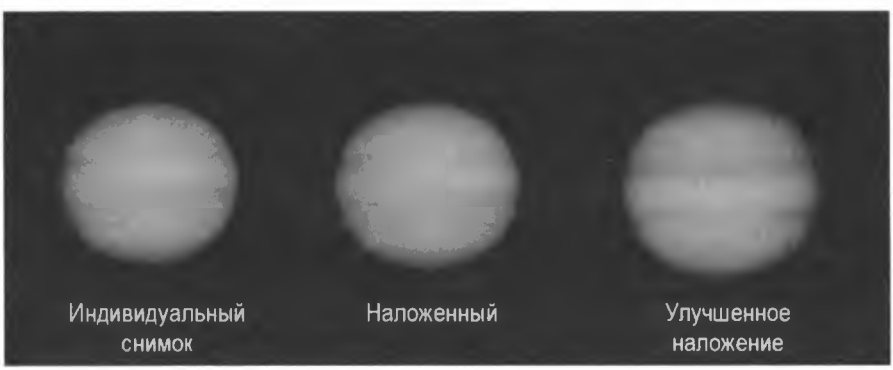


Рис. 15.10. Случайный видеокадр, результаты наложения 256 кадров и итоговый улучшенный снимок

ПРИМЕЧАНИЕ

Если вы знакомы с `pillow`, то, вероятно, знаете, что совместить снимки можно несколькими строками кода с помощью метода `Image.blend()`. На мой взгляд, однако, результирующий снимок получается заметно шумнее, чем тот, что получается при подразделении и усреднении отдельных цветовых каналов, как вы сделали с помощью программы stack_images.py.

Резюме

Окончательный снимок на рис. 15.10 никогда не получит никаких наград или не будет показан в журнале Sky & Telescope, но смысл состоял в том, чтобы принять вызов. И результатом является заметное улучшение его качества по сравнению со снимком, снятым на видео. Цвета получились ярче, полосы облаков резче и Большое красное пятно, определенно, лучше. Также можно разглядеть турбулентную зону с подветренной стороны Большого красного пятна (см. рис. 15.1).

Невзирая на то что вы начали с грубых входных данных, вы смогли выполнить регистрацию снимков, удалить шум с помощью наложения и улучшить качество конечного снимка посредством фильтров и преобразований. И все эти шаги были выполнены с помощью общедоступной ветки библиотеки обработки изображений Python (PIL) под названием `pillow`. Вы также приобрели опыт работы с модулями Python `shutil` и `os`, которые вы использовали для управления файлами и папками.

В целях более продвинутой обработки изображений можно воспользоваться общедоступной библиотекой методов компьютерного зрения OpenSource Computer Vision (OpenCV), которая реализуется путем установки и импорта `cv2` и `NumPy`. Другие варианты включают библиотеки `matplotlib`, `SciPy` и `NumPy`. Как водится с языком Python, существует не один способ снять кожуру с плода!

Дальнейшее чтение

Книга Эла Свейгарта "Автоматизируйте скучные вещи с помощью Python" (Sweigart A. Automate the boring stuff with Python. No Cramp Press, 2015) содержит несколько полезных глав по работе с файлами, папками и модулем `pillow`.

Среди онлайновых ресурсов для использования Python с астрономией имеются "Python для астрономов" (Python for astronomers, **https://prappleizer.github.io/**) и "Практический Python для астрономов" (Practical Python for astronomers, **https://python4astronomers.github.io/**).

Если вы хотите узнать о библиотеке OpenCV-Python побольше, то соответствующие учебники можно найти по адресу **https://docs.opencv.org/3.4.2/d0/de3/tutorial_py_intro.html**. Обратите внимание, что знание библиотеки `NumPy` является для указанных учебных пособий и для написания оптимизированного программного кода OpenCV обязательным условием. Кроме того, библиотека SimpleCV позволяет начать работу с компьютерным зрением и манипулировать изображениями с меньшей кривой усвоения, чем OpenCV, но работает только с Python 2.

Книга Тьерри Лего "Астрофотография" (Legault T. Astrophotography. Rocky Nook, 2014) является незаменимым ресурсом для всех, кто интересуется серьезной астрофотографией. Будучи всесторонним и читабельным справочником, она охватывает все аспекты данного предмета, от выбора оборудования до обработки изображений.

Блогпост "Выравнивание изображений Солнца с помощью Python" (Aligning Sun images using Python. LabJG, 2013) в блоге Джеймса Гилберта (James Gilbert) содержит код для обрезки изображений Солнца с помощью метода ограничительной рамки. Он также включает в себя умный метод повторного выравнивания повернутых изображений Солнца с использованием солнечных пятен в качестве точек регистрации. Данную статью можно найти по адресу **https://labjg.wordpress.com/2013/04/01/aligning-sun-images-using-python/**.

Исследовательская группа Google выяснила, как использовать наложение для удаления водяных знаков с изображений из веб-сайтов стоковой фотографии и как вебсайты могут лучше защитить свою собственность. Прочитать об этом можно по адресу **https://research.googleblog.com/2017/08/making-visible-watermarks-more-effective.html**.

Сложный проект: акт исчезновения

Методы наложения снимков способны делать больше, чем просто удалять шум, — они способны удалять все, что движется на фотосайте, в том числе людей. Adobe Photoshop, например, имеет скрипт наложения, который заставляет нестационарные объекты волшебным образом исчезать. Он опирается на статистическое среднее, именуемое *медианой*, которое является просто "срединным" значением в списке чисел, расположенных от наименьшего до наибольшего. Этот процесс требует нескольких фотографий — предпочтительно сделанных с помощью камеры, установленной на штативе, — для того чтобы объекты, которые вы хотите удалить, меняли положение от одного изображения к другому, в то время как фон оставался постоянным. Как правило, требуется от 10 до 30 снимков, сделанных с интервалом около 20 секунд, либо аналогично разнесенные кадры, извлеченные из видеоролика.

С помощью арифметического среднего вы суммируете числа и делите их на общую сумму. С помощью медианы вы сортируете числа и выбираете значение ровно посередине. На рис. 15.11 показан ряд из пяти снимков с одним и тем же расположением пикселов, выделенным в каждом из них. На четвертом снимке пролетел черный дрозд и испортил великолепный белый фон. Если вы совместите со средним значением, то присутствие птицы сохранится. Но стоит применить на этих снимках медианное наложение, т. е. отсортировать красный, зеленый и синий каналы и взять срединные значения, и вы получите значение фона для каждого канала (255). От птицы не осталось и следа.

При усреднении с помощью медианы ложные значения перемещаются в конец списка. Этот прием позволяет легко удалять выбросы, такие как спутники или самолеты в астрофотоснимках, до тех пор, пока число снимков, содержащих выбросы, меньше половины числа снимков.

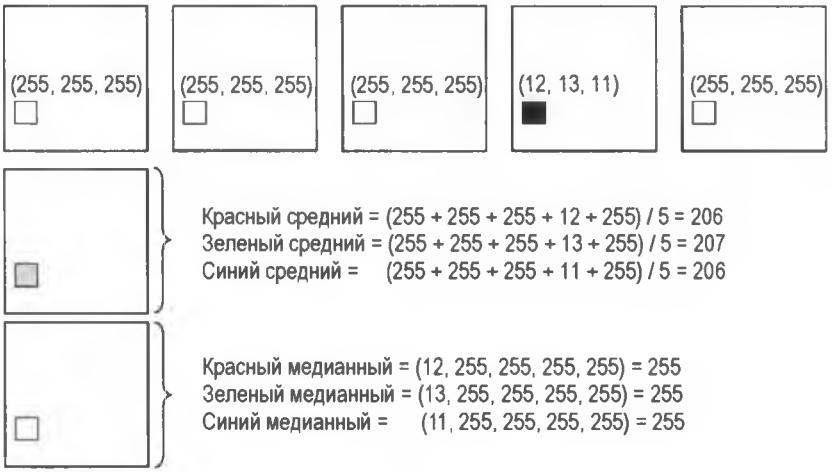


Рис. 15.11. Пять белых снимков, в которых выделен один и тот же пиксел и показаны значения RGB Медианное наложение удаляет черный пиксел

Рис. 15.12. Искусственные фотографии Луны для тестирования подхода на основе медианного усреднения

Вооружившись этим знанием, напишите программу наложения снимков, которая удалит нежелательных туристов из ваших отпускных фотомоментов. Для тестирования вы можете скачать папку moon_cropped с веб-сайта, которая содержит пять искусственных снимков Луны, каждый из которых "испорчен" пролетающим самолетом (рис. 15.12).

Ваш окончательный наложенный снимок не должен содержать никаких признаков самолета (рис. 15.13).

Поскольку этот проект является сложным, никакого решения не предусмотрено.

Рис. 15.13. Результат наложения снимков в папке moon_cropped с использованием медианного усреднения

16

ВЫЯВЛЕНИЕ МОШЕННИЧЕСТВА С ПОМОЩЬЮ ЗАКОНА БЕНФОРДА

До изобретения электронных калькуляторов если нужно было взять логарифм числа, то его искали в справочной таблице. Астроном Саймон Ньюкомб (Simon Newcomb) использовал такие таблицы и в 1881 г. заметил, что страницы в начале справочника с числами, начинающимися с самых маленьких цифр, были истрепаны больше, чем в конце справочника. Из этого приземленного наблюдения он понял — что, по крайней мере, для измерений и констант в природе — первые значащие цифры в числах с гораздо большей вероятностью были малыми, чем большими. Он опубликовал об этом короткую статью и занялся другими делами.

На протяжении десятилетий эта статистическая диковинка, как и одно кольцо Толкиена, "выпало из поля зрения". Затем, в 1938 г., физик Фрэнк Бенфорд (Frank Benford) заново открыл и подтвердил это явление, собрав более 20 тыс. образцов реальных чисел, используя такие разнообразные источники данных, как протяженность рек, уличные адреса, числа, содержащиеся в журнале Reader's Digest, молекулярные массы, бейсбольная статистика, смертность и многое другое. Как человек, который популяризировал это научное открытие, он получил все возможные почести и поощрения.

Согласно *закону Бенфорда*, также именуемому *законом первой цифры*, частота появления первых цифр в естественно возникающих числовых распределениях предсказуема и неравномерна. На самом деле, отдельно взятое число начинается в шесть раз чаще с 1, чем с 9! Это противоречит интуитивному восприятию, поскольку большинство людей ожидали бы равномерного распределения, причем каждое число имеет один из девяти (11,1%) шансов занять первое место. Из-за этого когнитивного диссонанса закон Бенфорда стал полезным инструментом обнаружения мошенничества в финансовых, научных и избирательных данных.

В этой главе вы напишете программу на языке Python, которая сравнивает реальные наборы данных с законом Бенфорда и определяет, выглядят ли они мошенническими или нет. Вы также еще разок отряхнете пыль с `matplotlib`, для того чтобы добавить в анализ полезную визуальную компоненту. В качестве набора данных вы будете использовать голоса, поданные на спорных выборах президента США 2016 года.

Проект 24: закон первых цифр Бенфорда

На рис. 16.1 показан столбчатый график первых значащих цифр в множестве чисел, подчиняющихся закону Бенфорда. Что удивительно, шкала не имеет значения. Сведéние в таблицу длин австралийских дорог будет подчиняться закону Бенфорда, будь то в милях, километрах или локтях! Как статистический принцип, он является инвариантным к шкале.

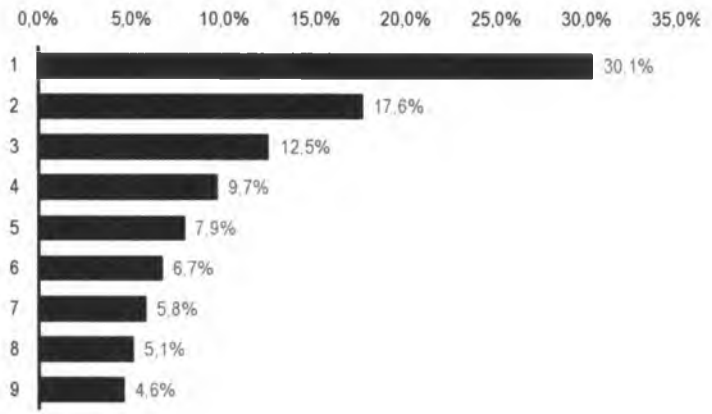


Рис. 16.1. Частота появления первых цифр в соответствии с законом Бенфорда

Математикам потребовалось около 100 лет на то, чтобы найти удовлетворительное объяснение закону Бенфорда. Для остальных из нас давайте просто скажем, что во Вселенной существует больше маленьких вещей, чем больших. Фрэнк Бенфорд использовал аналогию с землей, что легче владеть одним акром земли, чем девятью. По сути дела, можно в точности воспроизвести частоту, полученную по закону Бенфорда, просто допустив, что существует в два раза больше единиц, чем двоек, в три раза больше, чем троек и т. д. Просто берут инверсию каждой из девяти цифр ($1/d$) и делят на сумму всех инверсий (2,83). Затем результаты умножают на 100, получая процент (рис. 16.2).

Из-за только что рассмотренных соотношений размеров закон Бенфорда можно визуализировать с помощью логарифмической шкалы, которая используется для вычерчивания данных, связанных экспоненциальной зависимостью. В полулогарифмических графиках одна переменная, как правило, ограничивается, подобно множеству первых цифр (1–9), тогда как другая охватывает широкий диапазон значений, включающий несколько порядков величины.

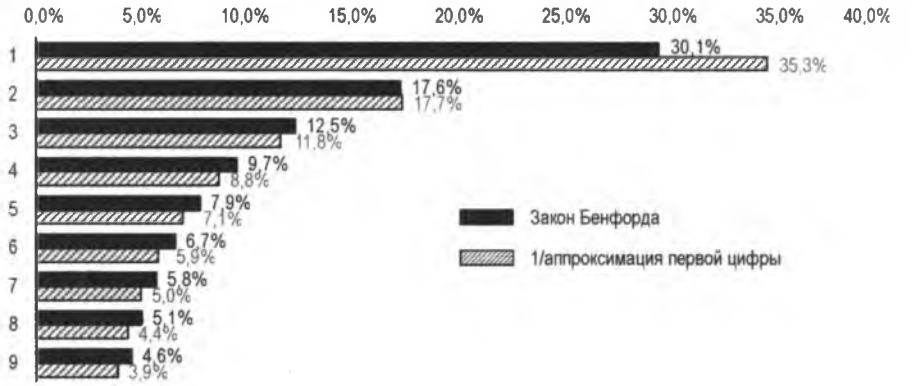


Рис. 16.2. Сравнение закона Бенфорда с аппроксимацией, пропорциональной обратному значению первой цифры

На полулогарифмической миллиметровой бумаге значения по горизонтальной оси x являются логарифмическими, а значения по вертикальной оси y, которые представлены горизонтальными линиями, таковыми не являются (рис. 16.3). На оси x горизонтальные деления располагаются нерегулярно, и эта нелинейная картина повторяется со степенями 10. Для каждых десяти разделов на логарифмической бумаге, например с 1 до 10 или с 10 до 100, ширина делений между числами пропорциональна длине столбиков на рис. 16.1. Например, расстояние между 1 и 2 на рис. 16.3 составляет 30,1% от расстояния между 1 и 10. Как выразился один автор, закон Бенфорда можно вывести, просто бросив дротики в кусок логарифмической бумаги!

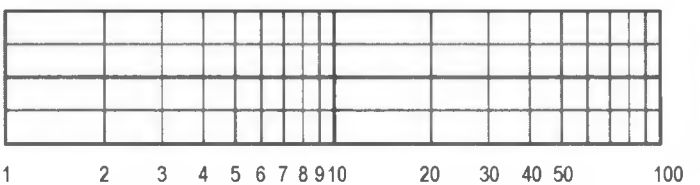


Рис. 16.3. Пример полулогарифмической миллиметровой бумаги с двумя десятками делений

Для того чтобы коллекция числовых данных соответствовала закону Бенфорда, должны быть соблюдены определенные условия. Числа должны быть случайными и незаданными, без навязанных минимумов или максимумов. Числа должны охватывать несколько порядков величины, и набор данных должен быть крупным; рекомендации в литературе требуют от 100 до 1000 образцов как минимум, хотя закон Бенфорда, как было показано, соблюдается для наборов данных, содержащих всего 50 чисел. Примеры распределений, которые не будут подчиняться закону Бенфорда, включают показатели роста профессиональных баскетболистов, телефонные номера США (в которых только последние четыре цифры являются действительно случайными), цены, на которые влияют психологические барьеры ($ 1,99 против $ 2,00), и медицинские доплаты.

Применение закона Бенфорда

Большинство финансовых и бухгалтерских данных согласуется с числами, появляющимися естественным путем, и, таким образом, подчиняются закону Бенфорда. Например, допустим, что вы владеете акцией взаимного инвестиционного фонда стоимостью $ 1000. Для того чтобы ваш депозит достиг $ 2000 по стоимости, он должен удвоиться, увеличившись на 100%. Для того чтобы вырасти с $ 2000 до $ 3000, ему нужно вырасти только на 50%. Для того чтобы первая цифра была 4, она должна вырасти еще на 33%. Как предсказывает закон Бенфорда, для того чтобы первая цифра 1 стала 2, необходимо больше роста, чем для того, чтобы 3 стала 4, и т. д. Поскольку распределение Бенфорда является "распределением распределений", наборы финансовых данных, как правило, согласуются, поскольку они являются результатом комбинирования чисел, хотя бывают и исключения.

Поскольку о законе Бенфорда люди обычно не осведомлены, они его не учитывают, когда фальсифицируют числовые записи. Это дает судебным бухгалтерам мощный инструмент для быстрого выявления наборов данных, которые могут быть мошенническими. В действительности, в США сличения с законом Бенфорда допустимы юридически в качестве доказательств в уголовных делах на федеральном, государственном и местном уровнях.

В деле 1993 г. "Штат Аризона против Нельсона" обвиняемый увел почти 2 млн долларов фиктивным поставщикам в попытке обмануть государство. Несмотря на то что он позаботился о том, чтобы фальшивые чеки выглядели легитимно, распределение первой цифры явно нарушало закон Бенфорда (рис. 16.4), что привело к осуждению обвиняемого.

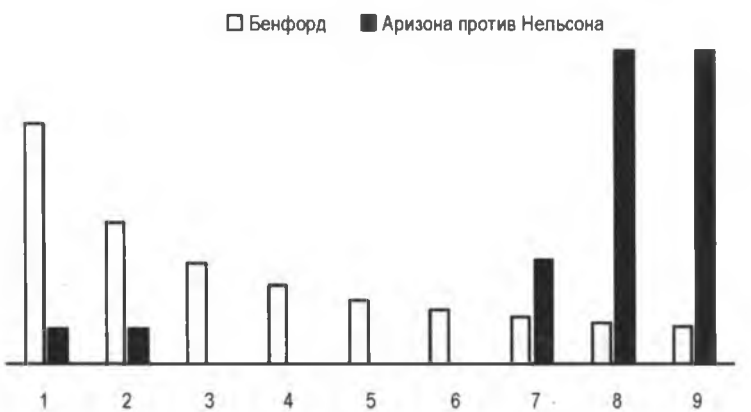


Рис. 16.4. Сличение частоты первых цифр в мошеннических чеках с ожидаемой частотой закона Бенфорда, штат Аризона против Уэйна Джеймса Нельсона (CV92-18841)

Закон Бенфорда также полезен для внутрикорпоративного аудита. Представьте себе случай, когда любые расходы на поездки и развлечения свыше $ 10 000 должны быть одобрены вице-президентом компании. Этот тип финансового порога может побудить сотрудников разделять счета-фактуры с целью обыграть систему. Рис. 16.5 основан на группе расходов в диапазоне от $ 100 до $ 12 000, где все зна-

чения, превышающие $ 9999, были разделены на две равные части. Как вы можете догадаться, вокруг 5 и 6 имеется всплеск частот первых цифр, что является явным нарушением закона Бенфорда.

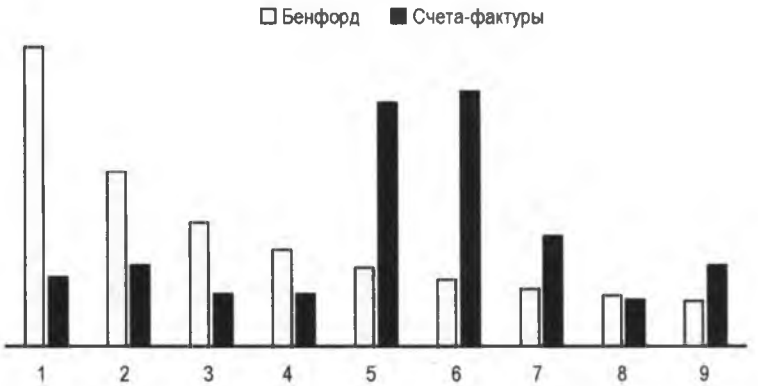


Рис. 16.5. Разделение счетов-фактур свыше $ 9999 для набора данных, варьирующихся в диапазоне от $ 100 до $ 12 000, нарушает закон Бенфорда

В более широком масштабе закон Бенфорда позволил выявить нарушения в финансовых данных — таких как суммы доходов — крупных корпораций. Пример обработки данных компании Enron, которая практиковала институциональное бухгалтерское мошенничество, показан на рис. 16.6. Банкротство Enron в 2001 г. было крупнейшим в истории в то время, и оно привело к тюремному заключению нескольких топ-менеджеров. Скандал также привел к роспуску фирмы Arthur Andersen, бухгалтерской фирмы "Большой пятерки" и одной из крупнейших в мире транснациональных корпораций.

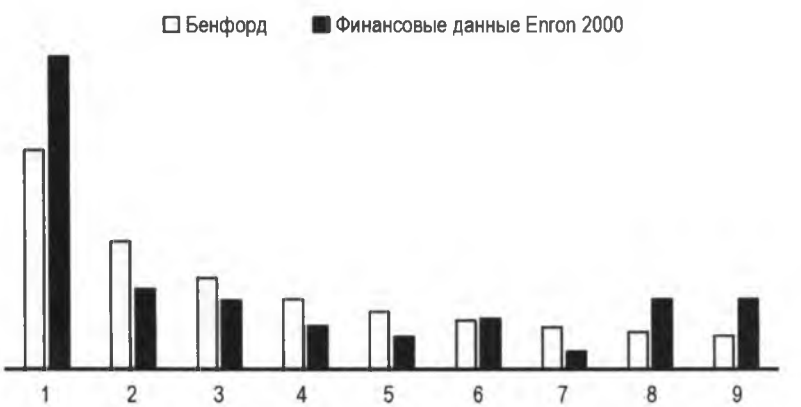


Рис. 16.6. Частота первых цифр из финансовых данных Enron за 2000 г. по сравнению с ожидаемой частотой, основанной на законе Бенфорда (по данным The Wall Street Journal)

Совершенно очевидно, что закон Бенфорда лучше всего работает для обнаружения мошенничества, когда преступники об этом не знают. Если вы знаете, как работает закон, то вы можете обмануть его, что мы и сделаем в практическом проекте в конце этой главы. В итоге закон Бенфорда можно использовать для отметки флажками

наборов данных, которые могли бы быть мошенническими, но вы не сможете это использовать для доказательства обратного.

Проверка по статистическому показателю хи-квадрат

Для верификации того, что набор данных подчиняется закону Бенфорда, аудиторы и исследователи используют несколько статистических методов. В этом проекте вы будете применять проверку степени соответствия по *статистическому показателю хи-квадрат*, которая является широко используемым методом определения того, отличается ли эмпирическое (наблюдаемое) распределение значимо от теоретического (ожидаемого) распределения. В качестве дискриминатора используется уровень значимости, или *p*-значение. Наиболее распространенным уровнем значимости является 0,05, но другие распространенные включают 0,01 и 0,10. Уровень значимости 0,05 указывает на 5-процентный риск ошибочного заключения о том, что разница существует тогда, когда она не существует.

Вот шаги, необходимые для выполнения проверки степени соответствия по статистическому показателю хи-квадрат:

1. Найти степени свободы (*df*), определяемые как число категорий (*k*) минус 1:

$$df = k - 1.$$

   Для закона Бенфорда категориальными уровнями являются первые цифры (1–9), поэтому $df = 8$.

2. На каждом уровне вычислить ожидаемую частотность (частотное количество) путем умножения размера выборки на теоретические пропорции на каждом уровне:

$$E_i = np_i,$$

   где E_i — это ожидаемая частота на *i*-м уровне; n — размер выборки; p_i — теоретическая вероятность на *i*-м уровне. Для 1000 образцов число образцов, которые, как ожидается, начнутся с 1 в распределении закона Бенфорда, будет равно $1000 \cdot 0{,}301 = 301$ (см. рис. 16.1).

3. Вычислить случайную величину хи-квадрат (χ^2), также именуемую *проверочным статистическим показателем*, или *проверочной статистикой*, который позволит вам судить, являются ли два распределения одинаковыми:

$$\chi^2_{df} = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i},$$

   где O_i — это наблюдаемая частотность для *i*-го уровня категориальной переменной; E_i — ожидаемая частотность для *i*-го уровня категориальной переменной; *df* — степени свободы.

4. Обратиться к таблице распределения хи-квадрат (табл. 16.1), читая поперек строки, которая соответствует вычисленным степеням свободы. Если проверочный статистический показатель меньше значения, указанного в столбце *p*-зна-

чения, считающегося значимым, то вы не можете отклонить гипотезу о том, что наблюдаемое и теоретическое распределения являются одинаковыми.

Таблица 16.1. *Таблица распределения хи-квадрат*

Степени свободы	Вероятность превышения критического значения								
	0,99	**0,95**	**0,90**	**0,75**	**0,50**	**0,25**	**0,10**	**0,05**	**0,01**
1	0,000	0,004	0,016	0,102	0,455	1,32	2,71	3,84	6,63
2	0,020	0,103	0,211	0,575	1,386	2,77	4,61	5,99	9,21
3	0,115	0,352	0,584	1,212	2,366	4,11	6,25	7,81	11,34
4	0,297	0,711	1,064	1,923	3,357	5,39	7,78	9,49	13,28
5	0,554	1,145	1,610	2,675	4,351	6,63	9,24	11,07	15,09
6	0,872	1,635	2,204	3,455	5,348	7,84	10,64	12,59	16,81
7	1,239	2,167	2,833	4,255	6,346	9,04	12,02	14,07	18,48
8	1,647	2,733	3,490	5,071	7,344	10,22	13,36	15,51	20,09
9	2,088	3,325	4,168	5,899	8,343	11,39	14,68	16,92	21,67
10	2,558	3,940	4,865	6,737	9,342	12,55	15,99	18,31	23,21
	Не значимы							Значимы	

В табл. 16.2 критическое значение для *p*-значения, равного 0,05 при 8 степенях свободы, составляет 15,51. Если ваш вычисленный проверочный статистический показатель меньше 15,51, то соответствующее *p*-значение превышает 0,05, и вы пришли бы к выводу, что статистически значимой разницы между наблюдаемым распределением и предсказанным законом Бенфорда нет. Здесь *p*-значение — это вероятность того, что проверочный статистический показатель, имеющий 8 степеней свободы, является более экстремальным, чем 15,51.

Обратите внимание, что проверка на основе статистического показателя хи-квадрат должна выполняться на *количествах*. Если данные представлены в процентах, средних значениях, соотношениях и т. д., то перед выполнением проверки необходимо конвертировать значения в количества.

Цель

Написать программу на языке Python, которая загружает числовые данные, записывает частоту появления первых цифр, сравнивает их с законом Бенфорда с помощью проверки степени соответствия на основе статистического показателя хи-квадрат и показывает результат сравнения как в табличной, так и в графической форме.

Набор данных

Президентские выборы 2016 г. в США изобиловали обвинениями в мошенничестве с голосами избирателей. Самые известные из них касались якобы вмешательства российской стороны в выборы и подтасовки голосов в пользу Дональда Трампа,

а Демократический национальный комитет был обвинен в том, что в процессе выдвижения этой партии он предпочел Хиллари Клинтон Берни Сандерсу. Президент Трамп также утверждал, что от 5 до 6 млн человек проголосовали незаконно, и в мае 2017 г. он подписал исполнительный приказ, который создал комиссию по рассмотрению случаев мошенничества с избирательными голосами и принуждению избирателей.

В этом проекте вы будете использовать набор данных результатов голосования на президентских выборах 2016 г. Он состоит из окончательных голосов по округам для 102 округов в штате Иллинойс, где победила Хиллари Клинтон. Начиная с июня 2016 г. база данных системы регистрации избирателей штата Иллинойс стала жертвой вредоносной кибератаки неизвестного происхождения. Власти штата Иллинойс подтвердили, что хакеры получили доступ к тысячам записей, но, по-видимому, не изменили никаких данных.

В избирательном бюллетене штата Иллинойс было удивительно много кандидатов на пост президента, поэтому набор данных был обработан так, чтобы включать только Хиллари Клинтон, Дональда Трампа, Гэри Джонсона и Джилл Стайн. Голоса за этих кандидатов были собраны в 408-строчный текстовый файл со следующими ниже первыми пятью строками:

```
962
997
1020
1025
1031
```

Полный список кандидатов и голосов можно найти онлайн по адресу **https://www.elections.il.gov/ElectionInformation/DownloadVoteTotals.aspx**.

В этом проекте нужны только голоса, которые можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/** как файл Illinois_votes.txt. Вам нужно будет сохранить этот файл в той же папке, что и ваш программный код Python.

Стратегия

Давайте притворимся, что вы являетесь следователем, изучающим обвинения в мошенничестве с избирателями на президентских выборах 2016 г. и вас направили в штат Иллинойс. Прежде чем глубоко погрузиться в данные, вы захотите отметить любые очевидные аномалии. Закон Бенфорда не может помочь вам выявить, голосуют ли люди незаконно, но он является хорошей отправной точкой для обнаружения фальсификации голосов, т. е. изменения количества голосов после их подачи.

В таких случаях способность сообщать результаты так же важна, как и количественный анализ. В состав избирательных комиссий входят не только эксперты, но и многие непрофессионалы с ограниченными знаниями в области статистики. И в состав жюри наверняка не войдет ни один эксперт. Для того чтобы убедить себя и других в том, что подсчеты голосов были допустимыми (или нет), вы захотите

показать несколько сравнений, таких как таблица, график и количественная переменная хи-квадрат (проверочный статистический показатель).

Индивидуальные шаги, участвующие в анализе, хорошо поддаются инкапсуляции в функции. Поэтому вместо псевдокода давайте посмотрим, какие функции вам могут понадобиться:

- `load_data()` — загружает данные в виде списка;
- `count_first_digits()` — табулирует первые цифры в наблюдаемой сумме голосов каждого округа;
- `get_expected_counts()` — определяет количество для каждой первой цифры, как предсказано Бенфордом;
- `chi_square_test()` — выполняет проверку степени соответствия по статистическому показателю хи-квадрат на наблюдаемых и ожидаемых значениях;
- `bar_chart()` — создает столбчатый график для сравнения наблюдаемых процентов первых цифр с ожидаемыми;
- `main()` — получает имя файла набора данных, вызывает функции и печатает статистику.

Код

В этом разделе вы будете использовать программный код benford.py, предназначенный для расследования мошенничества с избирателями, но он достаточно гибок и может использоваться в любом наборе данных, где были подсчитаны категориальные значения, такие как результаты медицинских анализов, поступления от подоходного налога или возврат средств клиентам. Также возможны приложения, не связанные с мошенничеством, такие как обнаружение неэффективности процесса, вызванного большим числом транзакций с низкой стоимостью, проблемы со сбором и обработкой данных, такие как отсутствующие данные, усеченные значения или опечатки, и систематическое смещение в стратегиях измерения или статистических исследованиях, такие как предпочтение выборок, соответствующих только наилучшему или наихудшему случаю.

Указанный код можно скачать с веб-сайта

https://www.nostarch.com/impracticalpython/.

Вам также понадобится текстовый файл Illinois_votes.txt, описанный в *разд. "Набор данных" ранее в этой главе*.

Импортирование модулей и загрузка данных

Листинг 16.1 импортирует модули и определяет функцию загрузки данных. В этом проекте вы будете использовать данные в виде текстового файла со знаком табуляции в качестве разделителей, экспортированного из Microsoft Excel, который вы загрузите как список символьных цепочек.

Листинг 16.1. Импортирует модули и определяет функцию загрузки данных. Файл benford.py, часть 1

```
import sys
import math
❶ from collections import defaultdict
❷ import matplotlib.pyplot as plt

# проценты закона Бенфорда для первых цифр 1-9
❸ BENFORD = [30.1, 17.6, 12.5, 9.7, 7.9, 6.7, 5.8, 5.1, 4.6]

❹ def load_data(filename):
    """Открыть текстовый файл и вернуть список символьных цепочек."""
  ❺ with open(filename) as f:
        return f.read().strip().split('\n')
```

Бóльшая часть инструкций импорта на данный момент должна быть уже знакома. Модуль `collections` предоставляет специализированные альтернативы стандартным контейнерам Python, таким как множества, кортежи, списки и словари ❶. Для подсчета частот первой цифры вам понадобится контейнерный тип `defaultdict`, т. е. подкласс словаря `dict`, который для предоставления отсутствующих значений вызывает фабричную функцию. С помощью контейнерного типа `defaultdict` можно построить словарь, используя цикл, и он автоматически создаст новые ключи и не выдаст ошибку. Он возвращает объект-словарь.

Заключительная инструкция импорта предназначена для построения графика с помощью библиотеки `matplotlib` ❷. Дополнительные сведения о `matplotlib` и ее установке *см. в разд. "Код вероятности обнаружения" главы 10.*

Теперь объявите переменную для списка, содержащего проценты закона Бенфорда, упорядоченные от 1 до 9 ❸. Затем определите функцию чтения текстового файла и верните список ❹. Как и раньше, используйте инструкцию `with`, т. к. она автоматически закроет файл, когда это будет сделано ❺.

Подсчет первых цифр

В листинге 16.2 определена функция подсчета первых цифр и сохранения результатов в словарной структуре данных. Окончательные количества и частота каждого количества в процентах возвращаются в виде списков для использования в последующих функциях. Указанная функция также выполняет контроль качества данных, и если она обнаруживает плохой образец, извещает об этом пользователя и закрывает программу.

Листинг 16.2. Определяет функцию подсчета первых цифр в списке и возвращает абсолютные количества и частоту. Файл benford.py, часть 2

```
❶ def count_first_digits(data_list):
    """Подсчитать 1-е цифры в списке чисел; вернуть количества
       и частоту."""
```

```
❷ first_digits = defaultdict(int)     # целочисленное значение
                                      # по умолчанию равно 0

❸ for sample in data_list:
    ❹ if sample == '':
          continue
      try:
          int(sample)
      except ValueError as e:
          print(e, file=sys.stderr)
          print("Образцы должны быть целыми числами. Завершение работы",
                                                        file=sys.stderr)
          sys.exit(1)
    ❺ first_digits[sample[0]] += 1

  # проверить отсутствие цифр
  keys = [str(digit) for digit in range(1, 10)]
  for key in keys:
      if key not in first_digits:
          first_digits[key] = 0

❻ data_count = [v for (k, v) in sorted(first_digits.items())]
  total_count = sum(data_count)
  data_pct = [(i / total_count) * 100 for i in data_count]
❼ return data_count, data_pct, total_count
```

Функция `count_first_digits()` в качестве аргумента принимает список символьных цепочек, возвращаемых функцией `load_data()` ❶. Вы будете вызывать обе функции в главной функции `main()`.

Создайте словарь с именем `first_digits`, используя `defaultdict` ❷. Этот шаг просто задает словарь для последующего заполнения. Первый аргумент в `defaultdict` является любым вызываемым типом (без аргументов). В этом случае вызываемым типом является конструктор целочисленного типа `int`, т. к. вы хотите подсчитывать целые числа. При использовании `defaultdict` всякий раз, когда операция обнаруживает отсутствующий ключ, вызывается функция без аргументов `default_factory`, а выходные данные используются в качестве значения. Любой несуществующий ключ получает значение, возвращенное функцией `default_factory`, и никакой ошибки `KeyError` не возникает.

Теперь начните цикл `for` и просмотрите образцы в `data_list` ❸. Если образец пуст (т. е. если текстовый файл содержит пустую символьную цепочку ❹), то проигнорируйте его, используя инструкцию `continue`. В противном случае используйте инструкцию `try` для конвертации образца в целое число. Если возникает исключение, то образец не является правильным значением количества, поэтому сообщите об этом пользователю и выйдите из программы. В следующем ниже примере результата входной файл содержит вещественное значение (`0.01`), и функция `main()` печатает имя файла.

```
Имя файла с КОЛИЧЕСТВЕННЫМИ данными: bad_data.txt
Недопустимый литерал для int() с базой 10: '0.01'
Образцы должны быть целыми числами. Завершение работы.
```

Если образец проверку проходит, то сделайте его первый элемент (начальную цифру) ключом словаря и добавьте значение 1 ❺. Поскольку вы применили контейнерный тип defaultdict с типом `int`, ключ с самого начала получает значение по умолчанию 0 на лету.

Как и все словари Python, словарь `first_digits` не упорядочен. Для того чтобы сравнить количества с распределением закона Бенфорда, вам нужны ключи, перечисленные в числовом порядке, поэтому примените операцию включения в список и сортировку с целью создать новую версию словаря `first_digits` под именем `data_count` ❻. В результате получатся значения, отсортированные по ключу, как показано ниже:

```
[129, 62, 45, 48, 40, 25, 23, 21, 15]
```

Далее просуммируйте количества, затем создайте новый список и конвертируйте количества в проценты. Завершите функцию, вернув эти два списка и просуммированные количества ❼. Поскольку количества в списках сортируются от 1 до 9, ассоциированная первая цифра вам не понадобится — она неявно присутствует в упорядочении.

Получение ожидаемых количеств

В листинге 16.3 определена функция `get_expected_counts()`, которая принимает наблюдаемые данные и вычисляет ожидаемые значения для первых цифр, основываясь на законе Бенфорда. Эти ожидаемые количества возвращаются в виде списка, который вы будете использовать позже вместе с проверкой степени соответствия по статистическому показателю хи-квадрат с целью увидеть, насколько хорошо наблюдаемые данные соответствуют закону Бенфорда.

Листинг 16.3. Определяет функцию, которая вычисляет ожидаемые по закону Бенфорда значения для набора данных. Файл benford.py, часть 3

```
❶ def get_expected_counts(total_count):
      """Возвращает список ожидаемых по закону Бенфорда значений
         для суммарного количества образцов."""
   ❷ return [round(p * total_count / 100) for p in BENFORD]
```

Аргументом этой функции является суммированное количество, возвращенное из функции `count_first_digits()` листинга 16.2 ❹. Для получения количеств, которые вы ожидаете для закона Бенфорда, вам нужно будет использовать частотную вероятность каждой цифры, поэтому конвертируйте проценты списка `BENFORD`, разделив на 100. Затем умножьте переменную `total_count` на эту вероятность. Все это можно сделать с помощью операции включения в список как часть инструкции `return` ❷.

Определение степени соответствия

Листинг 16.4 определяет функцию, которая реализует проверку хи-квадрат, описанную в *разд. "Проверка по статистическому показателю хи-квадрат" ранее в этой главе*. Эта проверка вычисляет соответствие наблюдаемых количеств ожидаемым количествам, предсказанным законом Бенфорда. Указанная функция сначала вычисляет проверочный статистический показатель хи-квадрат, а затем сравнивает его с элементом таблицы распределения хи-квадрат при *p*-значении, равном 0,05 для 8 степеней свободы. Опираясь на это сравнение, указанная функция возвращает `True` либо `False`.

Листинг 16.4. Определяет функцию, которая измеряет соответствие наблюдаемых данных закону Бенфорда. Файл benford.py, часть 4

```
❶ def chi_square_test(data_count, expected_counts):
      """Вернуть булево значение на проверке хи-квадрат
        (8 степеней свободы и p-значения, равное 0.05)."""
    ❷ chi_square_stat = 0   # проверочный стат. показатель хи-квадрат
    ❸ for data, expected in zip(data_count, expected_counts):
        ❹ chi_square = math.pow(data - expected, 2)
          chi_square_stat += chi_square / expected
    ❺ print("\nПроверочный статистический показатель хи-квадрат = {:.3f}"
            .format(chi_square_stat))
      print("Критическое значение при p-значении 0.05 составляет 15.51.")

    ❻ return chi_square_stat < 15.51
```

Проверка по статистическому показателю хи-квадрат работает на количествах, поэтому указанная функция нуждается в списках количественных данных и ожидаемых количествах, которые возвращаются функциями `count_first_digits()` и `get_expected_counts()` ❶. Объявите переменную с именем `chi_square_stat` для хранения тестового статистического показателя хи-квадрат и назначьте ей значение `0` ❷.

Используйте встроенную функцию `zip` для перебора девяти значений в `data_count` и `expected_counts`; указанная функция будет попарно выбирать первый элемент в одном списке и первый элемент во втором списке и т. д. ❺. Для вычисления статистического показателя хи-квадрат сначала вычтите количества для каждой цифры и возведите результат в квадрат ❹. Затем разделите это значение на ожидаемое количество цифр и добавьте результат в переменную `chi_square_stat`. Затем напечатайте результат до трех знаков после точки ❺.

Верните булеву проверку переменной `chi_square_stat` относительно 15,51, т. е. критического значения, соответствующего *p*-значению 0,05 при 8 степенях свободы (см. табл. 16.1) ❻. Если `chi_square_stat` меньше этого значения, то данная функция вернет значение `True`; в противном случае она вернет `False`.

Определение функции построения столбчатого графика

В листинге 16.5 определяется первая часть функции, которая выводит на экран проценты наблюдаемых количеств в виде столбчатого графика модуля `matplotlib`. Аналогичный код использовался в *главе 12* для построения графика результатов симуляций пенсионных накоплений. Указанная функция также построит проценты закона Бенфорда в виде красных точек, благодаря чему можно будет визуально оценить то, насколько хорошо наблюдаемые данные соответствуют ожидаемому распределению.

Сайт модуля `matplotlib` содержит целый ряд примеров кода для построения самых разнообразных графиков. Приведенный ниже код частично основан на демонстрационном примере на веб-странице **https://matplotUb.org/examples/api/barchart_demo.html**.

Листинг 16.5. Определяет первую часть функции `bar_chart()`. Файл benford.py, часть 5

```
❶ def bar_chart(data_pct):
      """Изготовить столбчатый график наблюдаемой частоты
         первой цифры против ожидаемой (%)."""
   ❷ fig, ax = plt.subplots()

   ❸ index = [i + 1 for i in range(len(data_pct))] # 1-е цифры для оси x

      # текст для надписей, заголовка и делений
   ❹ fig.canvas.set_window_title('Процент первых цифр')
   ❺ ax.set_title('Данные против значений Бенфорда', fontsize=15)
   ❻ ax.set_ylabel('Частота, %', fontsize=16)
   ❼ ax.set_xticks(index)
      ax.set_xticklabels(index, fontsize=14)
```

Определите функцию `bar_chart()`, которая в качестве аргумента принимает список частот — в процентах — первых цифр в наблюдаемых данных ❶. Функция `plt.subplots()` возвращает кортеж объектов `figure` (рисунок) и `axes` (оси); распакуйте этот кортеж в переменные с именами `fig` и `ax` ❷.

Далее примените операцию включения в список для составления списка цифр от 1 до 9 ❸. Эта индексная переменная (`index`) будет определять местоположение на оси *x* каждого из девяти вертикальных столбиков графика.

Настройте заголовок графика, надписи и т. д. Назовите окно графика "Процент первых цифр" ❹, а затем покажите заголовок внутри графика ❺. Я использую общие названия, но вы можете настроить их так, чтобы они были более конкретными. Для установки размера текста равным 15 используйте именованный аргумент

fontsize. Обратите внимание, что заголовок окна является атрибутом объекта fig, но другие надписи будут атрибутами объекта ax.

Примените метод set_ylabel() для именования оси *y* как "Частота, %" ❻, затем задайте метки делений на оси *x*, основываясь на переменной index ❼. Метки делений будут иметь номера от 1 до 9, поэтому снова используйте переменную index и установите размер шрифта равным 14.

Завершение функции построения столбчатого графика

Листинг 16.6 завершает функцию bar_chart(), определяя столбики, аннотируя верхнюю часть каждого столбика своим значением частоты и нанося значения распределения Бенфорда в виде красных кружков.

Листинг 16.6. Завершает функцию генерирования столбчатого графика. Файл benford.py, часть 6

```
    # состроить столбики
❶   rects = ax.bar(index, data_pct, width=0.95, color='black',
                   label='Data')

    # прикрепить текстовую надпись поверх каждого столбика
    # с информацией о его высоте
❷   for rect in rects:
❸       height = rect.get_height()
❹       ax.text(rect.get_x() + rect.get_width()/2, height,
                '{:0.1f}'.format(height), ha='center', va='bottom',
                fontsize=13)

    # нанести значения Бенфорда как красные точки
❺   ax.scatter(index, BENFORD, s=150, c='red', zorder=2,
                label='Бенфорд')

    # спрятать верхний и нижний правые лучи и добавить легенду
❻   ax.spines['right'].set_visible(False)
    ax.spines['top'].set_visible(False)
❼   ax.legend(prop={'size':15}, frameon=False)

❽   plt.show()
```

Объявите переменную с именем rects для прямоугольников и используйте ее для хранения столбиков столбчатого графика ❶. Вы генерируете их с помощью метода bar(), который возвращает контейнер со всеми столбиками. Передайте ему индекс-

ную переменную и список частотностей в процентах, установите ширину каждого столбика равной `0.95`, заполните их черным цветом и установите аргумент `label` равным `'Data'`. Последний параметр представляет собой очень удобный способ автогенерирования легенды. Вы воспользуетесь им ближе к концу функции.

Я люблю наносить фактическое значение столбика чуть выше самого столбика, благодаря чему не нужно коситься на ось *y* и пытаться угадать его значение. Для этого начните с обхода каждого столбика (`rect`) в `rects` в цикле ❼ и получите его высоту ❽, т. е. его значение на оси *y*. Затем вызовите метод `text()` объекта `ax` ❶, передайте ему местоположение на оси *x* левой части столбика, полученное с помощью метода `get_x()`, и добавьте к нему половину ширины столбика с целью центрирования надписи над столбиком. Поскольку вы используете метод `get_width()`, вам нужно назначить ширину столбика только один раз, что вы и сделали в шаге ❶. Далее идет высота столбика, отформатированная до одного десятичного знака, вслед за которой идут горизонтальное и вертикальное выравнивания. Установите их соответственно по центру и низу текстовой ограничительной рамки. Завершите функцию, задав размер текста.

Теперь начните строить "маркеры" модуля `matplotlib` — в данном случае точки, которые будут отмечать местоположение частот распределения Бенфорда для каждой первой цифры. Сделайте это с помощью метода `scatter()`, который строит графики рассеяния ❾.

Первые два аргумента в `scatter()` являются местоположениями на осях *x* и *y* каждого маркера, представленного поочередными парами из списков `index` и `BENFORD`. Далее идет размер маркера, установленный равным 150, и затем цвет. Оба цвета — `red` и `DodgerBlue` — выглядят хорошо. Вы хотите разместить метки вверху столбиков, и поэтому установите аргумент порядка отображения `zorder` равным 2. Элементы на рисунке в `matplotlib` называются "художниками" (artists), причем художники с более высокими значениями `zorder` будут строить графики поверх тех, у кого он имеет более низкие значения. Закончите аргументом `label`, используемым для создания легенды.

Следующие две инструкции предназначены для эстетики. Модуль `matplotlib` по умолчанию рисует прямоугольник вокруг внутренней части графика, и верхняя граница может помешать надписям, размещенным вверху каждого столбика. Поэтому удалите верхнюю и правую границы, установив их видимость равной `False` ❻.

Примените `legend()` для построения легенды на графике ❿. Указанный метод будет работать без аргументов, но задайте свойству `size` значение 15 и отключите рамку вокруг легенды с целью получения, возможно, более привлекательного результата. Завершите функцию, вызвав метод `plt.show()` для вывода графика на экран ❽. Пример столбчатого графика показан на рис. 16.7.

Функция `main()` будет выводить дополнительную информацию в окне интерпретатора как текст. Указанная информация будет включать значение проверочного статистического показателя хи-квадрат.

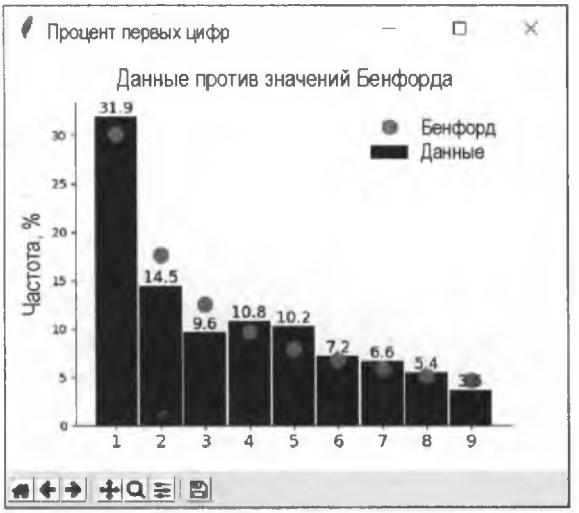


Рис. 16.7. Пример результата функции `bar_chart()`

Определение и выполнение функции *main()*

Листинг 16.7 определяет функцию `main()` и выполняет программу в виде модуля либо в автономном режиме. Поскольку подавляющая часть работы выполняется в отдельных функциях, главная функция `main()` "главным образом" вызывает эти функции и печатает некоторую статистику.

Листинг 16.7. Определяет функцию `main()` и выполняет программу в виде модуля либо в автономном режиме. Файл benford.py, часть 7

```
def main():
    """Вызвать функции и напечатать статистику."""
    # загрузить данные
    while True:
    ❶ filename = input("\nИмя файла с КОЛИЧЕСТВЕННЫМИ данными: ")
        try:
            data_list = load_data(filename)
        except IOError as e:
            print("{}. Попробовать еще раз.".format(e), file=sys.stderr)
        else:
            break
    ❷ data_count, data_pct, total_count = count_first_digits(data_list)
    ❸ expected_counts = get_expected_counts(total_count)
      print("\nнаблюдаемые количества = {}".format(data_count))
      print("ожидаемые количества   = {}".format(expected_counts), "\n")

    ❹ print("Вероятности первой цифры:")
    ❺ for i in range(1, 10):
```

```
        print("{}: наблюдается: {:.3f} ожидается: {:.3f}".
              format(i, data_pct[i - 1] / 100, BENFORD[i - 1] / 100))

    ❻ if chi_square_test(data_count, expected_counts):
          print("Наблюдаемое распределение совпадает с ожидаемым \
                распределением.")
      else:
          print("Наблюдаемое распределение не совпадает с ожидаемым.",
                file=sys.stderr)

    ❼ bar_chart(data_pct)

❽ if __name__ == '__main__':
      main()
```

Начните с того, что попросите пользователя ввести имя файла с количественными данными, подлежащими анализу ❶; вставьте этот запрос в цикл `while`, который будет продолжаться до тех пор, пока пользователь не введет допустимое имя файла или не закроет окно. Пользователь может ввести имя файла либо полный путь, в случае если он хочет загрузить набор данных, который хранится не в текущем рабочем каталоге. Например, в Windows:

```
Имя файла с КОЛИЧЕСТВЕННЫМИ данными: C:\Python35\Benford\Illinois_votes.txt
```

Используйте инструкцию `try` для вызова созданной ранее функции `load_data()` и передайте ей имя файла. Если имя файла является допустимым, то возвращенный список назначается переменной `data_list`. Если возникает исключение, то перехватите его и напечатайте ошибку. В противном случае прервите цикл `while`.

Затем передайте возвращенный список количеств данных в функцию `count_first_digits()` и распакуйте результаты в виде переменных `data_count`, `data_pct` и `total_count`, т. е. списки соответственно количеств первой цифры, процентов и общего числа количеств ❸. Затем сгенерируйте список количеств, ожидаемых в распределении по закону Бенфорда, вызвав функцию `get_expected_counts()` и передав ей переменную `total_count` ❸. Напечатайте списки наблюдаемых и ожидаемых количеств.

Теперь составьте таблицу, которая сравнивает частоту первой цифры в данных с ожидаемыми значениями. Используйте вероятности, т. к. в интерактивной оболочке десятичные значения легко выравниваются в привлекательном виде. Начните с инструкции печати заголовка ❹, затем переберите числа от 1 до 9, печатая результаты для наблюдаемых количеств (данных), а затем ожидаемые количества, причем каждый из них до трех знаков после точки ❸. Обратите внимание, что индексы в двух списках начинаются с нуля, поэтому из `i` следует вычесть 1.

Передайте два списка количеств в функцию `chi_square_test()`, которая вычислит степень соответствия наблюдаемых данных ожидаемому распределению ❻. Если указанная функция возвращает `True`, то примените инструкцию `print`, для того что-

бы сообщить пользователю, что наблюдаемое распределение соответствует закону Бенфорда (или, более технически, между ними нет значимой разницы). В противном случае сообщите, что они не совпадают и для пользователей интерактивной оболочки раскрасьте шрифт красным цветом.

Функция `chi_square_test()` выведет свои результаты в окне интерпретатора, поэтому для создания столбчатого графика вызовите функцию `bar_chart()` ❼. Передайте ей список количеств данных в процентах.

Вернувшись в глобальное пространство, завершите программу фрагментом кода для ее запуска в виде модуля либо в автономном режиме ❽.

Если эту программу выполнить на наборе данных Illinois_votes.txt, то вы увидите результат, показанный на рис. 16.8. Если основываться на законе Бенфорда, то в результатах голосования нет никакой явной аномалии.

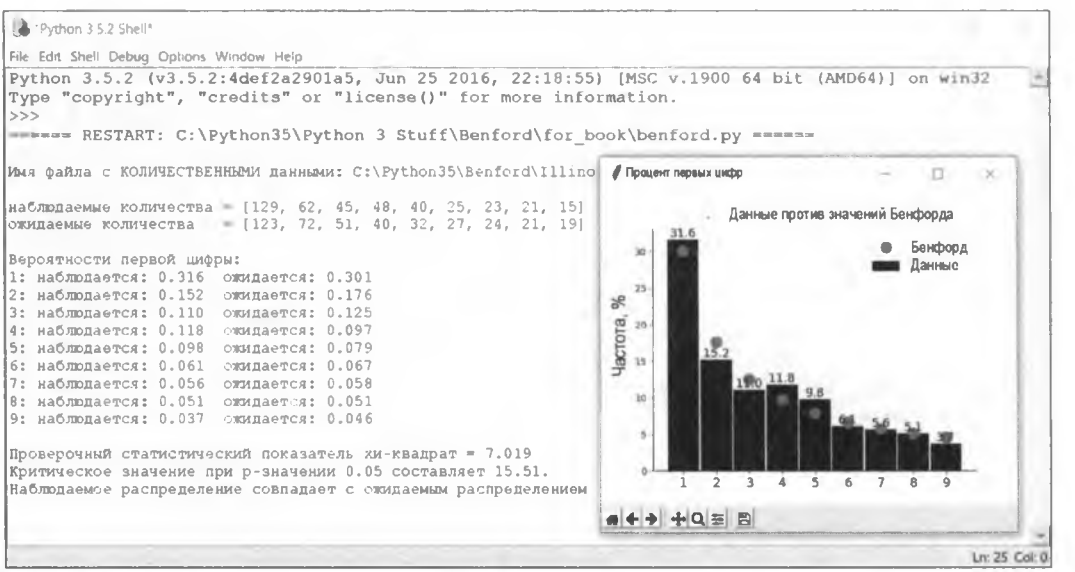


Рис. 16.8. Результат работы программы benford.py для набора данных Illinois_votes.txt

Если же выполнить эту программу, используя только голоса Трампа, а затем только голоса Клинтон, то вы получите результаты, показанные на рис. 16.9. Распределение голосов за Трампа с проверочным статистическим показателем 15,129 едва проходит проверку по статистическому показателю хи-квадрат.

В таких случаях вы должны быть осторожными, чтобы не делать опрометчивых выводов. Набор данных невелик — всего 102 образца на кандидата — и на результаты могут влиять такие факторы, как демографические различия и различия в явке избирателей в сельских и городских районах. Интересную статью о разделении между городом и сельской местностью можно найти по адресу

http://www.chicagotribune.com/news/data/ct-illinois-election-urban-rural-divide-2016-htmlstory.html.

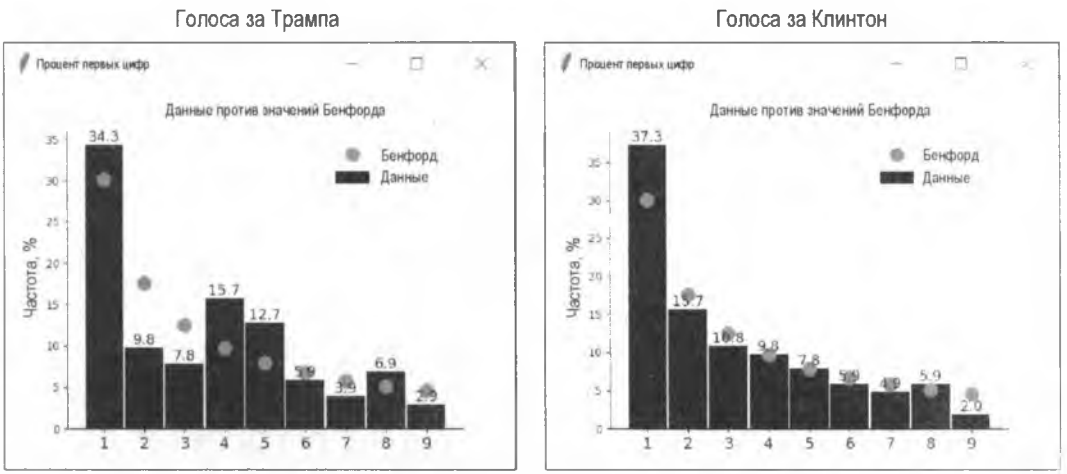


Рис. 16.9. Сравнение результатов Трампа (слева) и результатов Клинтон (справа) для штата Иллинойс, США

В практическом проекте "Победа над Бенфордом" далее в этой главе вы получите возможность вмешаться в подсчет голосов в штате Иллинойс и изменить результат. Затем вы воспользуетесь приведенным выше программным кодом для того, чтобы увидеть степень соответствия результатов закону Бенфорда.

Резюме

Еще в *главе 1* мы использовали практические проекты "Столбчатый график бедняка" и "Столбчатый график бедного чужака" с целью посмотреть на то, как частота встречаемости букв в языке является нерегулярной и предсказуемой. Этот результат обеспечивает мощный инструмент для криптоанализа. Здесь, в конце книги, мы прошли полный круг и обнаружили, что даже числа имеют эту общую черту, что приводит к мощному инструменту для обнаружения мошенничества. С помощью короткой и простой программы на языке Python вы можете потрясти сами основания небесных сводов и навлечь на землю великое и могучее громыхание — и все потому, что кто-то заметил, что первые страницы книги были грязными.

На этом, пожалуй, с "Непрактичными проектами на языке Python" мы и закончим. Надеюсь, вам было весело, вы узнали что-то новое и вдохновились на создание собственного набора непрактичных проектов!

Дальнейшее чтение

Книга Марка Нигрини "Закон Бенфорда: приложения для судебного учета, аудита и обнаружения мошенничества" (Nigrini M. Benford's Law: applications for forensic accounting, auditing, and fraud detection. John Wiley & Sons, 2012) охватывает математику, теорию и тестирование закона Бенфорда вместе с примерами приложений, включая мошенничество, уклонение от уплаты налогов и схемы Понци.

Практический проект: победа над Бенфордом

Проверьте свои навыки в фальсификации выборов, используя приведенный ниже практический проект. Решение задачи, beat_benford_practice.py, можно найти в приложении к книге либо скачать его с веб-сайта **https://www.nostarch.com/impracticalpython/**.

Набор данных не следует считать допустимым только потому, что он следует закону Бенфорда. Причина проста: если вы знаете о законе Бенфорда, то можете его обойти.

В качестве подтверждения данного положения представьте, что вы — хакер высокого уровня, работающий на иностранное правительство и имеющий доступ ко всем результатам голосования в штате Иллинойс. Напишите программу на языке Python, которая вмешивается в голоса по всему округу с целью победы Дональда Трампа в этом штате, но количество голосов по-прежнему подчиняется закону Бенфорда. Будьте осторожны: Иллинойс — это "синий" штат (т. е. штат демократов), поэтому вы не хотите создавать сенсационную победу (грубо определяемую как преимущество в 10–15% в народном голосовании). Во избежание подозрений Трамп должен проскользнуть с несколькими процентными пунктами.

ПРИМЕЧАНИЕ

В штатах действуют правила, касающиеся пересчета голосов. Прежде чем манипулировать выборами, мошенник хотел бы об этом знать во избежание проверки, которую принесет пересчет. Фактические уставные правила по каждому штату читать неинтересно, но инициатива "Граждане за честные выборы в Миннесоте" предоставляет доступное резюме. Одно такое для штата Иллинойс можно найти по адресу **https://ceimn.org/searchable-databases/recount-database/illinois/**.

Ваша программа должна украсть голоса у других кандидатов, при этом сохраняя итоговые данные по округам нетронутыми; благодаря этому суммарное число поданных голосов не изменится. В качестве контроля качества напечатайте старые и новые итоги голосования по округам для Трампа и Клинтон, а также их старые и новые итоги по всему штату. Затем запишите текстовый файл, который можно ввести в программу benford.py для проверки того, как у вас получилось с законом Бенфорда.

Наборы данных для каждого кандидата уже подготовлены и перечислены ниже; их можно скачать с веб-сайта **https://www.nostarch.com/impracticalpython/**. Каждый из этих наборов данных является просто столбцом чисел, представляющих голоса, отсортированные в алфавитном порядке по округам (поэтому порядок не меняйте!).

- Clinton_votes_Illinois.txt.
- Johnson_votes_Illinois.txt.
- Stein_votes_Illinois.txt.
- Trump_votes_Illinois.txt.

На рис. 16.10 показаны результаты выполнения программы benford.py на результатах моей попытки, beat_benford_practice.py, где использовались приведенные выше

наборы данных. Распределение проходит проверку по показателю хи-квадрат и дает визуально убедительное — но правдоподобно несовершенное — соответствие значениям, предсказанным законом Бенфорда.

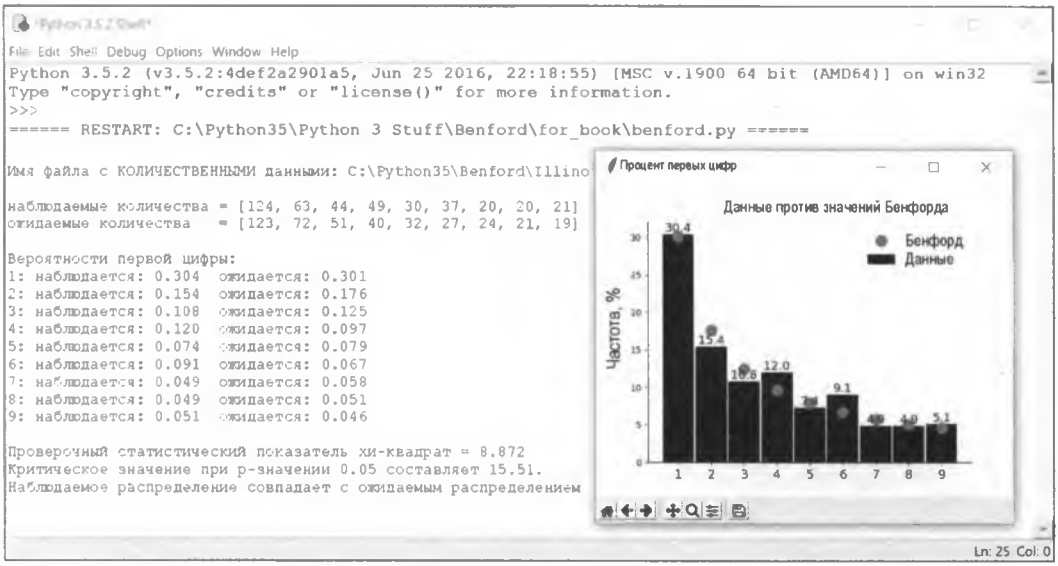


Рис. 16.10. Результаты выполнения распределения из beat_benford_practice.py в программе benford.py... Озорство удалось!

Ниже показано несколько строк вывода из beat_benford_practice.py, со старыми и новыми итогами голосования по округам:

```
old Trump: 12412     new Trump: 13223     old Clinton: 3945       new Clinton: 3290
-------------------------------------------------------------------------------------
old Trump: 13003     new Trump: 14435     old Clinton: 7309       new Clinton: 6096
-------------------------------------------------------------------------------------
old Trump: 453287    new Trump: 735863    old Clinton: 1611946    new Clinton: 1344496
-------------------------------------------------------------------------------------
old Trump: 6277      new Trump: 6674      old Clinton: 1992       new Clinton: 1661
-------------------------------------------------------------------------------------
old Trump: 4206      new Trump: 4426      old Clinton: 1031       new Clinton: 859
-------------------------------------------------------------------------------------
```

Третья строка сверху представляет округ Кук (Cook County), в котором находится Чикаго. Отметим, что Клинтон по-прежнему здесь выигрывает, но с меньшим отрывом. Победа Трампа в этом "синем" округе откровенно была бы огромным красным флагом, сигнализирующим о том, что произошло вмешательство в голосование, даже если бы он победил во всем штате лишь с небольшим отрывом!

Сложные проекты

Попробуйте свои силы в следующих ниже сложных проектах. Никаких решений не предусмотрено.

Бенфордирование колеблющихся штатов

Ни один кандидат не будет жульничать в штате, в котором он гарантированно побеждает. Если вы — следователь, проводящий расследование на предмет мошенничества с голосами избирателей, то вы, скорее всего, начнете с колеблющихся штатов. Они могут качнуться в любом направлении, и кандидаты тратят подавляющую часть своего боевого арсенала — и времени — на эти места. По данным некоммерческой и беспартийной политической онлайновой энциклопедия Ballotpedia (**https://ballotpedia.org**), колеблющимися штатами Трампа в 2016 г. были Аризона, Айова, Мичиган, Висконсин, Огайо, Пенсильвания, Северная Каролина и Флорида. У Клинтон это были Колорадо, Невада, Нью-Гэмпшир и Вирджиния.

Отчеты о заочном (онлайновом) голосовании в штатах обычно предоставляются в нескольких форматах, таких как электронные таблицы Microsoft Excel. Соберите их для колеблющихся штатов, конвертируйте их в текстовые файлы и пропустите их через программу benford.py. Для старта вы можете найти отчеты по штату Огайо вот здесь: **https://www.sos.state.oh.us/elections/**.

Пока никто не заметил

Бывший спикер Палаты представителей США Тип О'Нил (Tip O'Neill) прославился своим изречением "Вся политика является локальной". Примите это близко к сердцу и воспользуйтесь программой benford.py, для того чтобы проверить некоторые ваши местные предвыборные гонки, такие как выборы судей, мэров, руководителей округа, шерифов и членов городского совета. Эти события обычно привлекают меньше внимания, чем гонки за места в сенате, губернатора или президента. Если вы обнаружите нарушение, то, прежде чем устраивать скандал, обеспечьте ситуацию, в которой набор данных голосования был допустимым применением закона Бенфорда!

ПРИЛОЖЕНИЕ

РЕШЕНИЯ ПРАКТИЧЕСКИХ ПРОЕКТОВ

В этом приложении представлены решения практических проектов каждой главы. Цифровые версии доступны на веб-сайте книги по адресу **https://www.nostarch.com/impracticalpython/**.

Глава 1. Генератор дурацких имен

"Поросячья" латынь

pig_Latin_practice.py

```
"""Превратить слово в его эквивалент на "поросячьей" латыни."""
import sys

VOWELS = 'aeiouy'

while True:
    word = input("Наберите слово и получите его перевод на поросячью латынь: ")

    if word[0] in VOWELS:
        pig_Latin = word + 'way'
    else:
        pig_Latin = word[1:] + word[0] + 'ay'
    print()
    print("{}".format(pig_Latin), file=sys.stderr)

    try_again = input("\n\nПопробуете еще раз? (Нажмите Enter либо n
                                                    для остановки)\n ")
    if try_again.lower() == "n":
        sys.exit()
```

Столбчатый график бедняка

EATOIN_practice.py

```
"""Увязать буквы символьной цепочки со словарем и напечатать
   столбчатый график частоты."""
import sys
import pprint
from collections import defaultdict

# Примечание: для того чтобы столбцы уместились в окне интерпретатора IDLE,
# текст должен представлять собой короткую фразу
text_ru = 'Как замок в своем углу из средневековой игры, я предвижу \
ужасную беду, и я все равно здесь остаюсь.'
text = 'Like the castle in its corner in a medieval game,  \
I foresee terribletrouble and I stay here just the same.'

ALPHABET = 'abcdefghijklmnopqrstuvwxyz'

# модуль defaultdict позволяет строить ключи словаря на лету!
mapped = defaultdict(list)
for character in text:
    character = character.lower()
        if character in ALPHABET:
            mapped[character].append(character)

# модуль pprint позволяет печатать результат стопкой
print("\nЕсли текст переносится на новую строку, то вам нужно растянуть
                                                    консольное окно.\n")
print("text = ", end='')
print("{}\n".format(text), file=sys.stderr)
pprint.pprint(mapped, width=110)
```

Глава 2. Поиск палиндраммных заклинаний

Очистка словаря

dictionary_cleanup_practice.py

```
"""Удалить однобуквенные слова из списка, если не 'a' или 'i'."""
word_list = ['a', 'nurses', 'i', 'stack', 'b', 'cats', 'c']

permissible = ('a', 'i')

# удалить однобуквенные слова, если не "a" или "i"
for word in word_list:
    if len(word) == 1 and word not in permissible:
        word_list.remove(word)

print("{}".format(word_list))
```

Глава 3. Решение анаграмм

Поиск диграмм

count_digrams_practice.py

```
"""Сгенерировать буквенные пары в словарь Voldemort и отыскать
   их частоту в словаре. Требует модуля load_dictionary.py
   для загрузки файла словаря английского языка.
"""
import re
from collections import defaultdict
from itertools import permutations
import load_dictionary

word_list = load_dictionary.load('2of4brif.txt')

name = 'Voldemort'   #(tmvoordle)
name = name.lower()

# сгенерировать уникальные словарные пары из имени
digrams = set()
perms = {''.join(i) for i in permutations(name)}
for perm in perms:
    for i in range(0, len(perm) - 1):
        digrams.add(perm[i] + perm[i + 1])
print(*digrams, sep='\n')
print("\nЧисло диграм = {}\n".format(len(digrams)))

# применить регулярные выражения для отыскания повторяющихся
# диграмм в слове
mapped = defaultdict(int)
for word in word_list:
    word = word.lower()
    for digram in digrams:
        for m in re.finditer(digram, word):
          mapped[digram] += 1

print("частотность диграмм:")
count = 0
for k in mapped:
    print("{} {}".format(k, mapped[k]))
```

Глава 4. Декодирование шифров времен Гражданской войны в США

Взламывание сообщения Линкольна

Кодовое слово	Открытый текст
WAYLAND	captured
NEPTUNE	Richmond

Открытый текст: correspondents of the Tribune captured at Richmond please ascertain why they are detained and get them off if you can this fills it up

Перевод: корреспонденты Tribune захвачены в Ричмонде пожалуйста выясните причину их задержания и спасите их если можете терпеть нету мочи

Идентификация типов шифров

identify_cipher_type_practice.py

```
"""Загрузить шифротекст и использовать присутствующую долю ETAOIN
для классификации типа шифра."""
import sys
from collections import Counter

# установить произвольную долю отсечения из 6 наиболее распространенных
# букв в английском шифротексте с целевой долей
# или больше = перестановочный шифр
CUTOFF = 0.5

# загрузить шифротекст
def load(filename):
    """Открыть текстовый файл и вернуть список."""
    with open(filename) as f:
        return f.read().strip()

try:
    ciphertext = load('cipher_a.txt')
except IOError as e:
    print("{}. Завершение программы.".format(e),
          file=sys.stderr)
    sys.exit(1)

# подсчитать 6 наиболее распространенных букв в шифротексте
six_most_frequent = Counter(ciphertext.lower()).most_common(6)
print("\nШесть наиболее часто используемых букв в английском языке = ETAOIN")
```

```
print('\ nШесть наиболее часто используемых букв в шифротексте =')
print(*six_most_frequent, sep='\n')

# конвертировать список кортежей в множество букв для сравнения
cipher_top_6 = {i[0] for i in six_most_frequent}

TARGET = 'etaoin'
count = 0
for letter in TARGET:
    if letter in cipher_top_6:
        count += 1

if count/len(TARGET) >= CUTOFF:
    print("\nЭтот шифротекст, скорее всего, был порожден ПЕРЕСТАНОВОЧНЫМ шифром")
else:
    print("Этот шифротекст, скорее всего, был порожден ПОДСТАНОВОЧНЫМ шифром")
```

Хранение ключа в виде словаря

key_dictionary_practice.py

```
"""Ввести символьную цепочку с ключом шифра, взять введенное пользователем
   значение с направлением маршрута как значение словаря."""
col_order = """1 3 4 2"""
key = dict()
cols = [int(i) for i in col_order.split()]
for col in cols:
    while True:
        key[col] = input("Направление чтения столбца {} (u = вверх, d = вниз):
                                                    ".format(col).lower())
        if key[col] == 'u' or key[col] == 'd':
            break
        else:
            print("Входным значением должно быть 'u' либо 'd'")

    print("{}, {}".format(col, key[col]))
```

Автоматическое генерирование возможных ключей

permutations_practice.py

```
"""Для суммарного числа столбцов отыскать все уникальные расстановки
   столбцов. Строит список списков, содержащий все возможные уникальные
   расстановки индивидуальных номеров столбцов, включая отрицательные
   значения для направления маршрута (чтение в столбце вверх или вниз).
Ввод:
- суммарное число столбцов
```

```
Возвращает:
- список списков уникальных порядков столбцов, включая отрицательные значения
для направления шифрования маршрутным шифром
"""
import math
from itertools import permutations, product

#------НАЧАЛО ВХОДНЫХ ДАННЫХ ---------------------------------------
# Input total number of columns:
num_cols = 4

#------НЕ РЕДАКТИРОВАТЬ НИЖЕ ЭТОЙ СТРОКИ----------------------------

# сгенерировать список индивидуальных номеров столбцов
columns = [x for x in range(1, num_cols+1)]
print("columns = {}".format(columns))

# построить список списков комбинаций номеров столбцов
# функция product модуля itertools вычисляет декартово произведение
# входных итерируемых типов
def perms(columns):
    """Взять целое число столбцов и сгенерировать
    положительные и отрицательные перестановки."""
    results = []
    for perm in permutations(columns):
        for signs in product([-1, 1], repeat=len(columns)):
            results.append([i*sign for i, sign in zip(perm, signs)])
    return results

col_combos = perms(columns)
print(*col_combos, sep="\n")    # закомментируйте для num_cols > 4!
print("Факториал числа num_cols без отрицательных значений = {}"
      .format(math.factorial(num_cols)))
print("Число комбинаций столбцов = {}".format(len(col_combos)))
```

Маршрутный перестановочный шифр: атака с применением грубой силы

В данном практическом проекте используются две программы. Вторая, perms.py, применяется в качестве модуля в первой программе, route_cipher_hacker.py. Она была собрана из программы permutations_practice.py, описанной в *разд. "Автоматическое генерирование возможных ключей" ранее в этом приложении.*

route_cipher_hacker.py

```
"""Взламывание маршрутного шифра Союзников грубой силой
(route_cipher_hacker.py).

Предназначен для полнословных перестановочных шифров с переменными строками
и столбцами.
```

```
Исходит из того, что шифрование началось вверху или внизу столбца.
Возможные ключи автоматически генерируются на основе числа вводимых столбцов
и строк.
Ключ указывает на порядок чтения столбцов и направление прохождения.
Отрицательные номера столбцов означают начало снизу и чтение вверх.
Положительные номера столбцов означают начало сверху и чтение вниз.

Приведенный ниже пример предназначен для матрицы 4 x 4 с ключом -1 2 -3 4.
Примечание: "0" не допускается.
Стрелки показывают маршрут шифрования; для отрицательных значений ключа читать
ВВЕРХ.

  1   2   3   4
 ___ ___ ___ ___
| ^ | | | ^ | | | СООБЩЕНИЕ НАПИСАНО
|_|_|_V_|_|_|_V_|
| ^ | | | ^ | | | ПОПЕРЕК КАЖДОЙ СТРОКИ
|_|_|_V_|_|_|_V_|
| ^ | | | ^ | | | ТАКИМ ОБРАЗОМ
|_|_|_V_|_|_|_V_|
| ^ | | | ^ | | | ПОСЛЕДНЯЯ СТРОКА ЗАПОЛНЯЕТСЯ ФИКТИВНЫМИ СЛОВАМИ
|_|_|_V_|_|_|_V_|
НАЧАЛО       КОНЕЦ

Необходимые входные данные - текстовое сообщение, число столбцов,
                             число строк, символьная цепочка с ключом
Для генерирования ключей требует специальный модуль "perms"
Печатает использованный ключ и переведенный открытый текст
"""

import sys
import perms
#=====================================================================
# ВХОДНЫЕ ДАННЫЕ ПОЛЬЗОВАТЕЛЯ:

# символьная цепочка, подлежащая расшифровке
# (наберите или вставьте между тройными кавычками):
ciphertext = """REST TRANSPORT YOU GODWIN VILLAGE ROANOKE WITH ARE YOUR IS
JUST SUPPLIES FREE SNOW HEADING TO GONE TO SOUTH FILLER
"""

# число столбцов, которые, как ожидается, находятся
# в перестановочной матрице:
COLS = 4

# число строк, которые, как ожидается, находятся в перестановочной матрице:
ROWS = 5
# КОНЕЦ ВХОДНЫХ ДАННЫХ ПОЛЬЗОВАТЕЛЯ - НЕ РЕДАКТИРОВАТЬ НИЖЕ ЭТОЙ СТРОКИ!
```

```
#=====================================================================
def main():
    """Превратить шифротекст в список, вызвать функции валидации и
    расшифровки."""
    cipherlist = list(ciphertext.split())
    validate_col_row(cipherlist)
    decrypt(cipherlist)

def validate_col_row(cipherlist):
    """Проверить, что входные столбцы и строки являются допустимыми
    относительно длины сообщения."""
    factors = []
    len_cipher = len(cipherlist)
    for i in range(2, len_cipher):  # диапазон исключает 1-столбцовые шифры
        if len_cipher % i == 0:
            factors.append(i)
    print("\nДлина шифра = {}".format(len_cipher))
    print("Приемлемые значения столбца/строки включают: {}"
          .format(factors))
    print()
    if ROWS * COLS != len_cipher:
        print("\nОшибка - Входные столбцы и строки не являются кратными \
              длине шифра. Завершение программы.", file=sys.stderr)
        sys.exit(1)

def decrypt(cipherlist):
    """Превратить столбцы в элементы списка списков и расшифровать
    шифротекст."""
    col_combos = perms.perms(COLS)
    for key in col_combos:
        translation_matrix = [None] * COLS
        plaintext = ''
        start = 0
        stop = ROWS
        for k in key:
            if k < 0:    # чтение снизу вверх столбца
                col_items = cipherlist[start:stop]
            elif k > 0:  # чтение сверху вниз столбца
                col_items = list((reversed(cipherlist[start:stop])))
            translation_matrix[abs(k) - 1] = col_items
            start += ROWS
            stop += ROWS
        # перебрать вложенные списки в цикле, выталкивая
        # последний элемент в новый список:
        for i in range(ROWS):
            for matrix_col in translation_matrix:
                word = str(matrix_col.pop())
                plaintext += word + ' '
```

```
        print("\nиспользуя ключ = {}".format(key))
        print("переведено = {}".format(plaintext))
    print("\nчисло ключей = {}".format(len(col_combos)))

if __name__ == '__main__':
    main()
```

Модуль perms.py

```
"""Для суммарного числа столбцов отыскать все уникальные расстановки столбцов.

Строит список списков, содержащий все возможные уникальные расстановки
индивидуальных номеров столбцов, включая отрицательные значения для
направления маршрута

Вход:
-суммарное число столбцов

Возвращает:
-список списков уникальных порядков столбцов, включая отрицательные
 значения для направления шифрования маршрутным шифром

"""
from itertools import permutations, product

# строит список списков комбинаций номеров столбцов
# функция product модуля itertools вычисляет декартово произведение
# входных итерируемых типов
def perms(num_cols):
    """Взять целое число столбцов и сгенерировать
    положительные и отрицательные перестановки."""
    results = []
    columns = [x for x in range(1, num_cols+1)]
    for perm in permutations(columns):
        for signs in product([-1, 1], repeat=len(columns)):
            results.append([i*sign for i, sign in zip(perm, signs)])
    return results
```

Глава 5. Кодирование шифров времен Гражданской войны в Англии

Спасение королевы Марии

save_Mary_practice.py

```
"""Спрятать нулевой шифр внутри списка имен, используя переменный шаблон."""
import load_dictionary
```

```
# написать короткое сообщение и не использовать никаких знаков
# препинания или чисел!
message = "Give your word and we rise"
message = "".join(message.split())

# открыть файл имен
names = load_dictionary.load('supporters.txt')

name_list = []

# начать список с нулевого слова, не используемого в шифре
name_list.append(names[0])

# добавить букву нулевого шифра ко 2-й букве имени, затем к 3-й,
# затем повторить
count = 1
for letter in message:
    for name in names:
        if len(name) > 2 and name not in name_list:
            if count % 2 == 0 and name[2].lower() == letter.lower():
                name_list.append(name)
                count += 1
                break
            elif count % 2 != 0 and name[1].lower() == letter.lower():
                name_list.append(name)
                count += 1
                break

# добавить два нулевых слова в начале сообщения,
# для того чтобы сбить криптоаналитиков
name_list.insert(3, 'Stuart')
name_list.insert(6, 'Jacob')

# показать фиктивное письмо и список с нулевым шифром
print("""
Your Royal Highness: \n
It is with the greatest pleasure I present the list of noble families who
have undertaken to support your cause and petition the usurper for the
release of your Majesty from the current tragical circumstances.
""")

"""ПЕРЕВОД:
Ваше Королевское Высочество:
С величайшим удовольствием представляю список благородных семей, которые
обязались поддержать ваше дело и ходатайствовать перед узурпатором
об освобождении Вашего Величества от нынешних трагических обстоятельств.
"""

print(*name_list, sep='\n')
```

Колчестерская поимка

colchester_practice.py

```
"""Решить нулевой шифр, основываясь на каждой n-й букве
   в каждом n-м слове."""
import sys

def load_text(file):
    """Загрузить текстовый файл в виде символьной цепочки."""
    with open(file) as f:
        return f.read().strip()

# загрузить и обработать сообщение:
filename = input("\nВведите полное имя файла переводимого сообщения: ")
try:
    loaded_message = load_text(filename)
except IOError as e:
    print("{}. Завершение программы.".format(e), file=sys.stderr)
    sys.exit(1)

# проверить загруженное сообщение и его число строк
print("\nИСХОДНОЕ СООБЩЕНИЕ = {}\n".format(loaded_message))

# конвертировать сообщение в список и получить его длину
message = loaded_message.split()
end = len(message)

# получить от пользователя входное значение о проверяемом интервале
increment = int(input("Введите макс. слово и буквенную позицию для \
                      проверки (напр., каждые 1 из 1, 2 из 2 и т.д.): "))
print()

# отыскать буквы через определенные интервалы
for i in range(1, increment + 1):
    print("\nИспользуя инкремент, буква {} слова {}".format(i, i))
    print()
    count = i - 1
    location = i - 1
    for index, word in enumerate(message):
        if index == count:
            if location < len(word):
                print("буква = {}".format(word[location]))
                count += i
            else:
                print("Интервал не работает", file=sys.stderr)
```

Глава 6. Написание текста невидимыми чернилами

Проверка числа пустых строк

elementary_ink_practice.py

```
"""Добавить код проверки на пустые строки в фиктивном сообщении
относительно строк в реальном сообщении."""
import sys
import docx
from docx.shared import RGBColor, Pt

# получить текст из фиктивного сообщения и сделать каждую строку
# элементом списка
fake_text = docx.Document('fakeMessage.docx')
fake_list = []
for paragraph in fake_text.paragraphs:
    fake_list.append(paragraph.text)

# получить текст из реального сообщения и сделать каждую строку
# элементом списка
real_text = docx.Document('realMessageChallenge.docx')
real_list = []
for paragraph in real_text.paragraphs:
    if len(paragraph.text) != 0:   # удалить пустые строки
        real_list.append(paragraph.text)

# определить функцию проверки имеющегося потайного пространства:
def line_limit(fake, real):
    """Сравнить число пустых строк в фиктивном и реальном сообщениях
    и предупредить пользователя, если будет недостаточно строк,
    для того чтобы уместить реальное сообщение.

    ПРИМЕЧАНИЕ: потребуется импортировать 'sys'

    """
    num_blanks = 0
    num_real = 0
    for line in fake:
        if line == '':
            num_blanks += 1
    num_real = len(real)
    diff = num_real - num_blanks
    print("\nЧисло пустых строк в фиктивном сообщении = {}"
          .format(num_blanks))
```

```
    print("Число строк в реальном сообщении = {}\n".format(num_real))
    if num_real > num_blanks:
        print("Фиктивному сообщению требуется еще {} пустых строк."
              .format(diff), file=sys.stderr)
        sys.exit()

line_limit(fake_list, real_list)

# загрузить шаблон, который задает стиль, шрифт, края и т. д.
doc = docx.Document('template.docx')

# добавить фирменный бланк
doc.add_heading('Морланд Холмс', 0)
subtitle = doc.add_heading('Глобальный консалтинг и ведение переговоров', 1)
subtitle.alignment = 1
doc.add_heading('', 1)
doc.add_paragraph('17 декабря 2019')
doc.add_paragraph('')

def set_spacing(paragraph):
    """Использовать docx для задания интервала между абзацами."""
    paragraph_format = paragraph.paragraph_format
    paragraph_format.space_before = Pt(0)
    paragraph_format.space_after = Pt(0)

length_real = len(real_list)
count_real = 0   # индекс текущей строки в реальном (скрытом) сообщении

# чередовать строки реального и фиктивного сообщений
for line in fake_list:
    if count_real < length_real and line == "":
        paragraph = doc.add_paragraph(real_list[count_real])
        paragraph_index = len(doc.paragraphs) - 1

        # сделать цвет реального сообщения белым
        run = doc.paragraphs[paragraph_index].runs[0]
        font = run.font
        font.color.rgb = RGBColor(255, 255, 255)  # для проверки сделать
                                                  # его красным
        count_real += 1

    else:
        paragraph = doc.add_paragraph(line)

    set_spacing(paragraph)

doc.save('ciphertext_message_letterhead.docx')

print("Готово."))
```

Глава 8. Подсчет слогов в стихотворениях хокку

Счетчик слогов против файла словаря

test_count_syllables_w_dict.py

```
"""Загрузить файл словаря, подобрать случайные слова, выполнить модуль
подсчета слогов."""
import sys
import random
from count_syllables import count_syllables

def load(file):
    """Открыть текстовый файл и вернуть список
    символьных цепочек в нижнем регистре."""
    with open(file) as in_file:
        loaded_txt = in_file.read().strip().split('\n')
        loaded_txt = [x.lower() for x in loaded_txt]
        return loaded_txt

try:
    word_list = load('2of4brif.txt')
except IOError as e:
    print("{}\nОшибка при открытии файла. Завершение программы."
          .format(e), file=sys.stderr)
    sys.exit(1)

test_data = []
num_words = 100
test_data.extend(random.sample(word_list, num_words))

for word in test_data:
    try:
        num_syllables = count_syllables(word)
        print(word, num_syllables, end='\n')
    except KeyError:
        print(word, end='')
        print(" не найдено", file=sys.stderr)
```

Глава 10. Мы одни? Разведывание парадокса Ферми

Далекая-предалекая галактика

galaxy_practice.py

```
"""Применить спиральную формулу для построения изображения галактики."""
import math
from random import randint
import tkinter

root = tkinter.Tk()
root.title("Galaxy BR549")
c = tkinter.Canvas(root, width=1000, height=800, bg='black')
c.grid()
c.configure(scrollregion=(-500, -400, 500, 400))
oval_size = 0

# построить спиральные рукава
num_spiral_stars = 500
angle = 3.5
core_diameter = 120
spiral_stars = []
for i in range(num_spiral_stars):
    theta = i * angle
    r = math.sqrt(i) / math.sqrt(num_spiral_stars)
    spiral_stars.append((r * math.cos(theta), r * math.sin(theta)))
for x, y in spiral_stars:
    x = x * 350 + randint(-5, 3)
    y = y * 350 + randint(-5, 3)
    oval_size = randint(1, 3)
    c.create_oval(x-oval_size, y-oval_size, x+oval_size, y+oval_size,
                  fill='white', outline='')

# построить пучки
wisps = []
for i in range(2000):
    theta = i * angle
    # разделить на num_spiral_stars для получения более
    # качественных пыльных дорожек
    r = math.sqrt(i) / math.sqrt(num_spiral_stars)
    spiral_stars.append((r * math.cos(theta), r * math.sin(theta)))
for x, y in spiral_stars:
    x = x * 330 + randint(-15, 10)
    y = y * 330 + randint(-15, 10)
    h = math.sqrt(x**2 + y**2)
```

```
    if h < 350:
        wisps.append((x, y))
        c.create_oval(x-1, y-1, x+1, y+1, fill='white', outline='')

# построить ядро галактики
core = []
for i in range(900):
    x = randint(-core_diameter, core_diameter)
    y = randint(-core_diameter, core_diameter)
    h = math.sqrt(x**2 + y**2)
    if h < core_diameter - 70:
        core.append((x, y))
        oval_size = randint(2, 4)
        c.create_oval(x-oval_size, y-oval_size, x+oval_size, y+oval_size,
                      fill='white', outline='')
    elif h < core_diameter:
        core.append((x, y))
        oval_size = randint(0, 2)
        c.create_oval(x-oval_size, y-oval_size, x+oval_size, y+oval_size,
                      fill='white', outline='')

root.mainloop()
```

Построение галактической империи

empire_practice.py

```
"""Построить двухмерную модель галактики,
разместить кольца расширения галактической империи."""
import tkinter as tk
import time
from random import randint, uniform, random
import math
#=====================================================================
# ГЛАВНЫЕ ВХОДНЫЕ ДАННЫЕ

# местоположение родного мира галактической империи на карте:
HOMEWORLD_LOC = (0, 0)

# максимальное число лет для симуляции:
MAX_YEARS = 10000000

# средняя скорость расширения как доля скорости света:
SPEED = 0.005

# шкальные единицы
UNIT = 200
```

```
#=====================================================================
# настроить холст изображения
root = tk.Tk()
root.title("Галкатика Млечного Пути")
c = tk.Canvas(root, width=1000, height=800, bg='black')
c.grid()
c.configure(scrollregion=(-500, -400, 500, 400))

# фактические размеры Млечного Пути (в световых годах)
DISC_RADIUS = 50000

disc_radius_scaled = round(DISC_RADIUS/UNIT)

def polar_coordinates():
    """Сгенерировать равномерную случайную точку x,y
    внутри диска для 2-мерного изображения."""

    r = random()
    theta = uniform(0, 2 * math.pi)
    x = round(math.sqrt(r) * math.cos(theta) * disc_radius_scaled)
    y = round(math.sqrt(r) * math.sin(theta) * disc_radius_scaled)
    return x, y

def spirals(b, r, rot_fac, fuz_fac, arm):
    """Построить спиральные рукава для изображения tkinter,
    используя формулу логарифмической спирали.

    b = произвольная константа в уравнении логарифмической спирали
    r = радиус шкалированного галактического диска
    rot_fac = фактор поворота
    fuz_fac = случайный сдвиг в местоположении звезд в рукаве,
              применительно к переменной 'fuzz'
    arm = спиральный рукав (0 = главный рукав,
                            1 = тянущиеся позади звезды)
   """

    spiral_stars = []
    fuzz = int(0.030 * abs(r))    # случайно сдвинуть местоположения звезд
    theta_max_degrees = 520
    for i in range(theta_max_degrees): # range(0, 700, 2) для
                                       # отсутствующей черной дыры
        theta = math.radians(i)
        x = r * math.exp(b*theta) * math.cos(theta + math.pi * rot_fac)\
            + randint(-fuzz, fuzz) * fuz_fac
        y = r * math.exp(b*theta) * math.sin(theta + math.pi * rot_fac)\
            + randint(-fuzz, fuzz) * fuz_fac
        spiral_stars.append((x, y))
```

```
    for x, y in spiral_stars:
        if arm == 0 and int(x % 2) == 0:
            c.create_oval(x-2, y-2, x+2, y+2, fill='white', outline='')
        elif arm == 0 and int(x % 2) != 0:
            c.create_oval(x-1, y-1, x+1, y+1, fill='white', outline='')
        elif arm == 1:
            c.create_oval(x, y, x, y, fill='white', outline='')

def star_haze(scalar):
    """Случайно распределить тусклые звезды tkinter
    по галактическому диску.

    disc_radius_scaled = радиус галактического диска,
                         прошкалированный в диаметр радиопузыря
    scalar = множитель для варьирования числа размещаемых звезд
    """
    for i in range(0, disc_radius_scaled * scalar):
        x, y = polar_coordinates()
        c.create_text(x, y, fill='white', font=('Helvetica', '7'),
                      text='.')

def model_expansion():
    """Смоделировать расширение империи из родного мира
    с помощью концентрических кругов."""
    r = 0   # радиус от родного мира
    text_y_loc = -290
    x, y = HOMEWORLD_LOC
    c.create_oval(x-5, y-5, x+5, y+5, fill='red')
    increment = round(MAX_YEARS / 10)  # годовой интервал для размещения кругов
    c.create_text(-475, -350, anchor='w', fill='red',
                  text='Increment = {:,}'.format(increment))
    c.create_text(-475, -325, anchor='w', fill='red',
                  text='Скорость как доля света = {:,}'.format(SPEED))

    for years in range(increment, MAX_YEARS + 1, increment):
        time.sleep(0.5)    # задержать перед размещением
                           # нового круга расширения
        traveled = SPEED * increment / UNIT
        r = r + traveled
        c.create_oval(x-r, y-r, x+r, y+r, fill='', outline='red',
                      width='2')
        c.create_text(-475, text_y_loc, anchor='w', fill='red',
                      text='Годы = {:,}'.format(years))
        text_y_loc += 20
        # обновить холст с учетом нового круга;
        # функция mainloop() больше не нужна
        c.update_idletasks()
        c.update()
```

```
def main():
    """Сгенерировать изображение галактики,
    смоделировать расширение империи, выполнить главный цикл."""
    spirals(b=-0.3, r=disc_radius_scaled, rot_fac=2, fuz_fac=1.5, arm=0)
    spirals(b=-0.3, r=disc_radius_scaled, rot_fac=1.91, fuz_fac=1.5,
            arm=1)
    spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=2, fuz_fac=1.5, arm=0)
    spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=-2.09, fuz_fac=1.5,
            arm=1)
    spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=0.5, fuz_fac=1.5,
            arm=0)
    spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=0.4, fuz_fac=1.5,
            arm=1)
    spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=-0.5, fuz_fac=1.5,
            arm=0)
    spirals(b=-0.3, r=-disc_radius_scaled, rot_fac=-0.6, fuz_fac=1.5,
            arm=1)
    star_haze(scalar=9)
    model_expansion()

    # выполнить цикл модуля tkinter
    root.mainloop()

if __name__ == '__main__':
    main()
```

Окольный путь предсказания обнаруживаемости

rounded_detection_practice.py

```
"""Вычислить вероятность обнаружения радиопузыря диаметром 32 св. года
с учетом 15.6 М случайно распределенных цивилизаций в галактике."""
import math
from random import uniform, random
from collections import Counter

# единицы длины в св. годах
DISC_RADIUS = 50000
DISC_HEIGHT = 1000
NUM_CIVS = 15600000
DETECTION_RADIUS = 16

def random_polar_coordinates_xyz():
    """Сгенерировать равномерную случайную точку xyz внутри
    3-мерного диска."""
    r = random()
    theta = uniform(0, 2 * math.pi)
```

```
    x = round(math.sqrt(r) * math.cos(theta) * DISC_RADIUS, 3)
    y = round(math.sqrt(r) * math.sin(theta) * DISC_RADIUS, 3)
    z = round(uniform(0, DISC_HEIGHT), 3)
    return x, y, z

def rounded(n, base):
    """Округлить число до ближайшего числа, обозначенного базовым
    параметром."""
    return int(round(n/base) * base)

def distribute_civs():
    """Распределить местоположения xyz в модели галактического диска
    и вернуть список."""
    civ_locs = []
    while len(civ_locs) < NUM_CIVS:
        loc = random_polar_coordinates_xyz()
        civ_locs.append(loc)
    return civ_locs

def round_civ_locs(civ_locs):
    """Округлить местоположения xyz и вернуть список округленных
    местоположений."""

    # конвертировать радиус в кубические размеры:
    detect_distance = round((4 / 3 * math.pi * DETECTION_RADIUS**3)**(1/3))
    print("\nрадиус обнаружения = {} LY".format(DETECTION_RADIUS))
    print("кубическое расстояние обнаружения = {} LY"
          .format(detect_distance))

    # округлить цивилизацию xyz до расстояния обнаружения
    civ_locs_rounded = []

    for x, y, z in civ_locs:
        i = rounded(x, detect_distance)
        j = rounded(y, detect_distance)
        k = rounded(z, detect_distance)
        civ_locs_rounded.append((i, j, k))

    return civ_locs_rounded

def calc_prob_of_detection(civ_locs_rounded):
    """Подсчитать местоположения и вычислить вероятность повторных
    значений."""
    overlap_count = Counter(civ_locs_rounded)
    overlap_rollup = Counter(overlap_count.values())
    num_single_civs = overlap_rollup[1]
    prob = 1 - (num_single_civs / NUM_CIVS)

    return overlap_rollup, prob
```

```
def main():
    """Вызвать функции и напечатать результаты."""
    civ_locs = distribute_civs()
    civ_locs_rounded = round_civ_locs(civ_locs)
    overlap_rollup, detection_prob = calc_prob_of_detection(civ_locs_rounded)
    print("длина предокругленных civ_locs = {}".format(len(civ_locs)))
    print("длина округленных civ_locs_rounded = {}"
          .format(len(civ_locs_rounded)))
    print("overlap_rollup = {}\n".format(overlap_rollup))
    print("вероятность обнаружения = {0:.3f}".format(detection_prob))

    # Шаг контроля качества с проверкой округления
    print("\nПред- и постокругление первых 3-х местоположений:\n")
    for i in range(3):
        print("предокругление: {}".format(civ_locs[i]))
        print("постокругление: {} \n".format(civ_locs_rounded[i]))

if __name__ == '__main__':
    main()
```

Глава 11. Задача Монти Холла

Парадокс дня рождения

birthday_paradox_practice.py

```
"""Вычислить вероятность совместного дня рождения в расчете на x число
людей."""
import random

max_people = 50
num_runs = 2000

print("\nВероятность, что как минимум 2 человека имеют одинаковый день
                                                        рождения:\n")

for people in range(2, max_people + 1):
    found_shared = 0
    for run in range(num_runs):
        bdays = []
        for i in range(0, people):
            bday = random.randrange(0, 365)  # игнорировать високосные годы
            bdays.append(bday)
        set_of_bdays = set(bdays)
        if len(set_of_bdays) < len(bdays):
            found_shared += 1
```

```
    prob = found_shared/num_runs
    print("Число людей = {} Вероятн. = {:.4f}".format(people, prob))

print("""
Согласно парадоксу дня рождения, если в комнате находятся 23 человека,
то имеется 50%-й шанс, что 2 из них будут иметь одинаковый день рождения.
""")
```

Глава 13. Симуляция инопланетного вулкана

Полет пули

practice_45.py

```
import sys
import math
import random
import pygame as pg

pg.init()   # инициализировать модуль pygame

# определить цветовую таблицу
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
LT_GRAY = (180, 180, 180)
GRAY = (120, 120, 120)
DK_GRAY = (80, 80, 80)

class Particle(pg.sprite.Sprite):
    """Строит выбрасываемые частицы в симуляции вулкана."""

    gases_colors = {'SO2': LT_GRAY, 'CO2': GRAY, 'H2S': DK_GRAY, 'H2O': WHITE}

    VENT_LOCATION_XY = (320, 300)
    IO_SURFACE_Y = 308
    GRAVITY = 0.5     # пикселов на кадр
    VELOCITY_SO2 = 8  # пикселов на кадр

    # скаляры (атомная масса SO2/атомная масса частицы),
    # используемые для скорости
    vel_scalar = {'SO2': 1, 'CO2': 1.45, 'H2S': 1.9, 'H2O': 3.6}

    def __init__(self, screen, background):
        super().__init__()
        self.screen = screen
        self.background = background
        self.image = pg.Surface((4, 4))
```

```
        self.rect = self.image.get_rect()
        self.gas = 'SO2'
        self.color = ''
        self.vel = Particle.VELOCITY_SO2 * Particle.vel_scalar[self.gas]
        self.x, self.y = Particle.VENT_LOCATION_XY
        self.vector()

    def vector(self):
        """Рассчитать конкретный вектор при запуске."""
        angles = [65, 55, 45, 35, 25]   # 90 значит вертикально
        orient = random.choice(angles)
        if orient == 45:
            self.color = WHITE
        else:
            self.color = GRAY
        radians = math.radians(orient)
        self.dx = self.vel * math.cos(radians)
        self.dy = -self.vel * math.sin(radians)  # отрицательно
                                                 # по мере роста y вниз

    def update(self):
        """Применить гравитацию, нарисовать траекторию
        и обработать граничные условия."""
        self.dy += Particle.GRAVITY
        pg.draw.line(self.background, self.color, (self.x, self.y),
                     (self.x + self.dx, self.y + self.dy))
        self.x += self.dx
        self.y += self.dy
        if self.x < 0 or self.x > self.screen.get_width():
            self.kill()
        if self.y < 0 or self.y > Particle.IO_SURFACE_Y:
            self.kill()

def main():
    """Настроить, открыть окно и запустить цикл игры."""
    screen = pg.display.set_mode((639, 360))
    pg.display.set_caption("Симулятор вулкана на Ио")
    background = pg.image.load("tvashtar_plume.gif")

    # Настроить цветную легенду
    legend_font = pg.font.SysFont('None', 26)
    text = legend_font.render('Белый = 45 градусов', True, WHITE, BLACK)

    particles = pg.sprite.Group()

    clock = pg.time.Clock()
```

```
    while True:
        clock.tick(25)
        particles.add(Particle(screen, background))
        for event in pg.event.get():
            if event.type == pg.QUIT:
                pg.quit()
                sys.exit()

        screen.blit(background, (0, 0))
        screen.blit(text, (320, 170))

        particles.update()
        particles.draw(screen)

        pg.display.flip()

if __name__ == "__main__":
    main()
```

Глава 16. Выявление мошенничества с помощью закона Бенфорда

Победа над Бенфордом

beat_benford_practice.py

```
"""Подделать подсчет голосов так, чтобы окончательные результаты
сочетались с законом Бенфорда."""

# предметом приведенного ниже примера являются
# результаты Трампа и Клинтон в шт. Иллинойс
# на президентских выборах 2016 г.

def load_data(filename):
    """Открыть текстовый файл с количеством и явкой в список целых
    чисел."""
    with open(filename) as f:
        lines = f.read().strip().split('\n')
        return [int(i) for i in lines]  # конвертировать симв. цепочки
                                        # в целые числа

def steal_votes(opponent_votes, candidate_votes, scalar):
    """Применить скаляр для снижения количества голосов для одного
    и увеличить для другого, вернуть в виде списков.
```

```
    Аргументы:
    opponent_votes - похищаемые голоса
    candidate_votes - голоса, увеличиваемые на количество украденных
    scalar - долевой процент, < 1, используется для снижения голосов

    Возвращает:
    Список измененных голосов оппонента
    Список измененных голосов кандидата

    """
    new_opponent_votes = []
    new_candidate_votes = []
    for opp_vote, can_vote in zip(opponent_votes, candidate_votes):
        new_opp_vote = round(opp_vote * scalar)
        new_opponent_votes.append(new_opp_vote)
        stolen_votes = opp_vote - new_opp_vote
        new_can_vote = can_vote + stolen_votes
        new_candidate_votes.append(new_can_vote)
    return new_opponent_votes, new_candidate_votes

def main():
    """Выполнить программу.
    Загрузить данные, задать целевое победное количество голосов,
    вызвать функции, показать результаты анализа закона Бенфорда.
    """

    # загрузить данные голосования
    c_votes = load_data('Clinton_votes_Illinois.txt')
    j_votes = load_data('Johnson_votes_Illinois.txt')
    s_votes = load_data('Stein_votes_Illinois.txt')
    t_votes = load_data('Trump_votes_Illinois.txt')
    total_votes = sum(c_votes + j_votes + s_votes + t_votes)

    # допустим, что Трамп набирает множество голосов с 49%
    t_target = round(total_votes * 0.49)
    print("\nПобедная цель Трампа = {:,} votes".format(t_target))

    # вычислить лишние голоса, необходимые для победы Трампа
    extra_votes_needed = abs(t_target - sum(t_votes))
    print("extra votes needed = {:,}".format(extra_votes_needed))

    # вычислить скаляр, необходимый для генерирования лишних голосов
    scalar = 1 - (extra_votes_needed / sum(c_votes + j_votes + s_votes))
    print("scalar = {:.3}".format(scalar))
    print()

    # перевернуть подсчеты голосов, опираясь на скаляре,
    # и построить новый комбинированный список голосов
    fake_counts = []
```

```
    new_c_votes, new_t_votes = steal_votes(c_votes, t_votes, scalar)
    fake_counts.extend(new_c_votes)
    new_j_votes, new_t_votes = steal_votes(j_votes, new_t_votes, scalar)
    fake_counts.extend(new_j_votes)
    new_s_votes, new_t_votes = steal_votes(s_votes, new_t_votes, scalar)
    fake_counts.extend(new_s_votes)
    fake_counts.extend(new_t_votes)  # добавить последний,
                                     # т. к. до сих пор изменялся

    # сравнить старое и новое количества голосов и итоговые числа
    # в табличной форме;
    # подменить "Трампа" на "Клинтон", если необходимо
    for i in range(0, len(t_votes)):
        print("старый Трамп: {} \t новый Трамп: {} \t "
              "старая Клинтон: {} \t " новая Клинтон: {}"
              .format(t_votes[i], new_t_votes[i], c_votes[i],
                      new_c_votes[i]))
        print("-" * 95)
    print("TOTALS:")
    print("старый Трамп: {:,} \t новый Трамп: {:,} \t "
          "старая Клинтон: {:,} новая Клинтон: {:,}"
          .format(sum(t_votes), sum(new_t_votes),
                  sum(c_votes), sum(new_c_votes)))

    # записать текстовый файл для использования
    # как вход в программу benford.py;
    # указанная программа будет проверять соответствие фиктивных
    # голосов закону Бенфорда
    with open('fake_Illinois_counts.txt', 'w') as f:
        for count in fake_counts:
            f.write("{}\n".format(count))

if __name__ == '__main__':
    main()
```

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Д

З

И

К

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

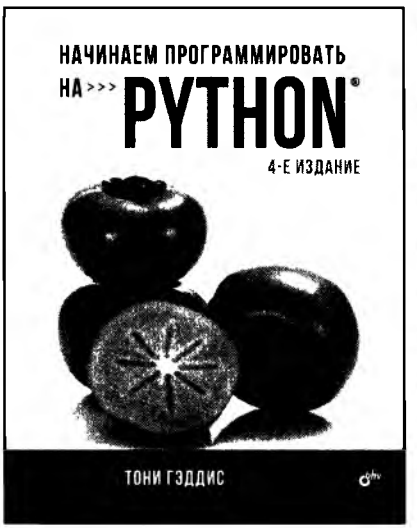
Э

www.bhv.ru

Гэддис Т.

Начинаем программировать на Python, 4-е изд.

Отдел оптовых поставок:
E-mail: opt@bhv.ru



- Краткое введение в компьютеры и программирование
- Ввод, обработка и вывод данных
- Управляющие структуры и булева логика
- Структуры с повторением и функции
- Файлы и исключения
- Списки и кортежи
- Строковые данные, словари и множества
- Классы и объектно-ориентированное программирование
- Наследование и рекурсия
- Функциональное программирование

В книге изложены принципы программирования, с помощью которых вы приобретете навыки алгоритмического решения задач на языке Python, даже если у вас нет опыта программирования. Для облегчения понимания сути алгоритмов широко использованы блок-схемы, псевдокод и другие инструменты. Приведено большое количество сжатых и практичных примеров программ. В каждой главе предложены тематические задачи с пошаговым анализом их решения.

Отличительной особенностью издания является его ясное, дружественное и легкое для понимания изложение материала.

Книга идеально подходит для вводного курса по программированию и разработке программного обеспечения на языке Python.

Тони Гэддис — ведущий автор всемирно известной серии книг «Начинаем программировать...» (Starting Out With) с двадцатилетним опытом преподавания курсов информатики в колледже округа Хейвуд, шт. Северная Каролина, удостоен звания «Преподаватель года», лауреат премии «Педагогическое мастерство».

Моделируй, экспериментируй, играй

Включая версию Python 3

Данная книга — это набор забавных, в том числе образовательных, проектов, предназначенных для развлечения программистов и одновременного повышения их навыков. Это хорошее дополнение к традиционным самоучителям, отличная «следующая книга», расширяющая полученные ранее навыки и знакомящая с новыми полезными инструментами.

Каждый проект включает в себя интригующий поворот с историческими событиями, литературными персонажами или ссылками на поп-культуру — и все это используя модули tkinter, matplotlib, cProfile, Pylint, pygame, pillow и python-docx.

Потренируйтесь в решении задач, чтобы:

- Помочь Джеймсу Бонду взломать высокотехнологичный сейф с использованием алгоритма восхождения к вершине холма
- Сочинить стихи с помощью анализа марковских цепей
- Породить расу гигантских крыс, применив генетические алгоритмы
- Запланировать обеспеченную жизнь на пенсии с использованием метода Монте-Карло
- Смоделировать Млечный Путь и рассчитать наши шансы обнаружить инопланетные цивилизации
- Нарисовать карту Марса и изучить орбитальную механику с использованием вашего собственного космического спутника

И проделать многое другое! Если вы хотите приобрести немного новых навыков в программировании или же просто слегка творчески поупражняться, то с данной книгой вы получите бесконечное удовольствие и станете настоящим фанатом этого языка.

Об авторе

Ли Воган — программист, энтузиаст поп-культуры и педагог. Профессионал в сфере построения и анализа компьютерных моделей, разработки, тестирования и коммерциализации программного обеспечения, а также подготовки ИТ-специалистов. Эта книга написана им с целью помочь читателям отточить свои навыки программирования на Python и получить от этого удовольствие!

no starch press

bhv

ISBN 978-5-9775-6751-0

191036, Санкт-Петербург, Гончарная ул., 20
Тел.: (812) 717-10-50, 339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru