Зед А. Шоу

# Легкий способ выучить Python 3 ЕЩЕ ГЛУБЖЕ

Уникальная методика обучения программированию для начинающих


Мировой
компьютерный
бестселер

Zed A. Shaw

# Learn More Python 3

Addison
Wesley

Зед А. Шоу

# Легкий способ выучить Python 3

## ЕЩЕ ГЛУБЖЕ

Москва
2020

УДК 004.43
ББК 32.973.26-018.1
Ш81

Zed A. Shaw
LEARN MORE PYTHON 3 THE HARD WAY
The Next Step for New Python Programmers 1st Edition

Authorized translation from the English language edition, entitled Learn More Python 3 the Hard Way: The Next Step for New Python Programmers 1st edition; ISBN 0134123484; by Zed A. Shaw; published by Pearson Education, Inc., publishing as Addison-Wesley Professional.


**Шоу, Зед.**

Ш81 Легкий способ выучить Python 3 еще глубже / Зед Шоу ; [перевод с английского М.А. Райтмана]. — Москва : Эксмо, 2020. — 272 с. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-093107-1

Воплотите ваши идеи в код самого высокого качества!

Зед Шоу — один из тех, кто по-настоящему разбирается в Python. Его советы помогли миллионам программистов по всему миру, помогут они и вам. От вас потребуются лишь дисциплина, желание и упорство, все остальное вы найдете в книге «Легкий способ выучить Python 3 еще глубже». Это вторая часть «Легкого способа выучить Python 3», где Зед описывал базовые принципы программирования на Python 3. Вторая часть углубит ваши знания и поможет приобрести новые навыки с помощью 52 прекрасно составленных заданий.

**УДК 004.43**
**ББК 32.973.26-018.1**

**ISBN 978-5-04-093107-1**

# Содержание

# Предисловие автора

Движение, креативность и качество. Тщательно заучивайте эти три слова в процессе чтения данной книги. Движение. Креативность. Качество. В книге есть множество упражнений на изучение важных тем, в которых должен разбираться каждый программист, но настоящие знания заключаются в этих трех словах. При написании этой книги по программированию моей целью было обучить вас тому, что, в моем представлении, является тремя наиболее важными основами для любого программиста. Без движения вы будете бултыхаться, не зная, как приступить к работе, и у вас появятся проблемы с сохранением прогресса на длинных дистанциях. Без креативности вы не сможете решать проблемы, с которыми как программист будете сталкиваться каждый день. Без качества вы не поймете, хорошо ли вообще то, что вы делаете.

Обучить трем указанным принципам нетрудно. Я мог бы просто написать в блоге три поста и сказать: «Ну вот, теперь вы знаете, что означают эти три слова». Это не сделает вас лучше как программиста и, определенно, не сделает вас человеком, который смог бы самостоятельно заниматься разработкой в течение следующих 10 или 20 лет. Иметь представление о чем-то не подразумевает умение применять это на практике. Чтение поста о креативности не позволит узнать, насколько креативны вы сами. Для настоящего понимания этих сложных тем их необходимо усвоить, и лучший способ сделать это – применить их в простых проектах.

Когда вы приступите к выполнению упражнений в этой книге, я сообщу вам, над какой из тем вы будете работать. В этом отличие от других моих книг, где я «подло» обучаю вас принципам без вашего ведома. На сей раз я буду откровенен, ведь мне важно, чтобы вы держали принцип в уме и могли практиковать его во время упражнения. Затем вы оцените, насколько удачной была ваша попытка применить знание на практике, и подумаете, что предпринять, чтобы в следующий раз сделать ее более удачной. В этой книге ключевым моментом является способность рефлексировать над своими возможностями и самосовершенствоваться. Лучше всего это делать, сосредоточиваясь на одной технике или виде практики за раз, выполняя одновременно какие-либо другие задачи.

В дополнение к движению, креативности и качеству вы также узнаете, в каких шести важных вопросах, по моему мнению, должен разбираться современный программист. Эти темы могут измениться в будущем, но вот уже на протяжении десятилетий они оказывались совершенно необходимыми, и,

если не случится резкого изменения в технологиях, по-прежнему останутся актуальными. Даже что-то вроде языка SQL, о котором говорится в части VI по-прежнему применяется, поскольку учит вас тому, как структурировать данные, чтобы в дальнейшем они не разваливались на части. Ваши вторичные образовательные цели заключаются в следующем:

начало работы: вы научитесь быстро начинать проект;

структуры данных: я не рассматриваю каждую отдельную структуру данных, но подвожу вас к более полному их изучению;

алгоритмы: структуры данных бессмысленны без способа их обработки;

разбор (анализ) текста: это основа компьютерных наук, и умение его выполнять поможет вам в изучении языков программирования;

моделирование данных: я использую SQL, чтобы научить вас основам моделирования хранимых данных логичным способом;

инструменты Unix: инструменты командной строки используются на протяжении всей книги в качестве копируемых проектов; затем вы также изучите продвинутые инструменты командной строки Unix.

В каждой части книги внимание будет уделяться одному, двум или трем упражнениям за раз, пока, наконец, в завершающей части VII вы не примените все знания при создании простого веб-сайта. Ваши итоговые проекты не будут выглядеть привлекательно. Вы не узнаете, как создать свой следующий стартап. Но эти приятные небольшие проекты позволят вам использовать то, что вы узнаете при изучении Django.

## Все это персонально

Множество книг учат этим трем принципам в контексте работы в команде. Когда в них идет речь о движении, все сводится к тому, что вы работаете над проектом по поддержанию кода с другим человеком. Когда речь идет о творчестве – к тому, как вы со своей командой отправляетесь на встречу, чтобы задавать вопросы клиентам. К сожалению, большая часть этих «профессиональных» книг по-настоящему не обучают качеству. В этом нет ничего страшного, но у многих новичков с таким «командным» подходом возникает две проблемы.

1. У вас нет команды, поэтому применять на практике то, что изучаете, вы не можете. Книги, ориентированные на командное взаимодействие, предназначены для младших программистов, у которых уже есть работа – и необходимость работать в новой команде. Пока вы не окажетесь в таком положении, любая подобная книга будет для вас бесполезна.

2. Какой смысл учиться работать в команде, если ваш собственный уровень движения, креативности и качества чрезвычайно низок? Что бы там ни говорили приверженцы «командной игры», подавляющее большинство программистских задач выполняются в одиночку, как и процедура вашей оценки собственных навыков. Если вы работаете в команде, но при этом пишете низкокачественный код, так что постоянно приходится просить членов команды о помощи, ваш босс не отзовется о вас хорошо. При всех разговорах о прелестях командного подхода, они никогда не винят команду, если программист не может работать в одиночку. Они винят самого программиста.

Эта книга не о том, как стать хорошим рабочим трутнем в ООО «Мега Компания». Ее предназначение – помочь вам улучшить свои навыки, чтобы вы смогли действовать в одиночку, когда получите работу. Если вы будете совершенствовать лично себя, тогда вы естественным образом станете сильнее и в командном смысле тоже. Это также значит, что вы сможете создавать и развивать собственные идеи, с чего и начинается подавляющее большинство проектов.

## Используйте приложенные видеоролики

К «Легкому способу продолжить изучение *Python 3* еще глубже» прилагается обширный набор видеороликов, иллюстрирующих процесс работы кода, отладки, и, что важнее, решения к заданиям. Эти видео – лучшее место для демонстрации многих распространенных ошибок. Ошибки нарочно вносятся в код Python, чтобы затем можно было показать, как все исправить. Кроме того, я использую различные хитрости и приемы отладки и получения данных. На видео я показываю, как «перестать пялиться и спросить» код, что не так. Эти видео можно просмотреть онлайн по адресу **informit.com/title/9780134123486**.

Зарегистрируйте свою копию «Легкого способа выучить *Python 3* еще глубже» на сайте InformIT, чтобы получать доступ к обновлениям и исправлениям по

мере их появления. Для начала процесса регистрации перейдите по адресу **informit.com/register**, затем войдите в систему или создайте учетную запись. Введите номер ISBN (9780134123486) и ответьте на простой вопрос для подтверждения покупки. Затем перейдите на вкладку «Зарегистрированные продукты» (Registered Products) и найдите ссылку на бонусный контент рядом с соответствующим продуктом. Перейдите по этой ссылке, чтобы получить доступ к бонусным материалам.

# Часть I

## Начальные знания

Первое, что вам необходимо выучить, – это все. Знаю, звучит пугающе, но, как я уже упомянул в предисловии, эта книга будет обучать вас лишь трем навыкам. В каждом упражнении вы станете подкреплять каждый из навыков при выполнении других задач. Я могу сказать вам: «сделать копию команды `cat`», — но что вы будете делать в действительности, так это развивать свою креативность. Я могу приказать «создать структуру данных связанного списка», но на самом деле вы примените процессы структурированного обзора кода в своей программистской практике. Секрет этой книги заключается в том, чтобы использовать проекты и упражнения как средства для изучения трех важных практик: движения, креативности и качества.

В идеале в этих трех концептах нет ничего сверхъестественного. Движение – это просто шаги, которые вы совершаете при создании чего-либо. Креативность – всего лишь способ создания и воплощения идей в жизнь. Используя качество, вы убеждаетесь, что эти воплощения не мусор. Самое же приятное в применении. Как вы примените движение к навыкам личностного развития? Как поймете, создали вы качественное программное обеспечение или

нет? Как возьмете идею и сделаете ее реальностью? Все три принципа взаимосвязаны, так как вам необходимо движение, чтобы развить креативность и обеспечить качество, что также требует креативности, поскольку движение не длится постоянно. Это красивый порочный круг.

Процесс обучения по этой книге таков.

1. В каждой части книги я буду давать вам цель работы над движением, креативностью или качеством. Обычно это будет две концепции за раз; иногда только одна. Например, в части II вы станете работать над креативностью, на протяжении 45 минут осуществляя простой взлом. Вы также проанализируете свое начальное движение, и если вам окажется трудно сдвинуться с места, значит, вы не были достаточно креативными.

2. В начале каждого упражнения вам будет даваться наводка или цель, над которой нужно подумать во время выполнения упражнения. Каждая из них предложит сосредоточиться на одном или нескольких аспектах вашей работы. Упражнение 4 в части II просто дает задачу реализовать что-то, а затем в упражнении 5 уже нужно перечислить тормозящие вас факторы и попытаться устранить их или уменьшить их влияние. Другие упражнения советуют осмотреть свое рабочее место и убрать все, что может вас отвлечь. В каждом упражнении вы станете размышлять над этими целями, а затем работать над упражнением, пытаясь сосредоточиться на конкретной задаче.

3. В конце каждого упражнения есть практические задания для дополнительной работы. Они могут быть непосредственно связаны с проектом или просто относиться к проблемам движения, креативности и качества, с которыми вы имеете дело в упражнении.

4. Некоторые упражнения имеют повышенный уровень сложности. Это означает, что вам дается описание инструмента для реализации, обычно основанного на существующем инструменте Unix, а затем предлагается реализовать его, но никакого кода не предоставляется. Возможно, сначала вам нужно будет изучить небольшие фрагменты образца кода, но, как правило, в этих упражнениях Python не используется. Решения доступны в интернете в проекте Git на Github по адресу **bit.ly/lmpthwsolve**.

5. Другие упражнения будут представлены в виде описаний чего-либо, что вам необходимо реализовать, основываясь на моем коде. В этих упражнениях будут объясняться определенные вещи (например,

алгоритм), а затем вам придется реализовать их как можно точнее и найти как можно больше ошибок. Обычно такие упражнения нацелены на повышение качества, так что вам предложат пройти автоматизированные тесты, отследить частоту ошибок и решить дополнительные задания.

6. Наконец, вы будете использовать записную книжку, чтобы делать заметки и отслеживать показатели, которые пригодятся вам для совершенствования своего способа работать. Крайне важно относиться к этому как к дневнику, личной учетной записи, содержащей историю вашего развития, которой вы ни с кем не должны делиться, в особенности с вашим менеджером. Такая информация может быть использована для извлечения выгоды из вас как работника, поэтому храните ее в надежном месте.

Ваша цель при чтении этой книги состоит не только в том, чтобы сделать пару копий некоторых инструментов Unix. Ваша цель – использовать эти небольшие проекты Unix, чтобы сосредоточиться на аспектах вашей способности работать над проектами более крупными.

## Что, если я ненавижу твое дурацкое персональное движение, Зед?

Никаких проблем. Эта книга задумывалась как нечто, что позволит вам расти и совершенствоваться. Если вы не в полной мере готовы анализировать то, как вы работаете, тогда отложите книгу на время. Можете просто выполнить все задания самостоятельно когда хотите, а затем вернуться и приступить к проектам с ограничениями в собственном движении. Каждое упражнение автономно и обеспечивает персональное развитие. Делайте, что можете, и возвращайтесь, когда вам понадобится понять, как вы работаете.

## Что, если окажется, что у меня все плохо?

Весьма вероятно, так и произойдет. Но мой метод заключается в том, чтобы помочь вам понять, почему у вас все плохо и что нужно сделать, чтобы это исправить. Тогда вопрос развития сведется лишь к объемам проделанной работы. Держите свой дневник в тайне, и никто не узнает, насколько у вас все плохо. Тогда совсем скоро вы узнаете, каков ваш уровень и на что нужно

обратить внимание. Больше никакого гадания, мошенник вы или действительно способны выполнять работу. Вы объективно оцените свои сильные и слабые стороны и перестанете беспокоиться о том, каково ваше место в этом мире.

Однако, скорее всего, вы не так ужасны, как думаете. Предназначение этой книги – быть личным курсом по улучшению объективной оценки своих навыков. Это означает, что вы должны сосредоточиться не на том, насколько вы хороши в чем-либо, но на том, насколько вы растете. Если ваш результат в конкретном упражнении вас расстраивает, нужно взять перерыв и подумать, что можно улучшить. Вы также должны взглянуть на это упражнение в контексте всех остальных, ранее выполненных, и сделать объективный вывод о том, насколько вы стали лучше. Сосредоточение внимания на улучшении помогает мыслить объективно (не положительно или отрицательно) и продолжать учиться.

УПРАЖНЕНИЕ 0

# Настройка

Чтобы пользоваться этой книгой, вам нужно установить и настроить несколько инструментов. Вероятно, большинство из них у вас уже установлены, но на всякий случай давайте перепроверим.

## Редактор программиста

Вам понадобится текстовый редактор для программирования, а не среда разработки, или IDE. Примеры текстовых редакторов программиста – Vim, Emacs и Atom. Это не упрощенные текстовые редакторы, оперирующие только текстом. Они разработаны для того, чтобы помогать вам управлять целыми проектами и одновременно работать с большим количеством файлов. В них также присутствуют базовые функции IDE, такие как возможность запуска встроенных команд, создания сценариев и т. д. Но есть одно ключевое различие: IDE обычно привязана к одному языку, она выполняет расширенный анализ источника и предоставляет вам кратчайшие пути для написания кода. Тогда вам не нужно ничего запоминать, ведь через большинство проектов можно просто проложить путь комбинацией клавиш **Ctrl+Пробел**. Это замечательно, когда у вас есть еще 100 разработчиков, которые работают с десятикратной мощностью и создают больше технического долга, чем вы можете вынести, но это ужасная функция, когда вы пытаетесь учиться. Другая проблема заключается в том, что вам приходится ждать, пока кто-то напишет IDE для новых языков. И если Microsoft или JetBrains язык не понравился, тогда вы в тупике.

Все, что вы можете сделать в IDE, вы также можете сделать в настоящем текстовом редакторе программиста. Так как редакторы вроде Vim, Emacs и Atom поддерживают сценарии и могут быть изменены, за ними будущее. Если Haskell++ станет следующим популярным языком программирования, вы сможете сейчас одновременно работать с ним и со всеми своими предыдущими проектами. Но если вы станете зависимыми от IDE, вам придется ждать, пока с языком не разберется кто-то другой.

Если вы только начинаете и хотите обзавестись достойным, но бесплатным редактором, обратите внимание на Atom (**atom.io**) или VisualStudio Code (**code.visualstudio.com**). Эти редакторы работают на всех платформах, которые я

использую в книге, поддерживают сценарии, имеют много плагинов и просты в использовании. Если хотите, вы также можете использовать Vim или Emacs.

## Python 3.6

Для этой книги необходимо наличие Python версии 3.6. Теоретически вы можете использовать Python 2.7, поскольку многие упражнения обходятся без кода. Тем не менее решения в видео будут представлены на Python 3.6, и официальный репозиторий для решений также будет на Python 3.6. Это значит, что у вас возникнут проблемы с переводом решений обратно на Python 2.7. Если вы не знаете Python 3.6, для овладения основами можете прочесть книгу «Легкий способ выучить Python 3»[1].

## Рабочий терминал

Если вы знакомы с «Легким способом выучить *Python 3*», то знаете, что я требую использовать Терминал (Terminal). На этом этапе вам уже нет необходимости рассказывать, как его устанавливать. Но на всякий случай на видео показаны несколько вариантов установки. Это видео пригодится пользователям Windows, поскольку особенности поддержки Терминала и написание сценариев оболочки в Microsoft сильно изменились, и теперь они поддерживают гораздо более широкий спектр инструментов Unix.

## Рабочая конфигурация pip+virtualenv

Эта книга требует установки огромного количества дополнительных библиотек и множества программного обеспечения. В мире Python это проще всего сделать при помощи инструментов `pip` и `virtualenv`. Инструмент `pip` устанавливает пакеты из интернета на компьютер, чтобы вы могли импортировать (`import`) их в сценарии Python. Проблема `pip` заключается в том, что вы вынуждены осуществлять установку в каталоги по умолчанию, для чего необходимо иметь права `root` или администратора. В качестве решения можно воспользоваться инструментом `virtualenv`, который создает в каталоге своего рода «песочницу пакетов python», а затем позволяет запускать `pip` для

[1] Книга вышла в 2019 году в издательстве Бомбора. – *Прим. ред.*

установки пакетов в нее. В приложенном видео я покажу вам, как установить конфигурацию `pip+virtualenv` для всех платформ и использовать ее.

## Записная книжка

Во время работы над проектами вы будете делать заметки и фиксировать различные показатели. Для этого вам пригодится записная книжка с бумагой в клетку или, возможно, с точками вместо строк для более удобного построения графиков, а также пенал с карандашами. Вы можете пользоваться, чем захотите, но эта книга уделяет внимание обстановке и за пределами компьютера, пытаясь изменить таким образом вашу точку зрения при решении проблем. Возможно, что бумагой вы пользовались дольше, чем компьютером (впрочем, это меняется со временем), и поэтому она может казаться вам более «настоящей», а компьютер – совершенно бессмысленным. Пишите сначала на бумаге, а затем вводите записи в компьютер, тогда вы сможете преодолеть эту сложность в восприятии. Наконец, рисовать на бумаге куда удобнее.

## Учетная запись на Github.com

Если у вас еще нет аккаунта на **github.com**, вам нужно будет его создать. Для всех видео и проектов я буду выкладывать бесплатный код для самопроверки. Если вы застрянете на месте, то сможете открыть страницу с проектом для этой книги и взглянуть, как я решил задание. Также в качестве упражнения я иногда буду предлагать вам исправить проект, в который целенаправленно внес ошибки.

## git

Если у вас уже есть учетная запись на **github.com**, вам также понадобится инструмент командной строки под названием `git`. На **github.com** есть подробная информация о том, как его установить. Но чтобы сделать все наилучшим образом, посмотрите видео.

## Дополнительно: программное обеспечение для записи с экрана

Это делать не обязательно, но если вы сможете установить программное обеспечение для захвата экрана и, в идеале, одновременного ведения записи своего лица, вам будет проще анализировать собственный рабочий процесс. Этот шаг не обязателен, так как полностью записывать свою работу, разбирая затем сделанную запись в попытках найти ключи к совершенству, может оказаться излишне трудоемким. Я делал так в течение какого-то времени, и это мне чрезвычайно помогло, но в то же время как будто убивало мою креативность. Мой совет – если вы можете найти и позволить себе приобрести подобное ПО, следует использовать его в тех случаях, когда просто не можете понять, что же делаете неправильно, и появляется необходимость проследить за собой во время работы. Я также считаю, что ведение записи лица и тела помогает проверить, соблюдаете ли вы правильную осанку; нет ли у вас привычек, которые могут причинить физическую боль. Но, опять же, записывать себя на протяжении всего рабочего дня – это слегка чересчур. Кроме того, вы не сможете осуществлять это в офисе с другими людьми.

## Дальнейшее обучение

Вот и все, что вам необходимо прямо сейчас. Далее в книге я буду время от времени давать инструкции насчет других вещей. Для завершения этого упражнения просмотрите соответствующее видео, а затем установите все, о чем я говорил. Если что-то у вас уже установлено, последуйте советам на видео и убедитесь, что все работает как надо и вы можете продолжать обучение по этой книге.

УПРАЖНЕНИЕ 1

# Движение

В мире разработки программного обеспечения существует два типа движения. Первый – командное движение, охватывающее такие вещи, как экстремальное программирование (англ. Extreme programming, XP), Scrum и Agile. Они призваны помочь команде специалистов скоординировать работу вокруг большой кодовой базы таким образом, чтобы не пересориться. Командное движение диктует то, как каждый человек будет координировать свои действия, а также стандарты поведения кода, отчетность и менеджмент. Обычно такие процессы сводятся к следующему.

1. Составьте список дел.
2. Сделайте все по списку.
3. Убедитесь, что все сделано правильно.

Проблемы с командным движением начинаются, когда оно пытается контролировать индивидуальные процессы, которые лучше не трогать. Экстремальное программирование, наверное, самое оскорбительное в этом отношении, поскольку здесь к каждому программисту приставлен другой программист, наблюдающий за его работой и кричащий на него всякий раз, как в текстовом редакторе появляется красный цвет. Я категорически против движения, которое навязывает людям элементы персонального поведения, если только это не происходит в образовательных целях. Это оскорбляет наш профессионализм и создает среду диктаторской снисходительности, которая не способствует повышению креативности или качества. Требование использования определенных методов персонального движения необходимо в образовательной среде, но не в рабочей. Например, я заставляю кого-либо заниматься парным программированием исключительно в том случае, если этот человек – младший программист или новичок в команде и, следовательно, должен учиться. После этого командное движение должно быть таким, чтобы каждый мог выполнять свою работу на требуемом уровне качества.

Второй тип движения – персональное. Эту идею я позаимствовал у художников, писателей и музыкантов. Часть развития человека как творческой личности, фокусирующейся на качестве, – это развитие движения, которое помогает вам последовательно выполнять работу. Фактически отсутствие представления о своем движении – признак того, что художник, музыкант или писатель является любителем. Обычно у людей, утверждающих, что у них нет

творческого движения, оно на самом деле есть; они просто не догадываются об этом и потому постоянно ошибаются. Большинство других творческих дисциплин разрабатывают стратегии и тактики, которые помогают строить из идей готовые произведения, не бросая все на полпути. Для художников это способ разбить проблему написания картин на логические шаги, выполнение которых с большей вероятностью завершится успехом. В случае с музыкантами это аналогичный процесс, сочетающийся с балансированием внутри структуры выбранного ими музыкального стиля. У писателей движение – это способ структурировать текст, чтобы он струился естественным образом и не изобиловал сюжетными дырами и логическими несоответствиями (то, чего большинство сценаристов на телевидении, похоже, не понимает).

В контексте программного обеспечения ваш персональный процесс должен осуществлять следующее.

1. Определять жизнеспособные идеи.

2. Помогать вам начать работу над этими идеями и быстро их менять.

3. На протяжении многих рабочих сессий постепенно улучшать ваши идеи таким образом, который позволяет избегать проблем или легко производить восстановление.

4. Обеспечить качество реализации ваших идей, чтобы позже вас не раздавил вес ошибок.

5. Подтверждать, что вы можете работать с другими (при желании).

Обратите внимание, я говорю, что вы не обязаны работать с другими. С момента появления открытого исходного кода концепция создания программного обеспечения предполагает непропорциональный спрос на общность. Если вы не хотите делиться или работать с другими, то вы оскорбляете само их существо, а ваше поведение считается антисоциальным. Проблема в том, что крайне малое количество креативных решений рождаются в группе, а те, что там рождаются, в итоге обычно оказываются совсем не креативными. Та самая творческая искра часто становится результатом появления идеи у одного или двух человек, которую они затем реализовывают из ничего. Для производства готового продукта может потребоваться большая команда, как в случае с книгами, фильмами и музыкальными альбомами. Многие другие творческие действия, вроде живописи или большинства изобразительных искусств, могут выполняться в одиночку.

Вы никогда не найдете художественную школу, в которой от художников требуют работать только в команде. Аналогично нет причин, по которым написание программного обеспечения не может быть сольным творческим процессом, подобным живописи и писательству. Программное обеспечение – модульная дисциплина. Это означает, что вы вправе создавать что-то полностью самостоятельно, а другие люди все еще могут это использовать, даже если они никогда с вами не говорили и никогда не работали над этим. Вы можете быть несносным человеком, и люди все равно станут использовать ваше программное обеспечение. То же самое с писательством и живописью. Существует море невежливых и злобных писателей, художников и музыкантов, которым, тем не менее, поклоняются миллионы людей.

Если вы начнете работать самостоятельно и кто-то попытается сказать вам, что вы должны поделиться, или назовет вас социопатом, то это они невежливы. У людей есть право на личные вещи, право работать в одиночку и создавать собственные проекты. Кажется, единственные, кто требует, чтобы вы вносили вклад в крупные проекты, это люди, которые начали эти крупные проекты и в итоге заберут все деньги. Поверьте мне. Я внес огромный вклад в мир программного обеспечения, и все же, когда я иду на конференцию, люди говорят мне, что я не принимаю должного участия, потому что не написал строки кода для их проекта (хотя они никогда не сделали ничего, чтобы помочь мне).

В этой книге, когда я говорю «движение», я имею в виду персональное движение. Я редко упоминаю что-либо, непосредственно относящееся к работе с другими, поскольку на эту тему уже написана гора книг. Но существуют лишь несколько книг, которые помогут вам работать над собственным движением и определять, что работает на вас и почему. Нет совершенно ничего плохого, эгоцентричного, жадного, антисоциального или оскорбительного в том, чтобы фокусироваться на самом себе и, таким образом, становиться лучше в любимом деле.

## Задача упражнения

Это упражнение заключается в простом описании того, что вы считаете своим движением, а также в обозначении своих проблем. На данном этапе вы, вероятно, не имеете представления о собственном рабочем процессе, так как вам не хватает опыта. Чтобы облегчить задачу, я составил список вопросов.

- Испытываете ли вы проблемы с работой над долгосрочными проектами?

- Склонны ли вы писать код с ошибками, не понимая, почему так получается?
- Гоняетесь ли вы за языками программирования, так ничего и не реализуя?
- Вы никогда не вспоминаете API? (Да, я тоже не вспоминаю.)
- Ощущаете ли вы себя подчиненным или мошенником, которого скоро поймают?
- Вы беспокоитесь о том, являетесь ли вы «настоящим программистом»?
- Вы не можете придумать, как извлечь идею из своей головы и воплотить ее в коде?
- Испытываете ли вы проблемы с началом работы?
- Работаете ли вы в беспорядочной среде?
- Вы не знаете, как продвинуть проект дальше после его первой реализации?
- Продолжаете ли вы нагромождать код поверх кода, пока не получится беспорядок?

Подумайте над этими вопросами, потом попытайтесь описать ваши действия во время работы над проектом. Если у вас нет опыта работы, напишите, как, по вашему мнению, вы должны работать.

## Практические задания

1. Придумайте и запишите еще несколько вопросов, подобных этим. Затем дайте на них ответы.
2. Спросите у ваших знакомых программистов, что является их движением. Не ожидаю, что вы получите ответ.

## Дальнейшее обучение

Помните: то, что люди описывают как свое движение, и само движение могут значительно отличаться. Мы, человеческие создания, склонны запоминать события в гораздо более позитивном и логичном ключе, чем они происходили на самом деле. Читая эту книгу, вы нарушите указанную привычку и будете использовать записанные вами внешние показатели (и, возможно, запись с экрана), чтобы узнавать наверняка, что делаете. Вы не обязаны делать это отныне и впредь, но так вы окажете себе услугу, работая над улучшением своих навыков кодинга. Но когда вы спросите у более успешного программиста, что является его движением, просто помните, что он такими вещами не занимался и, скорее всего, его ответ на самом деле не соответствует тому, что он делает. Если вы сможете найти более опытного программиста, который согласится записывать свой экран во время работы, это может оказаться более поучительным, чем самим спрашивать его. Я предложил бы перейти к скринкастам других программистов и просто понаблюдать, как они решают проблемы и ведут записи.

# Креативность

В креативности нет ничего особенного. Если вы человек со средним (или выше) уровнем интеллекта, значит, вы креативны. Способность генерировать мысли и идеи и затем воплощать их в жизнь – обычный аспект человеческого мышления. Проблема в том, что креативность стала визитной карточкой особого магического класса людей под названием "Творческие люди". Существуют целые книги, посвященные мифическому священнику от мира искусства, который может вообразить идею и одним взмахом своих безумно креативных золотых рук создать чистое, эмоциональное, гениальное, эмпатическое произведение искусства, заставляющее детей небесных проливать слезы из чистой платины. Честно говоря, слово «креативность» – это избитое клише, которое используется, чтобы изолировать людей от реализации собственных идей. Но у меня нет иного выбора, кроме как использовать это слово в своей книге.

Здесь под ним подразумевается лишь воплощение идеи в жизнь. Я не связываю с этим словом никакой позиции превосходства и не считаю, что люди, умеющие материализовать свои мысли, обладают некой волшебной значимостью. Единственное различие между мной – предположительно очень креативным человеком – и вами в том, что я тренировался воплощать идеи в жизнь. Я завел блокнот, заношу туда идеи и стараюсь регулярно их реализовывать. Я изучаю живопись, музыку, писательство и программирование как средства реализации и воплощения своих мыслей в реальность. Просто пытаясь создать что-то на регулярной основе, я стал в этом лучше, и здесь нет никакой магии. Я просто продолжаю пытаться, пока не смогу это сделать.

В процессе обучения созданию того, что у меня на уме, я произвел непостижимое количество гор мусора, но на самых вершинах этих гор находятся отдельные работы, которые приводят меня в восторг. Если вы хотите работать над своей способностью создавать, у вас также будут свои кучи мусора. Но вы не можете просто случайно создать одну из них и надеяться, что, когда достигнете вершины, у вас все будет получаться. Ключ к тому, чтобы быть продуктивным креативным человеком, – учиться реализовывать ваши идеи внутри движения или набора ограничений, ведущего вас по пути обучения, но при этом избегать ловушек строгой прямолинейности, убивающей креативность. Вы балансируете на границе между движением, которое направляет вас, и движением, убивающим ваши идеи. Надеюсь, что благодаря этой книге вы найдете то верное место.

## Задача упражнения

Чтобы начать работу над своей креативностью, вам сначала нужно поработать над умением генерировать случайные мысли. Думаю, одна из моих самых сильных сторон заключается в придумывании двух вроде бы случайных идей и превращении их в нечто интересное или полезное. Можете начать практиковаться в этом, каждый день выполняя такое небольшое упражнение.

1. Запишите словосочетания как минимум из трех случайных слов. В дурацких лесах водятся игуаны. Символизм порождает крепы. Python может призывать пришельцев.

2. Теперь за 10 минут напишите эссе об этих трех словах (или одном из них), передав в нем как можно больше своих чувств – зрение, слух, осязание, обоняние, вкус, чувство равновесия. Подумайте, сколько чувств понадобится человеку, чтобы понять то, что вы написали. Не цензурируйте текст, просто позвольте словам струиться. Вы также можете визуально изобразить идею или сочинить стихотворение.

3. По ходу этого упражнения вам в голову могут внезапно прийти настоящие идеи, относящиеся к программному обеспечению или другим областям вашего интереса. Запишите их в надежном месте, чтобы изучить впоследствии, или даже нарисуйте, если сможете.

Хотите верьте, хотите нет, но одно такое простое упражнение совершенствует великое множество различных вещей, связанных с созданием программного обеспечения.

1. Оно учит, как позволять идеям свободно течь, не редактируя их.

2. Оно учит строить свободные ассоциации между, казалось бы, несвязанными идеями с целью находить возможные связи.

3. Оно открывает глаза на возможность обходиться без самокритики.

4. Оно совершенствует способность выражать мысли в письменной форме или рисунке, что обычно является первым шагом к воплощению идей в жизнь.

5. Оно заставляет вас представить, как ваши чувства работают для вас и для других людей, что помогает вам реализовать идеи в реальном мире.

6. Оно также заставляет людей думать, что они глубокие, талантливые художники. После выполнения этого упражнения вам может захотеться пойти купить себе берет и переехать в Париж.

Такой процесс случайного записывания и размышления об абсурдных концепциях может показаться трудным людям, которые привыкли потеть над деталями программного обеспечения и беспокоиться о его качестве. Это совершенно понятно, и, конечно, вам по-прежнему необходимо то чувство качества, что вы развили. Креативность без критического взгляда на качество производит лишь мусор. Тем не менее одно качество без креативности не предполагает воображения, необходимого, чтобы понять, что именно в вашей работе может пойти не так. Что вам нужно, так это то сочетание креативности и качества, которое поможет вам создавать программное обеспечение и быть уверенным в его надежности.

## Практическое задание

Если вам не нравится составлять случайные конструкции вроде «Унитарии склонны летать на омлетах», тогда можете просто выбрать случайное слово в словаре и написать о нем с позиции своих чувств. Это работает так же хорошо, и вы не ощущаете себя легкомысленными. Тем не менее я буду требовать от вас быть немного легкомысленными. Никого еще не уволили за стихи о золотистых пчелах на жемчужном берегу. Другой вариант – выразить то, что вы чувствуете, с точки зрения каждого из своих чувств. Это тоже поможет вам развить креативность и к тому же производит терапевтический эффект.

УПРАЖНЕНИЕ 3

# Качество

Сейчас я предложу научную теорию о познании, которую не могу доказать.

Когда вы припоминаете то, что сделали, конечный продукт кажется вам безошибочным.

Эта теория выведена из моих наблюдений, сделанных во время творческих занятий; дело обстоит примерно так.

1. Вы создаете нечто, на что уходит много времени. Это может быть программное обеспечение, живопись, написание произведения или что-либо другое, отнимающее время.

2. Вы «завершаете» работу и отступаете на шаг, чтобы полюбоваться результатами, и тут заходит ваш друг.

3. Друг указывает на одну вопиюще очевидную проблему; все ваше видение созданного внезапно меняется.

4. Теперь все, о чем вы думаете, это та ошибка, на которую указал ваш друг, и у вас нет ни малейшего понятия, как вы могли пропустить ее.

Я полагаю, такое явление происходит по причине того, что ваши воспоминания о процессе создания вещи влияют на ваш же образ ее восприятия. Акт создания воспринимается как благоприятное течение рабочих идей, так что воспоминания становятся нейтральными либо положительными. Тогда это разукрашивает ваше восприятие работы, заставляя думать, что все было гораздо лучше, чем в действительности, пряча множество изъянов и нюансов. Дело также в эмоциональной привязанности к плодам вашей работы, возникающей вследствие того, что она сделана вами и вы помните, как выполняли ее, – это мешает трезвой оценке. Однако у вашего друга отсутствуют эти воспоминания, и он смотрит на проделанную работу объективно, чем облегчает для себя обнаружение недостатков. Вот почему редакторы находят гораздо больше ошибок, чем сами авторы. Или почему профессионалы в области безопасности находят куда больше недостатков в программном обеспечении, чем те, кто создавал его. У этих внешних рецензентов просто нет эмоциональной привязанности и нет воспоминаний о процессе создания, поэтому они видят все более четко.

В мире живописи это так распространено, что художники придумывают различные трюки, дабы нивелировать влияние такого явления. Эти трюки упомянуты еще в записных книжках Леонардо да Винчи, и их предназначение – позволить живописцу взглянуть на работу с точки зрения своего критически настроенного друга.

- Переверните рисунок вверх ногами и взгляните на него с большего расстояния. Это поможет определить очевидные проблемы с цветом и контрастностью, а также укажет на повторяющиеся формы, от которых необходимо избавиться. В изобразительном искусстве повторяющиеся формы нежелательны.

- Поднесите рисунок к зеркалу, что отразит его по горизонтали – тогда ваш мозг не сможет определить принцип, согласно которому рисунок создан. Отражение превратит его в совершенно новый, ранее вами не виденный рисунок, и вот вы уже стали тем самым противным критически настроенным другом.

- Посмотрите на рисунок сквозь красное стекло или поднесите его к черному зеркалу. Так вы увидите его в черно-белом варианте и заметите слишком яркие или слишком темные области, из-за которых рисунок выглядит странно в цвете.

- Посмотрите на рисунок и на изображаемый объект сверху в зеркало, которое переворачивает все вверх ногами, и сравните их отражения. Этот метод укажет на очевидные проблемы с рисовкой, замаскировав и предмет, и рисунок под абстрактные фигуры, о которых у вашего мозга нет воспоминаний.

- Уберите рисунок подальше на несколько месяцев, чтобы забыть о том, как вы его создавали, а затем достаньте и снова взгляните на него.

- Попросите взглянуть на рисунок своего противного друга и сказать, что он о нем думает.

Некоторые художники пошли еще дальше и просто поместили позади себя зеркало, чтобы во время написания картины можно было повернуться и оценить результат. Для проверки своих картин я часто использую черное зеркало (или просто экран выключенного смартфона), прижав его ко лбу.

В остальных творческих дисциплинах подобные методы самооценки встречаются не так часто, а в программировании они практически отсутствуют. Вообще, я нахожу, что при написании кода программисты придерживаются

принципа «и так сойдет». Это когда программист кромсает кусок кода, нагромождая строки, благодаря чему код с трудом компилируется, а затем объявляет, что его работа завершена, и уходит. Правда в том, что после остается еще гора работы, начиная с чистки кода, выполнения проверок качества, добавления инвариантов и утверждений, написания тестов, написания документации и подтверждения работы кода в широком контексте всей системы. Но нет, программисты часто прекращают работу, если компилятор (или тестировщик) работает без ошибок.

Эта книга научит вас осуществлять собственные проверки, похожие на те, что применяют художники. Это способ взглянуть на код, отделенный от истории его написания, и все дело будет в контрольных списках. Вы ниспровергаете воспоминания о своей работе, следуя порядку проверок, выявляя ошибки в том, что вы написали. Я не обучу вас всему, что касается качества, но научу самостоятельно обнаруживать как можно больше ошибок, а также отслеживать, какие из них вы продолжаете делать, чтобы избежать их в будущем. После этого вы привлечете других людей к проверке вашего кода, а сами станете проверять чужой код, смотря на него свежим взглядом и обнаруживая еще больше недостатков.

Философия сокращения дефектов сводится к проблеме вероятности. Вы никогда не устраните все дефекты – вместо этого вы будете работать над снижением вероятности того, что дефект присутствует, и сможете дать приблизительную оценку этой вероятности. Так вы избавитесь от волнения, вызванного неопределенностью насчет исправности кода, и поймете, почему производите плохое программное обеспечение. Вместо того чтобы все время довольствоваться принципом «и так сойдет», вы научитесь понимать, когда работа действительно завершена и готова к оценке. Вместо того чтобы постоянно беспокоиться обо всех невозможных крайних случаях, вы сможете оценить их вероятность и разобраться с наиболее вероятными.

## Задача упражнения

В этом упражнении вам нужно найти код, написанный вами несколько месяцев назад, и сделать его обзор. Возможно, вы пока не знаете, как правильно осуществлять проверку кода, так что просто пройдитесь по нему и напишите комментарии ко всему, что вам не понравится. Секрет в том, чтобы пройтись по коду строка за строкой и файл за файлом, ничего не пропуская. Затем пометьте фрагменты кода, которые считаете неприемлемыми, и письменно обоснуйте свое мнение. Для упражнения не обязательно выбирать огромный кусок кода – просто то, что вы написали недавно.

## Практическое задание

Составьте список всех дефектов, которые вы обнаружили, и попробуйте их классифицировать. Можете использовать официальные категории, но лучше применить базовый набор: логика, тип данных и вызов. Ошибка в логике – это когда вы неправильно написали инструкцию if или цикл. Ошибка типа данных – когда использовали переменную и думаете, что это был неправильный тип. Ошибка вызова – когда неверно вызвали функцию. Это не официальные категории, но для начала вам хватит.

# Часть II

## Быстрые задания

У вас лучшая идея. Вы поразите мир! Вы станете миллиардером! Идея озарила ваш разум, она преследует вас во сне, как Юрэй[2]. Следующий шаг – воплотить ее в жизнь, сделать явью, реализовать, изъяв из своего мозга и поместив в компьютер. Вы должны убить призрак, вернуть Юрэя из потустороннего мира и привязать его к тотему под названием Python, бросив в океан интернета.

Как по вашему, достаточно креативно?

Враг креативности – начало. Как можно воплощать свои мечты в жизнь, если сразу после старта на вашем пути оказывается полоса препятствий из процедур настройки и мертвых коров? Что, если ваша идея настолько велика и требует стольких сил, что вы начинаете волноваться? Достаточно ли вы хороши для этого? Достаточно ли вы умны? Выйдет ли из себя тот всем известный программист, который постоянно на вас кричит, приказывая написать тесты, когда увидит, что вы не знаете, как это сделать? Начало работы

[2] Мятежный призрак в японской мифологии. – *Прим. ред.*

– в принципе одна из самых сложных вещей в творчестве, и эта часть книги предназначена для облегчения данного этапа.

Я – художник, музыкант, писатель и программист, так что знаю кое-что о креативности. Еще больше я знаю о том, как нужно начинать, а также о процессе. Процесс – это то, что заставляет меня делать муторную работу над проектом, когда он меня уже не интересует. Но вы не доберетесь до этой работы, если не сможете начать.

Для этого нужна отвага и немного безразличия к тому, что думают остальные. Когда я не мог приступить к работе над картиной, я просто выбирал краску случайного цвета и наносил ее на холст примерно в нужном месте. Так поступают многие уже состоявшиеся художники. Другие же на первом этапе проводят исследование – изучают, проверяют, делают наброски, а затем сводят все это вместе и начинают работу. Собираясь написать произведение, я хожу по квартире и лихорадочно разговариваю сам с собой, представляя, будто говорю с реальным собеседником, а когда надоедает – сажусь и пишу. Просто записываю первое, что приходит в голову.

Поначалу я не беспокоюсь о грамматике, не спрашиваю себя «Звучит ли это убедительно?», просто пишу так же, как разговариваю, выплескивая все на клавиатуру. После того, как я сочиню несколько абзацев, я их оцениваю. Есть ли в написанном мной смысл? Нужно ли отредактировать? Так я постоянно нахожусь в процессе. Возможно, то, что я сочиняю, – полный бред, но главное, что я смог начать. Тогда я рассчитываю на свое умение превращать отправную точку в полностью готовый текст.

Как же вам научиться креативному старту? Вам самим, друзья мои, и предстоит это выяснить, а моя книга должна помочь. В первую очередь нужно избавиться от страха начинаний. Возможно, это и не страх вовсе. Может, у вас просто есть тонна совершенно бессмысленных заданий, которые нужно сделать, прежде чем приступать к написанию кода, и вы должны расчистить себе путь.

## Как развивать креативность

В этой части книги вы будете практиковаться в креативности, заставляя себя начинать прямо с места. Я предложу вам небольшие и простые, но невероятно скучные проекты. Например, команда `cat` в Unix просто «выплевывает» файл – в простейшем случае это две строки кода на Python. Здесь первоочередную важность играет начало, так что вы будете безжалостно продвигаться

дальше. Вы сядете за компьютер и спрыгнете с обрыва, чтобы добиться своего. Не через полчаса. Прямо сейчас.

Как именно это сделать? Вам понадобится контрольный список и автоматизация. Контрольный список содержит все, что вам нужно, чтобы начать. Включите компьютер, выйдите из социальных сетей, запустите редактор, потрогайте счастливую резиновую уточку, помолитесь своему божеству, помедитируйте 10 минут, а затем приступайте к работе над проектом. Это один из примеров, но чем короче будет ваш список, тем лучше.

Вы еще не знаете, что будет в том списке. Вероятно, у вас есть определенные мысли, но действительно ли вам известно обо всех вещах, которые нужно будет сделать перед началом работы? Вот на что вы должны обращать внимание при выполнении каждого из этих проектов. В случае с первым проектом просто сядьте и попытайтесь его выполнить, но при этом запишите все, что вы делали. Нельзя управлять тем, что нельзя измерить, и это ваш первый шаг к измерению самого себя, к пониманию того, как вы действуете. Если у вас установлено программное обеспечение для записи с экрана, так даже лучше. Запустите его и снимайте, как вы пытаетесь оживить ужасный код, а затем просмотрите получившееся видео. Запишите, что конкретно вы сделали.

Чтобы убедиться, что вместо работы над началом вы не горбатитесь над проектом впустую, каждый раз ставьте таймер. Вы должны достичь лучшего возможного результата за 45 минут. Не больше, не меньше. Запустите 45-минутный таймер, когда будете приступать, приготовите блокнот и ручку, и начинайте. Как только таймер обнулится, остановитесь. Взгляните на проделанную работу и приступайте к приятной части.

По окончании каждого проекта просмотрите контрольный список и подумайте, что можно сделать, чтобы устранить трение. Вы вручную создаете множество небольших файлов, которые стоило поискать в интернете? Создайте каркас проекта. У вас проблемы с вводом команд в текстовый редактор? Научитесь лучше им пользоваться или изучите метод слепого набора. Зависаете, не в силах вспомнить базовые команды и API? Раздобудьте книги и подучитесь, друг мой.

Затем сотрите свой код и начните заново. С чистого листа. Переверните страницу в блокноте и начните писать или включите записывающее оборудование – что угодно, лишь бы иметь возможность отслеживать свои действия. Зашли ли вы дальше на этот раз? Стало ли меньше трения? Ваша цель – уменьшать количество времени между появлением идеи и ее воплощением до тех пор, пока начало работы не станет для вас таким же обычным делом, как дыхание или потребление пищи. В конце концов, этот процесс станет совершенно естественным, и вы сможете приступить к следующему проекту.

Помните, вы должны сесть и приступить к написанию кода немедленно. Просто начните. Если ваш внутренний голос говорит вам, что так делать неправильно, прикажите этому глупому голосу закрыть свой педантичный рот. Вот в чем секрет. Расслабьтесь и пишите, представляя, что просто рассказываете этот код другу, который знает, что вы слегка ненормальный, но все равно интересный человек. О педантичных вещах вроде тестирования и качества речь пойдет позже, а сейчас просто пишите код. Устраивайте беспорядок, но пишите. Это весело. Реализовать идею куда важнее, чем победить в воображаемом конкурсе качества кода.

После каждой 45-минутной небрежно выполненной работы внимательно просмотрите то, что сделали. Это принцип «создавай, затем оценивай», и в будущем он поможет вам совершенствоваться.

## Для начинающих программистов

Если вы новичок и еще теряетесь, когда дело доходит до начала проекта, я поделюсь с вами сокращенной инструкцией. Это упражнение рассчитано на 45 минут, но как начинающему программисту вам может понадобиться немного больше времени – или вы не будете знать, откуда начинать. В таком случае, можете смело тратить 60 минут или делать два подхода по 45 минут.

Для обеспечения должного движения перед началом работы (перед запуском таймера) начинающий программист должен сделать следующее.

1. Подготовить компьютер и подготовиться самому.

2. Прочесть текст задания и сделать заметки. Это фаза исследования, вам нужно собрать как можно больше информации и зафиксировать ее в письменной форме.

3. На основе сделанного исследования составьте список дел, необходимых для выполнения упражнения. Запишите все, что вам нужно сделать в задании. Какие файлы нужно создать? Какие каталоги? Какие у них будут особенности? Какими библиотеками вы воспользуетесь?

Как только составите список дел, считайте, что вы готовы запустить таймер. Во время выполнения упражнения действуйте следующим образом.

1. Выберите самое простое задание в списке дел и займитесь им. Вам нужен файл? Создайте его! Нужен каталог? Создайте его.

2. Убедитесь, что все работает как надо.

3. Вычеркните это задание и переходите к следующему.

Я отношусь к этой технике серьезно. Это упрощенная версия той, которую использую я, и она работает. Практически каждое действие сводится к принципу «составь список, выполни каждый пункт, проверь результат». Если это помогает мне, значит, оно поможет и вам. Так что придерживайтесь моих рекомендаций, если не знаете, что делать.

## Кодинг начинающего программиста

Все, сказанное выше, также может быть применено к написанию кода. Я освещал это в «Легком способе выучить Python 3». Если вы не уверены, как написать фрагмент кода, выполняйте следующую последовательность шагов.

1. Опишите, что должен осуществлять ваш код, на разговорном языке. Если вам легче записать это сплошным абзацем, тогда так и делайте, но лучше представить все в виде списка с заданиями. Из абзаца затем можно сделать список.

2. Превратите этот список в комментарии, добавив перед каждой строкой символ #.

3. Начиная сверху, пройдитесь по комментариям и под каждым из них напишите соответствующий код Python. Если комментарий слишком неопределенный, тогда разбейте его на несколько более коротких и повторите этот шаг.

4. Запустите код, чтобы убедиться, что вы не допустили синтаксических ошибок.

Вот и все, что вам необходимо сделать. Если вы можете выразить назначение своего кода на русском, английском или другом языке, значит, вы легко сможете его реализовать и вам не придется думать на языке кода. В итоге вам не нужно будет начинать с написания комментариев, но я по-прежнему делаю так, если оказываюсь в тупике.

# Аргументы командной строки

Прежде чем вы сможете приступить к работе над этой частью книги, вам нужно выполнить несколько упражнений, которые помогут освоить аргументы командной строки в Python.

Обычно мы называем такие упражнения «шипом» (spike). Термин происходит от выполнения небольшого тестового проекта, который охватывает все элементы более крупного процесса или проекта. Эти небольшие тестовые упражнения «пронизывают» все, что вам понадобится для использования аргументов командной строки. Цель «шипа» – выяснить, как пользоваться какой-то новой библиотекой или инструментом на генеральной репетиции, прежде чем садиться и действительно использовать их в своем проекте.

Это также первое упражнение в режиме повышенной сложности. Повышенный уровень сложности предполагает, что вы сами разберетесь, как нужно что-то делать, а потом можете пойти и проверить, как это сделал я. Я не предоставлю вам код, который можно перепечатать. Не-а, это для новичков, а вы больше не новичок. Теперь вы – человек, который читает текст задачи, а затем самостоятельно ее решает.

---

**Внимание!** Обязательно прочтите это! Никто не ждет, что вы напишете полностью рабочий и готовый к публикации фрагмент программного обеспечения за 45 минут. 45-минутное ограничение установлено затем, чтобы вы приступили к работе и не беспокоились, что делаете что-то неправильно. Это толчок, чтобы сдвинуть вас с места, а не испытание. Поэтому если вы смотрите на временные рамки и застываете на месте, предполагая, что не успеете создать монументальное произведение искусства, значит, вы подошли к заданию не с той стороны. Вы должны рассматривать это как «Давайте-ка посмотрим, что я смогу сделать за 45 минут». У этих упражнений нет окончания, поскольку разные люди за одинаковый промежуток времени выполняют разные объемы работы. Используйте ограничение, чтобы понять то, как вы работаете, а не то, ужасный вы программист или отличный.

---

# Задача упражнения

Вы напишете два крошечных сценария Python, которые будут проверять наши аргументы (параметры) командной строки, используя два метода.

1. Старый добрый `sys.argv`, которым вы пользуетесь обычно.

2. Пакет `argparse` Python – для более изящной обработки аргументов.

Ваш тестовый сценарий должен обрабатывать следующие ситуации.

1. Получение помощи при использовании `-help` или `-h`.

2. Как минимум три аргумента, являющиеся флагами, – то есть они не принимают дополнительный аргумент, но использование их в командной строке включает что-либо.

3. Как минимум три аргумента, являющиеся опциями, – то есть они принимают аргумент и устанавливают переменную в вашем сценарии, равной этой опции.

4. Дополнительные «позиционные» аргументы, являющиеся списками файлов в конце всех аргументов типа –, и способные обрабатывать подстановочные знаки Терминала, такие как `*/.txt`.

Так как это упражнение – «шип», имейте в виду, что если процесс для вас слишком мучителен, можете все бросить и перейти к другой задаче. Сначала попытайтесь найти решение с помощью `sys.argv`, а если ничего не получится, попробуйте `argparse`.

Помните, что выполнение ограничено 45 минутами – учитывайте это. Также вы должны отслеживать все, что делаете, когда приступаете к работе. Ваша цель в этом упражнении – понять, как вы сами становитесь у себя на пути в начале проекта. Вы отговариваете себя еще до начала? Не знаете, где находится текстовый редактор или как им пользоваться? Запишите проблему, а затем определите, как от нее избавиться.

Впрочем, ограничение по времени не подразумевает неудачу. Вы пытаетесь что-то сделать. Что угодно за 45 минут. Если ваш уровень навыков таков, что вы успеваете лишь создать файл `ex4.py`, и ничего больше, вы все равно сделали кое-что за 45 минут. Затем вам нужно понять, почему это было все, что вы сделали, и выяснить, что вам нужно предпринять в следующий раз, а затем сделать еще одну 45-минутную попытку.

# Решение

Чтобы вы не жульничали, код ко всем решениям представлен на сайте проекта этой книги по адресу **bit.ly/lmpthwsolve**, ведущему на github.com. Вместо того чтобы указывать код прямо здесь и вызывать у вас соблазн просто заглянуть в решение, я требую от вас открыть сайт и перейти в каталог `ex4`, где показано, как я решил задачу. Там я также оставил записки о том, как я смог приступить к работе и что смог улучшить в этом компоненте.

---

**Внимание!** Помните, если вы оказались в тупике, перечитайте вступление к части II и воспользуйтесь моим советом для начинающих программистов. Составьте список, выполните каждый пункт и проверьте результат. Вот и все.

---

# Практические задания

1. Сколько еще есть библиотек Python для семантической обработки аргументов командной строки? Какие из них вам нравятся больше?

2. В чем главное преимущество `argparse` перед `sys.argv`?

3. Как вы можете улучшить свое начало работы? Можете ли вы уже исключить что-либо из этого процесса?

УПРАЖНЕНИЕ 5

# Команда `cat`

В упражнении 4 вы начали поиск того, что мешает вам работать. Это было простое упражнение, где вы исследовали, как лучше получить аргументы командной строки от пользователя. Истинным предназначением упражнения было составление «лабораторных» заметок о том, что вы делаете при начале работы. Обнаружили ли вы что-либо, требующее изменения? Какие-то странные привычки или проблемы с подготовкой? В этом упражнении вы создадите копию простой команды под названием `cat`, но вашей настоящей целью будет выбор одной вещи, изменение которой позволит вам работать быстрее. Помните, ваше выполнение `cat` не является первостепенным. Первостепенным является то, как быстро вы сможете начать и сумеете ли сделать что-то полезное за 45 минут.

Как и в предыдущем упражнении, придерживайтесь 45-минутного лимита. Ограничение времени, затрачиваемого на упражнение, – полезная методика, позволяющая войти в режим кодинга. Вообще, если вы начнете проводить ежедневную 45-минутную разминку, это будет идеальным решением для продвижения дальше. Прежде чем вы сможете это делать, вам нужно стать лучше при начале работы, так что решите, какое препятствие вы устраните сегодня, и давайте приступать.

---

**Внимание!** Я упомяну это еще раз, чтобы было понятно: в этом упражнении вы не можете показать плохой результат. Если вы рассматриваете 45-минутное ограничение как вариант оценивания и устанавливаете какие-то ожидания насчет того, как хорошо вы обязаны справиться, тогда ваш подход неверен. Думайте о 45 минутах как о приспособлении, дающем вам пинок и заставляющем продолжать. Это не тест. Повторяю, это не тест. Говорите это про себя, пока не расслабитесь и просто не начнете делать дело.

---

## Задача упражнения

Команда `cat` – сокращение от concatenate («сцепить») – наиболее часто используется для вывода содержимого файла на экран. Используется она таким образом.

```
cat файл.txt
```

Эта команда разобьет содержимое файла `файл.txt`. Это не совсем ее настоящее предназначение. Сперва она использовалась для объединения нескольких файлов – потому и была названа `cat`. Чтобы осуществить это, просто добавьте каждый файл к `cat`:

```
cat A.txt Б.txt В.txt
```

Команда `cat` пройдется по каждому файлу, выпишет все их содержимое и прекратит выполнение после последнего файла. Проблема в том, что таким образом файлы не объединяются. Для этого вам нужно воспользоваться перенаправлением файлов POSIX:

```
cat A.txt Б.txt В.txt > Г.txt
```

Символ > должен быть вам знаком, в противном случае советую освежить в памяти базовую операцию оболочки Unix. Она просто берет стандартный вывод команды `cat` (что в нашем случае является объединенным содержимым файлов `A.txt`, `Б.txt` и `В.txt`) и записывает его в файл `Г.txt` в правой части.

Выполните команду `cat` как можно больше раз, как можно быстрее, используя то, чему научились в упражнении 4. Помните, чтобы осуществить обычный вывод, просто используйте функцию `print` в Python. Чтобы узнать больше о `cat`, воспользуйтесь командой `man`:

```
man cat
```

Это была инструкция относительно команды `cat`. Вы получите бонусные очки за то, что выполните ее как можно больше раз в течение 45 минут.

# Решение

Мое решение этого задания можно найти в репозитории проекта по адресу **bit.ly/lmpthwsolve** на GitHub. Оно находится в `ex5/`, где вы увидите, что я выбрал решение на скорую руку. Если, приступая к этому упражнению, вы беспокоились о качестве или креативности, вы были заняты не тем. Ваше дело – действовать небрежно, действовать быстро и сделать это. Смысл ограничения по времени – избавить вас от мнения, что каждый раз, когда вы

прикасаетесь к клавиатуре, вы обязаны производить на свет золотого теленка, которому бы все поклонялись. Делайте все как можно быстрее, а после завершения можете проанализировать работу и подумать, что можно усовершенствовать.

## Практические задания

1. Столкнулись ли вы с какими-либо неожиданными возможностями команды `cat`, которые вы ни разу не использовали или которые вызвали трудности?

2. Смогли ли вы устранить одно из препятствий (трений) в процессе начала работы? Это важнее команды `cat`, так что если вам не удалось избежать трения, нужно выполнить это упражнение повторно.

3. Определили ли вы еще больше преград, стоящих на вашем пути? Элементарные вещи, вроде боли в шее по причине того, что вы сидите слишком низко. У вас не развит навык слепого набора? Как насчет вашего настроения? В чем заключались вещи, что вам препятствовали? Можете ли вы перестать думать о них?

## Дальнейшее обучение

Эта книга не о самопомощи, и я не собираюсь исправлять вашу психику, но нахожу, что значительным препятствием на пути изучения чего-то нового является не сам предмет изучения, а ваши страхи. При выполнении этого упражнения вы поняли, что вредоносные мысли или страхи не давали вам начать, а затем, я полагаю, вы стали тратить 10 минут на исследование того, как вы себя чувствуете, прежде чем приступать к 45-минутному заданию. Записывание своих страхов, тревог и ощущений прояснит их и поможет осознать, как это непрактично – волноваться о подобных вещах при выполнении такого крошечного упражнения. Опробуйте этот способ. Вы будете поражены, что 10 минут записывания своих ощущений сделают с вашими ощущениями.

# Команда find

К счастью, вы обнаруживаете различные способы помешать самим себе еще до того, как приступаете к работе. Возможно, тогда это не так драматично, но вы должны по меньшей мере определять вещи из своего окружения, которые усложняют вам процесс начала. Эти небольшие упражнения полезны для развития умения фокусироваться на начале, поскольку они не имеют большой важности и длятся достаточно для того, чтобы вам было удобно анализировать себя. Если бы эти проекты были рассчитаны на часы работы, вам надоело бы изучать свои действия и производить улучшения. Короткий 45-минутный проект – как раз то, о чем можно сделать записи в блокноте или с помощью программного обеспечения, и очень быстро их просмотреть.

Это шаблон, который я использую во время обучения. Я определяю нечто, над чем мне необходимо поработать, например то, как я подхожу к началу работы или то, как я владею инструментом. Затем я разрабатываю упражнение, которое просто-напросто направлено на улучшение этого аспекта. Когда я учился живописи, я испытывал трудности с тем, чтобы выйти из дому и отправиться рисовать деревья. Я сел и проанализировал проблему, и первое, что обнаружил, – я тащил с собой слишком много вещей. Кроме того, мои вещи хранились в случайных местах квартиры. Я заказал специальную сумку только для своих принадлежностей для рисования и держал ее наготове. Когда я хотел порисовать на улице, я хватал эту сумку и направлялся в одно или несколько мест, не занимаясь планированием масштабных походов. Я практиковался только в том, как хватаю свою сумку, направляюсь в одно или несколько мест, там подготавливаюсь, рисую и возвращаюсь домой; практиковался, пока весь процесс не стал гладким, как шелк. Потом я смотрел программы Боба Росса[3], чтобы понять, как правильно рисовать деревья.

Вот чем вы должны заниматься. Единственное место, где люди впустую теряют время и усилия, – это их рабочее место. У вас есть специально выделенное место для работы, которое остается неизменным? Я оставил свой ноутбук и теперь пользуюсь настольным компьютером, имея постоянное рабочее место. Это также спасло мою спину и шею от таскания повсюду этого куска металла и позволило работать с большим экраном. В этом упражнении я хочу, чтобы вы обратили внимание на свою рабочую среду и, прежде чем начать, убедились, что она вам подходит.

1. Достаточно ли вам света? Не слишком ли его много?

[3] Американский художник и популяризатор живописи, автор серии телепередач. – *Прим. ред.*

2. Удобный ли у вас стул? Может, нужна клавиатура получше?

3. Есть ли проблемы с другими инструментами? Вы пытаетесь сделать на Windows то, что делается в Unix? Пытаетесь сделать на Linux то, что делается на Mac? Не спешите покупать новый компьютер, но если у вас возникает слишком много трудностей с тем, что вы хотите осуществить, рассмотрите и такой вариант при случае.

4. Что со столом? Он у вас вообще есть? Вы занимаетесь программированием, целый день сидя в кафе с ужасными стульями и выпивая слишком много кофе?

5. Как насчет музыки? Вы слушаете музыку со словами? Если я слушаю музыку без слов, мне проще сосредоточиться на внутреннем голосе, помогающем писать код.

6. Вы работаете в офисе с открытой планировкой и ваши коллеги вас раздражают? Купите себе большие наушники, целиком закрывающие уши. Когда наденете их, перестанете обращать внимание на других людей, и они оставят вас в покое, сочтя, к тому же, что это более вежливо с вашей стороны, чем пользоваться незаметными наушниками. Это также поможет перестать отвлекаться и оставаться сосредоточенным.

В процессе выполнения этого упражнения подумайте о подобных вещах и попытайтесь упростить и улучшить свою рабочую среду. Впрочем, вот еще кое-что: не нужно скупать разные сумасшедшие штуковины и тратить уйму денег. Просто определите проблемы и попытайтесь найти пути их решения.

## Задача упражнения

В этой задаче вы реализуете базовую версию инструмента (утилиты) `find`, предназначенного для поиска файлов. Делается это следующим образом:

```
find . -name "*.txt" -print
```

Так вы отыщете в текущем каталоге все файлы, заканчивающиеся на `.txt`, и выведете их на экран. У `find` очень много аргументов командной строки, так что вам не нужно использовать их все за один 45-минутный сеанс. Общий формат поиска следующий.

1. Каталог, в котором нужно начать поиск: `.` или `/usr/local/`

2. Аргумент, вроде `-name` или `-type d` (файлы типа «каталог»)

3. Действие, которое следует применить ко всем найденным файлам: `-print`

Вы также можете осуществлять полезные манипуляции, например выполнять команду над каждым найденным файлом. Если вы хотите удалить все файлы Ruby в своем домашнем каталоге, сделайте так:

```
find . -name "*.rb" -exec rm {} \;
```

Пожалуйста, не выполняйте эту команду, если не сознаете, что она удалит все файлы, оканчивающиеся на `.rb`. Аргумент `-exec` принимает команду, заменяет каждую комбинацию `{}` именем файла, прекращает читать команду, когда доходит до `;` (точки с запятой). В предыдущей команде мы используем `\;`, поскольку в `bash` и многих других оболочках `;` является составляющей встроенного языка, так что этот символ нужно экранировать.

На самом деле это упражнение оценит ваше умение использовать `argparse` или `sys.argv`. Советую вам выполнить команду `man find`, чтобы получить список аргументов, а затем использовать `find`, чтобы выяснить, какие аргументы выполнять. У вас есть только 45 минут, так что, вероятно, много вы сделать не успеете, но обратите внимание на `-name` и `-type`, а также на `-print` и `-exec`. Правда, с аргументом `-exec` будет посложнее, так что оставьте его напоследок.

Когда вы закончите с этим, попытайтесь найти библиотеки, которые могут сделать работу за вас. Обратите внимание на модули `subprocess` и `glob`, а также на `os`.

## Практические задания

1. Сколько раз вы воспользовались `find`?

2. Какие библиотеки вы нашли, чтобы усовершенствовать выполнение?

3. Учитывали ли вы поиск библиотек как часть своего 45-минутного сеанса? Вы могли бы сказать, что исследование, проведенное до

начала упражнения, не считается, и я не возражал бы. Если вы хотите повысить уровень сложности, включите это исследование в свои 45 минут.

## Дальнейшее обучение

Сколько раз вы способны выполнить команду `find` при следующих 45-минутных сеансах? Можете сделать это своим разминочным упражнением на следующей неделе и посмотреть, что из этого выйдет. Помните, вы должны пытаться состряпать лучшее уродливое творение. Не волнуйтесь, я не скажу приверженцам Agile, что вы просто развлекаетесь.

# Команда `grep`

Команда `find` должна была показаться вам сложной, но выполнимой задачей для 45-минутного отрезка времени. На этом этапе вы уже должны устранить столько преград на пути к началу работы, сколько можете себе представить. Вам может показаться, что когда вы что-то устраняете, это ухудшает ваши навыки. К примеру, раньше перед началом рабочего дня я ходил пить кофе. Это занимало около получаса и приносило массу удовольствия, но 30 минут зачастую перерастали в несколько часов. Я решил положить этому конец, но тогда начал страдать на работе. Как оказалось, мне по-прежнему был необходим кофе, так что я купил отличную кофемашину и научился делать собственный латте. Теперь я просыпаюсь, иду делать себе латте, а затем немного рисую, чтобы войти в «творческий» режим.

Не все, что вы делаете, на самом деле неэффективно, так что будьте осторожны и не избавляйтесь от чего-то просто потому, что оно занимает время. Существуют небольшие ритуалы и личные привычки, которые помогают запустить мыслительные процессы – фокус в том, чтобы не убрать их, но упростить.

В данной части книги вы также должны освоить концепцию управления временем. Установление ограничения в 45 минут даст вам ясно понять, что вы не знаете, сколько времени вам необходимо для выполнения какой-либо задачи. Имея в распоряжении всего 45 минут, вы не можете потратить 30 из них на то, чтобы приводить в порядок окна `vim` и организовывать идеальную структуру каталогов, а затем создавать совершенно новый алгоритм для сортировки. Нужно бережливо относиться к тому, что вы выполняете и в каком порядке решаете задачи.

Хороший способ реализовать проект – начать с чего-то простого, с чем можно справиться в первую очередь. В случае с командой `find` это было, вероятно, получение файла с помощью шаблона `glob`. Человек, не владеющий навыками тайм-менеджмента, немедленно попытался бы использовать аргумент `-exec`, чтобы показать, какой он замечательный программист, но ведь `-exec` не работает без `-name`, и его куда сложнее выполнить. Лучший способ принять решение – помнить, что вам нужно нечто, что вы затем сможете использовать. Если 45 минут уже прошли и вы можете использовать `-exec`, но не можете помещать туда файлы, как вы собираетесь это делать? А если пройдет то же количество времени, но у вас появится возможность составить список файлов с совпадающим именем, тогда вы справились. За 45 минут вы получили результат.

Продолжайте работать над своим списком препятствий и оценивать свой старт, но теперь обращайте внимание на тайм-менеджмент. Планируйте, что вы собираетесь предпринимать, чтобы в случае неудачи у вас был запасной вариант. Это не должны быть полноценные стратегии, но два работающих варианта лучше, чем 10, которые вообще не работают из-за того, что вы позабыли о простейшей необходимой для них вещи. Или еще хуже – 10 неработающих из-за того, что вы бросали осуществление каждого из них на полпути.

## Задача упражнения

Теперь вы выполните команду `grep`. Как обычно, вам нужно будет прочесть справку по `grep` и поиграть с полученными данными. Предназначение `grep` – находить в файлах текстовые шаблоны при помощи регулярных выражений. Вы выполняли `find`, используя модуль `glob`; эта операция аналогична, но осуществляется внутри файла, а не внутри каталога. Например, если мне нужно найти слово help в своей книге, я делаю так:

```
grep help *.rst
```

Аргументы командной строки `grep` довольно просты. Сложная часть – это регулярные выражения, так что воспользуйтесь модулем `re`. Этот модуль позволит загрузить содержимое файла, а затем осуществить в нем поиск по заданному шаблону. Еще подсказка – скорее всего, нужно будет загрузить весь файл при помощи метода `readlines`, а не `read`. Большинство опций `grep` лучше работают таким образом, несмотря на то, что это менее эффективно.

Кроме того, можете перескочить к упражнению 31 и ознакомиться с введением в регулярные выражения.

## Практические задания

1. Есть ли у модуля `re` какие-либо особые опции, которые делают его похожим на `grep`?

2. Могли бы вы реализовать работу с `grep` в модуле, который затем использовали бы в утилите `find` для добавления возможности `grep`?

## Дальнейшее обучение

Модуль `re` крайне важен, так что уделите достаточное количество времени изучению всего, что с ним связано. Мы будем использовать его, а также регулярные выражения в другой части книги.

УПРАЖНЕНИЕ 8

# Команда `cut`

Надеюсь, вы продолжаете изучать Python, но еще больше самих себя и то, как вы работаете. В этой части книги вы узнаете кое-что о движении, а также о креативности, учась совершенствовать движение. Вы не достигнете креативности, не устранив препятствия на старте, но вы также должны сознавать, что простейший способ усовершенствовать личное движение – это наблюдать за своей работой. Простого выполнения упражнений недостаточно. Вы должны взглянуть на свой способ работы и попытаться улучшить его.

По ходу совершенствования своего движения вы можете обнаружить, что, занимаясь разными проектами, вам нужно по-разному подходить к началу работы. Когда я работаю над программным обеспечением, похожим на эти инструменты командной строки, я могу начать непосредственно с написания кода. Когда я работаю над проектом с графическим интерфейсом пользователя, мне нужно нарисовать пользовательский интерфейс, реализовать его ненастоящую версию, а затем заставить все работать. Далее в книге вы научитесь обоим способам и попрактикуете их.

В данном упражнении от вас требуется сосредоточиться на своем физическом здоровье и поведении. В процессе работы программисты зачастую не заботятся о собственном организме. Эта работа, кажется, и не должна причинять вам никакого вреда. Вы просто целый день сидите за столом, не пилите деревья и не охотитесь на животных в городских условиях. Правда в том, что любая работа, при которой вы на протяжении долгого периода времени сидите на одном месте и делаете что-то, вызывающее стресс, может разрушить ваш организм. Чтобы не допустить этого, в процессе работы следите за следующими вещами.

1. Хорошая ли у вас осанка? Сидеть с идеально ровной спиной не совсем правильно, но не стоит и горбиться. Нужно, чтобы тело было выпрямлено и расслаблено, голова поднята.

2. Вы подтягиваете плечи к самым ушам? Опустите их.

3. Ваши запястья напряжены и вы кладете их на стол для отдыха? Попытайтесь сделать так, чтобы они парили над клавиатурой – не слишком свободно и не слишком зажато.

4. Ваша голова расслаблена и вы смотрите прямо перед собой или напряженно следите за другим монитором сбоку?

5. Удобно ли кресло?

6. Вы делаете перерывы? 45 минут – это максимальный промежуток времени, в течение которого можно работать без перерывов.

7. Вы отлучаетесь в туалет? Я серьезно. Если нужно, встаньте и сходите. Гораздо хуже сидеть и держать все в себе.

Есть и другие моменты, но это наиболее важные. По-моему, многие программисты уверены, что, если они отойдут от своего компьютера, он взорвется. Компьютер будет терпеливо ожидать вашего возвращения, а перерывы помогут подойти к решению проблемы с другой стороны.

Также подумайте над тем, чтобы включить веб-камеру и вести запись того, как вы работаете. Вам может казаться, что вы не сутулитесь, но затем, в разгаре «сражения с кодом», вы возможно сделаете со своим телом что-то странное, не отдавая себе в этом отчет. Во время данного упражнения запишите свои действия, а затем отыщите все, что создает напряжение, проблемы, вызывает боль в спине или просто мешает.

## Задача упражнения

В этом упражнении вы будете использовать инструмент `cut`. Мне очень нравится `cut`, благодаря этой утилите я кажусь чародеем Unix, в то время как все, что она осуществляет, это извлечение (вырезание) потоков текста. Самый простой инструмент для обработки текста, который только можно придумать, чтобы он еще приносил пользу. Для работы с `cut` вам понадобится еще один инструмент для «скармливания» строк текста, поэтому можем сделать так:

```
ls -l | cut -d ' ' -f 5-7
```

Здесь вы можете получить тарабарщину, но на большинстве систем должны вывестись владелец и группа каждого файла. Команда `cut` принимает опции, устанавливающие тип разделителя (`-d ' '` для символа пробела), а следом список полей, которые нужно извлечь (в нашем случае 5–7). Мы используем команду `ls -l`, чтобы предоставить данные для вырезания.

С этим все, так что прочтите страницу man команды `cut` и посмотрите, сколько раз вы сможете осуществить выполнение, не забывая о собственном теле.

## Практическое задание

Какое влияние на выполнение оказывает Юникод?

## Дальнейшее обучение

Помните, тело – ваша неотъемлемая часть, и мысль о том, что единственная важная вещь – это разум, в корне неверна. Если вы будете относиться к своему телу как к бесполезной рухляди, ваш мозг станет работать менее эффективно, что не позволит комфортно выполнять эту работу на протяжении долгого времени. Советую как можно чаще заниматься физическими оздоровительными упражнениями – это могут быть йога, танцы, пешие прогулки, походы или посещение спортзала. Что угодно, что поддерживает ваше тело в хорошем состоянии и позволяет мозгу работать без помех.

Смотрите на это таким образом: если вы постоянно ощущаете дискомфорт, боль или слабость от плохого обращения со своим телом, значит, вашему мозгу приходится тратить силы на отслеживание проблем и оповещение вас. Если же тело станет как хорошо смазанный ухоженный механизм, тогда и мозгу будет не о чем беспокоиться.

Наконец, если по каким-либо причинам ваше тело не функционирует обычным образом, просто делайте, что можете. Никто не говорит, что вам необходимо иметь тело как у меня, чтобы быть программистом. Один из замечательных аспектов программирования заключается в том, что этим может заниматься кто угодно, даже если физические возможности человека ограничены. Я хочу сказать, не позволяйте программированию усугубить ситуацию. Берегите здоровье.

# Команда sed

Эти крошечные проекты полезны для изучения самого себя, но давайте сложим все вместе и вспомним основные моменты, на которые вы обращаете внимание.

1. Процесс начала работы, включающий такие аспекты, как текстовый редактор, ваше умение набирать текст и остальные вещи, касающиеся непосредственно компьютера.

2. Психологический настрой при начале и во время работы, а также, возможно, конспектирование как способ управлять им.

3. Рабочая среда, включая стол, освещение, кресло и тип используемого компьютера.

4. Осанка и физическое здоровье для предотвращения получения ущерба от работы.

В данном упражнении мы используем этот план по совершенствованию и сделаем следующий шаг, отследив некоторые показатели. Вы брали маленький инструмент командной строки, читали о нем, определяя его возможности, и тратили лишь 45 минут на быстрое задание. Теперь вы можете перечислить свои возможности, определить приоритеты, а затем выяснить, сколько вы способны сделать за 45 минут. По сути, можете вернуться ко всем проектам, что уже выполнили, и к записям, где вы фиксировали изменения, чтобы подсчитать показатели и понять, стали ли вы лучше.

Уделите время тому, чтобы вернуться к своим заметкам и оценить, насколько полно вы использовали возможности инструментов, которые применяли в каждом 45-минутном сеансе. Постройте на бумаге соответствующий график и посмотрите, произошли ли значительные изменения, хорошие или плохие, как только вы внесли коррективы в свой способ работы. Далее в этом упражнении попытайтесь предположить, сколько у вас получится сделать, основываясь на изменениях, что вы внесете. Вы даже можете восстановить некоторые препятствия, чтобы увидеть, как это отразится на производительности.

---

**Внимание!** Имейте в виду, что это личные показатели, ими не стоит делиться с кем-либо еще. Они имеют слабое отношение к науке и

предназначены для некоторого повышения объективности в анализе того, как вы работаете. Это не универсальные данные, описывающие всех программистов, но поверьте, если ваш менеджер узнает о них, он захочет взглянуть, а потом потребует, чтобы все остальные члены вашей команды также отслеживали эти показатели, и у вас возникнут большие неприятности. Считайте, что ваши записи – это ваш личный дневник, и никому их не показывайте.

## Задача упражнения

Это упражнение сложнее предыдущих, поскольку в нем мы будем использовать больше регулярных выражений и освоим инструмент под названием `sed`. Эта утилита позволяет вносить изменения в текст с помощью шаблона замены регулярного выражения. Используя его, можно задать, что необходимо заменить в каждой полученной строке. Сложность заключается в использовании формата выражений `sed`, так что советую следовать трехуровневому подходу.

1. Уровень 1: опции командной строки для самого простого использования `sed`, которое заключается в замене одной строки другой.

2. Уровень 2: включение регулярных выражений в опциях командной строки.

3. Уровень 3: реализация формата выражений `sed`.

Примером использования `sed` может служить замена одного слова другим в потоке текста. Если бы я захотел изменить вывод `ls` так, чтобы мое имя было заменено словом author, то осуществил бы это следующим образом:

```
ls -l | sed -e "s/zedshaw/author/g"
```

Однако мощь `sed` – в создании регулярных выражений для поиска совпадений и последующей их замены. Если вы пользуетесь редактором `Vim`, значит, уже знакомы с таким синтаксисом:

```
ls -l | sed -e "s/Jul [0-9][0-9]/DATE/g"
```

Прочтите man-страницу `sed`, но, вероятно, для правильного выполнения вам придется провести дополнительное исследование. Предлагаю заняться им

накануне, а на следующий день осуществить 45-минутный сеанс на основе этого исследования.

## Практические задания

1. Обнаружили ли вы что-то необычное или неожиданное, когда оформляли свои показатели?

2. Каким было ваше ожидание от работы перед началом сеанса?

3. Как оно соотнеслось с тем, что вы действительно сделали?

## Дальнейшее обучение

В приложенном видео я представлю вам нечто под названием «график бегущей последовательности» (run chart). Это простой график некой деятельности, которую вы хотели бы контролировать, демонстрирующий, как она изменяется со временем. График последовательности используют для определения значительных изменений поведения, поскольку это одновременно несложный и эффективный инструмент визуализации. Далее в книге вы также будете использовать графики последовательности, так как они просты, но обладают мощным потенциалом.

УПРАЖНЕНИЕ 10

# Команда `sort`

Вы постепенно осваиваете то, что я называю практикой персонального движения (Personal Process Practice, 3P). Эта идея не нова. Цель 3P – получить объективное представление о том, как вы делаете что-либо, не убивая при этом свою креативность и производительность. Путем обычного отслеживания своих показателей и построения графиков последовательности для управления улучшениями вы можете значительно повысить качество работы. Впрочем, есть риск, что тогда вы снизите скорость выполнения упражнений, перестанете делать, что нужно, или же 3P вовсе превратится в ваше основное занятие.

На протяжении более четырех лет своей программистской карьеры я следую этой практике, и она сильно помогла мне в изучении самого себя и того, как я работаю. Она также помогла пробиться через ложь, проталкиваемую сторонниками движения. У меня был простой способ по-настоящему проверить, улучшили ли взгляды на программирование некоторых ученых мужей мою личную производительность. Я сказал бы, что единственная совершенная мной ошибка заключалась в том, что я относился к этому слишком серьезно, на протяжении четырех лет убивая собственную креативность.

Вот почему вы выстраиваете концепцию своего начального движения и рабочей среды в виде небольших быстрых заданий. Когда на всю работу отводится только 45 минут, у вас не остается времени на то, чтобы производить сложные измерения и беспокоиться о конкретном способе выполнения задания. Позже мы перейдем к заданиям, требующим концентрации, и тогда вы потратите больше времени и снимете больше показателей. По мере того как вы работаете, старайтесь не допустить, чтобы показатели убивали вашу креативность, рабочий поток или благополучие. Если вы ненавидите измерять что-то, не делайте этого. Найдите способ автоматизировать процедуру или придумайте другую систему показателей.

Для данного упражнения нужно сделать график последовательности, отображающий процентную долю реализованных вами возможностей. Это значит, прежде чем приступить к работе, вы должны точно подсчитать все возможности, указанные на страницах мануала команды `sort`, а затем отметить, сколько из них вы реализовали. Не забудьте отсортировать их, чтобы успеть сделать достаточно для хоть какой-нибудь работы инструмента. Получить результат в 90% с инструментом, который не сортирует текст, на самом деле значит получить 0%.

Когда закончите, постройте график бегущей последовательности, отображающий процент возможностей, использованных вами в каждом проекте. Мы проанализируем его в следующем упражнении.

## Задача упражнения

В данном упражнении вы будете выполнять очень простую команду `sort`. Она принимает строки текста и просто сортирует их по порядку. Однако у нее есть несколько интересных свойств, так что прочтите man-страницу команды `sort` и выясните, что она способна делать. Большинство людей используют `sort`, просто чтобы упорядочить списки имен:

```
ls | sort
```

Сортировать можно также в обратном порядке:

```
ls | sort -r
```

Вы можете изменять принцип сортировки, например игнорировать регистр:

```
ls | sort -f
```

Или даже сортировать в числовом порядке:

```
ls | sort -g
```

Так вы мало что сделаете с выводом команды `ls`, если только вы не выводите одни лишь цифры.

Ваша задача – реализовать как можно больше возможностей, отслеживая каждую из них. Записывайте результаты, чтобы позже их проанализировать.

## Практические задания

1. Достигли точки, когда вам больше нечего улучшать? Продолжайте искать, а также прислушайтесь к предложениям других людей.

2. Мы программисты, люди кода. Пытались ли вы найти код, который позволит вам повысить свою эффективность? У моих друзей Обри и Дэнни есть проект под названием «Готовый рецепт» (cookie-cutter), с которым вы обязаны ознакомиться: **cookiecutter.readthedocs.io/en/latest/**.

3. Теперь вы должны разобраться, как вычислять среднее значение (`mean`) набора чисел. Это пригодится вам для расчета на Python центральной оси вашего графика последовательности.

## Дальнейшее обучение

Если вы действительно хотите получить правильный график, вам также нужно вычислить стандартное отклонение (`standard deviation`) полученных чисел. Необязательно делать это сейчас, но если вы желаете соблюсти техническую точность, тогда это не помешает.

# Команда `uniq`

В начале этих двух последних упражнений мне уже особо нечего сказать. Вы должны понимать, как оценивать свою рабочую среду, начало работы, то, как вы сидите, и все, что влияет на вашу способность приступать к выполнению задания. Вы должны были прорваться через эту начальную фазу, используя крохотные 45-минутные проекты. Если вы это еще не освоили, надежный способ заставить себя начать работать – установить таймер на 45 минут и кричать: «ПОШЕЛ, ПОШЕЛ, ПОШЕЛ!» Цель состоит не в том, чтобы получить образцовый результат, но в том, чтобы просто сдвинуться с места.

У вас также должна быть достойная записная книжка с графиками последовательности, индицирующими, насколько хорошо работают вносимые вами улучшения. Эти графики не слишком научны, но по ним можно понять, что работает, а что – нет. Пользуясь графиками последовательности, просто ищите резкие выступы в любую сторону, а затем – соответствующую причину выступа. Если это был выступ вверх, выясните, что его спровоцировало, и примените это снова. Если это был отрицательный выступ, выясните, что его спровоцировало, и в дальнейшем избегайте этого.

Когда я говорю «выступ», я имею в виду значительные изменения. Следует сказать, что график бегущей последовательности должен колебаться. Фактически, если он не меняется после новых 45-минутных упражнений, это тоже нехорошо, и вы должны выяснить, почему так происходит. Обычно графики колеблются и скачут вокруг центральной оси, и вам просто нужно найти причины значительных выступов в любую сторону. Если в предыдущем упражнении вы занимались «Дальнейшим обучением», тогда для выявления проблем можете воспользоваться `2 * std.dev` (дважды стандартное отклонение), получив линии выше и ниже среднего значения (`mean`).

---

**Внимание!** Ознакомьтесь с видео к данному упражнению, где графики последовательности рассмотрены более подробно. Этот материал гораздо проще объясняется визуально.

---

## Задача упражнения

Команда `uniq` просто принимает список отсортированных `sort` строк и удаляет все повторения. Это очень удобно, когда вы хотите получить только

уникальные строки списка. Если вы выполняли эти команды, то сможете сделать следующее:

```
history | sed -e "s/^[ 0-9]*//g" | cut -d ' ' -f 1 | sort | uniq
```

Команда `history` выводит список всех команд, которые вы запускали. Команда `sed` принимает регулярное выражение, позволяющее «отцепить» начало от вывода `history`. Затем я использую команду `cut`, чтобы выбрать только имя команды (поскольку это первое слово). После этого я выполняю сортировку с помощью `sort`, пропускаю результат через `uniq` и получаю все запущенные команды.

Выполните достаточное количество `uniq` и любых других команд, необходимых для работы предыдущей команды. Можете изменить формат, если ваша команда `sed` пока не обрабатывает выражения, но по завершении работы с упражнением вы должны получить список команд.

# Практические задания

1. Теперь у вас есть список команд, которые вы можете начинать выполнять, если хотите заняться дальнейшим обучением.

2. Это первое упражнение с несколькими проектами, здесь вы объединяете предыдущие упражнения в одно. Узнали ли вы что-нибудь новое о своем движении?

3. Как идут дела с вашими графиками последовательности? Приносят ли они пользу?

# Дальнейшее обучение

Изучите графические библиотеки для Python и посмотрите, сможете ли создать эти графики с помощью Python. Также отслеживайте, сколько у вас уходит времени на то, чтобы приступить к работе, и подумайте, может ли график помочь вам сократить это время.

# Обзор

Первый этап моего помешательства на методике завершен – для меня, но не для вас. Сейчас мы рассмотрим стратегию этой части книги, чтобы вы могли продолжать самостоятельно пользоваться ею в будущем. Стратегия такова.

1. Вы должны работать над стартом каждого проекта.

2. Чтобы изолировать проблему, вы работаете над небольшими проектами, с которыми можно справиться за 45 минут. Это позволяет сфокусироваться на проблемной области, связанной с подходом к началу проекта, и повторять соответствующую часть своего движения.

3. Выполняя эти проекты, вы определяете возможные причины своей проблемы с началом работы. Она может скрываться в компьютерных настройках, рабочей среде, психических процессах или физическом здоровье. Существуют другие источники, но эти наиболее вероятны.

4. Определив возможные причины, вы начинаете работать над их устранением в границах 45-минутных сеансов.

5. Наконец, вы записываете показатели и строите соответствующие графики, чтобы проверить, помогают ли вносимые изменения, а также чтобы убедиться, что они не вредят вашей производительности.

Чтобы приносить пользу, эта стратегия не обязательно должна иметь вид формального «научного» процесса. Вам нужно относиться к ней как к журналу учета, который помогает вам объективно оценивать то, как вы работаете. Если вы сделаете все правильно, то столкнетесь с неожиданными вещами, о которых прежде не думали. Процесс сбора данных вынуждает вас исследовать новые возможности и расширять предположения о причинах чего-либо.

Пожалуйста, помните, что этими личными показателями не следует делиться с остальными, особенно с людьми из руководства. Менеджеры неизбежно попытаются наложить эти показатели на вашу работу, и в таком случае вам придется приложить усилия, чтобы им отказать. Это ваши личные записи, и никто не имеет права их читать – так же, как дневник или личную электронную почту.

## Задача упражнения

В последнем упражнении вам нужно выбрать любимый инструмент и совершенствовать его использование посредством серии 45-минутных сеансов на протяжении недели или дольше. Учитывая все, что вы узнали о себе, возьмите этот проект, начните с чистого листа и создайте что-нибудь более основательное. По-прежнему ограничивайте себя 45 минутами, но не относитесь к этому финальному проекту, как к быстрому программистскому заданию. Наоборот, вы работаете над своим следующим шагом.

После того как я заканчиваю быстро проверять какую-нибудь идею, я либо удаляю результат, либо привожу его в порядок. Если работа настолько отвратительна, что никогда не должна увидеть света дня, я удаляю ее и начинаю заново с самого начала. Вы не забудете, что вы делали, и то, как вы это делали, но сосредоточение внимания на качестве поможет выполнить задание лучше. Если результат не так уж плох, я все подчищаю и продолжаю работу.

Один из эффективных способов превратить результат быстрого задания во что-то основательное – извлечь ключевые элементы в библиотеку. Так вы начнете рассматривать код как нечто, что будет использовано в другом коде. Я делаю это следующим образом.

1. Прохожусь по файлу и превращаю весь этот поток сознания в набор функций.

2. Применяю к коду принцип DRY («Не повторяй себя», англ. Don't Repeat Yourself), удостоверившись, что убрал повторения. Но не переусердствуйте здесь: код, полностью лишенный повторений, выглядит как зашифрованная бессмыслица.

3. Как только код приведен в порядок и работает, как и раньше, но уже с функциями, я вытаскиваю эти функции в модуль и удостоверяюсь, что исходный код продолжает работать. Помните, пока вы подчищаете код, не меняйте ничего, просто преобразовывайте и исправляйте.

4. Когда код перемещен и снова работает, я сажусь писать тесты, чтобы убедиться, что все продолжит работать, когда я действительно начну вносить изменения.

В этом упражнении вам нужно взять свой любимый проект и проделать с ним манипуляции, придающие ему «официальный характер». Занимайтесь по 45 минут за раз и выполните описанную выше процедуру. Можете работать и дольше 45 минут, но тогда обязательно делайте 15–30-минутные перерывы

между каждым сеансом. Временные рамки остаются теми же, только теперь вы не выполняете быстрые задания, все становится серьезнее.

## Практические задания

1. Сравните ваш код до и после придания ему официального характера. Нашли ли вы ошибки? Были ли другие улучшения?

2. Если изначальный и усовершенствованный код работают схожим образом, тогда действительно ли вам нужно вносить эти усовершенствования? Зачем вообще приводить код в порядок, если он отлично функционирует и, скорее всего, более прост?

3. Выберите новую команду из своего списка часто запускаемых команд (см. упражнение 11) и опробуйте на ней весь процесс. Выполните быстрое задание, затем усовершенствуйте результат.

## Дальнейшее обучение

Вот список других команд, выполнением которых следует заниматься на протяжении 45 минут:

- `ls`
- `rm`
- `rmdir`
- `mkdir`
- `cal`
- `tail`
- `yes`
- `false`

Попробуйте реализовать какие-нибудь из них.

# Часть III

## Структуры данных

Вы уже прошли определенный путь, выстраивая персональное движение, которое позволяет приступать к выполнению быстро и с минимальными усилиями. Хорошее персональное движение и развитие умения просто брать и начинать – это фундамент креативности. Творческий образ мышления предполагает текучесть и расслабленность. Если вы сталкиваетесь с препятствиями при начале работы и расстраиваетесь, становится трудно «попасть в поток». Обучение способности переключать мозг в режим креативности позволит творчески подходить к решению проблем и повысить собственную производительность.

Но если вы производите лишь мусор, то нет никого смысла быть креативным. Поначалу да, очевидно, большая часть будет мусором, но вряд ли вы хотите производить ужасное программное обеспечение всю свою жизнь. Вам нужно найти баланс между креативным подходом к выполнению заданий и скрупулезным достижением качества. Я выступаю за то, чтобы люди переключались

между выражением творчества и критическим мышлением. Вы генерируете и воплощаете свои идеи в жизнь, действуя свободно и творчески, но затем вы аккуратно повышаете их качество, критически оценивая свою работу.

На самом деле вы уже работали таким образом, когда в части II отслеживали количество возможностей команд, которые вам удавалось реализовать за 45 минут, и пытались понять, что можно усовершенствовать. Однако вы не могли одновременно выполнять задания и анализировать свое движение, поскольку критическое мышление – убийца креативности. Этот совет распространяется практически на все известные мне творческие дисциплины и помогает во время работы не становиться на пути у самого себя.

---

**Внимание!** Критика во время созидания вредит вашему воображению. Креативность без критики не приводит к созданию ничего, кроме мусора. Вам необходимо и то, и другое, но не одновременно.

---

В части III вы «переключите передачу», сфокусировавшись на качестве и развитии повышающих качество процессов. Чтобы не сбивать вас с толку, я дам следующее определение качеству:

низкий процент брака и понятный код.

Большинство программистов пугающе ужасны в обоих компонентах. Подавляющее большинство разработчиков уверены, что их работа заканчивается, как только завершает свою работу компилятор. Они провели тест, так что дело сделано! Я называю это «и так сойдет». Этот принцип не предполагает никакой самокритичной оценки собственной работы, поскольку такие программисты полностью доверяют поиск ошибок компьютеру. Они также никогда не беспокоятся, поймет ли их код кто-нибудь еще, обращая внимание лишь на то, отвечает ли он минимальным требованиям. Если бы вы спросили, какой у них дневной процент брака, они бросили бы на вас свирепый взгляд и сказали бы, что это не имеет значения. Охват кода? Ха. Их тестовый набор содержит сто тысяч строк кода! Он должен проверять все!

Чтобы стать лучше как программист, вы должны начать с жестокостью изучать собственные практики и показатели качества. Я называю это жестокой работой, потому что она ясно и четко показывает, насколько вы плохи, – и такая демонстрация может превратиться в трагедию для людей, счастливо полагающих, что у их работы недостатков нет. Те же, кто страдает от синдрома

самозванца[4], найдут этот анализ качества освежающим, так как он даст достойное представление о том, насколько хорошо вы справляетесь, и поможет составить план по совершенствованию.

## Обучение качеству с использованием структур данных

Структуры данных – довольно простое понятие. У вашего компьютера есть память и данные, которые нужно поместить в эту память. Вы можете либо запихивать данные в случайные места, либо организовать структуру, которая упростит их обработку. С самого начала существования компьютеров люди исследовали, как структурировать данные для различных целей, а также насколько хорошо функционируют созданные структуры. Поскольку структуры данных так хорошо определены, их можно использовать для проверки качества. Вы будете воплощать в жизнь каждую структуру данных и создавать тест к ней, а затем за два шага определять качество воплощения.

Каждое упражнение по структурам данных выполняется следующим образом.

1. В каждом упражнении будет описываться структура данных и что с ней можно сделать. Это описание будет содержать диаграммы и пример кода. Я также дам полноценное описание структуры без кода, чтобы добиться правильного понимания.

2. У вас уже может быть набор тестов, которыми вы также должны воспользоваться, но они, вероятно, написаны на русском языке, так что вы дополнительно создадите автоматизированный тест.

3. Можете продолжить обучение в формате 45-минутных сессий с перерывами, а можете уделять каждой реализации больше времени. Я советую начать с нескольких быстрых сеансов, а затем «стать серьезнее» и совершенствовать реализацию в течение большего количества сеансов.

4. Когда вам покажется, что вы закончили, переключитесь в режим критики и выясните, что вы действительно сделали. Вы будете следовать проверке, проводящей вас по вашему коду, критически находя ошибки и отслеживая, сколько вы их допустили.

[4] Психологическое явление, при котором человек не может приписать свои достижения собственным способностям и усилиям. – *Прим. ред.*

5. Наконец, вы исправите найденные дефекты и продолжите работать над упражнением, пока не выполните его полностью.

Этот процесс сложен, поэтому первые два упражнения этой части (13 и 14) будут сделаны мной, со всем браком, что я допускаю, со всем кодом, что у меня есть. Вы сможете увидеть, как это работает, на видео и прочесть мой код в упражнениях, поняв, чего ожидать. Я буду следовать описанной выше процедуре как можно более строго, так что смотрите видео внимательно.

## Как изучать структуры данных

Формально алгоритмы и структуры данных изучают математически, но я не буду слишком углубляться в теорию. Если вам интересно это «легкое» введение, можете прочесть несколько книг по теме и заниматься изучением этого ответвления информатики еще многие годы. В этой книге я буду предоставлять вам упражнения, чтобы вы могли воплощать их по памяти и понимать, как они работают. Вам не понадобятся формальные доказательства, только простой код Python и желание программировать.

Я хочу, чтобы вы следовали особому способу выполнения упражнений, позволяющему делать это по памяти. Я пользуюсь им, когда изучаю музыку или пытаюсь нарисовать то, что вижу. Этот способ работает везде, где вам нужно запомнить концепцию и творчески применить ее в различных ситуациях – что лишает вас возможности просто вызубрить ее. Вместо этого вы осуществляете то, что я называю «запоминай, делай попытку, проверяй».

1. Соберите всю информацию, весь материал, описывающий то, что необходимо освоить. Сделайте все, что в ваших силах, чтобы запомнить его, даже если это небольшая часть информации.

2. Уберите всю информацию прочь, чтобы не видеть ее. Я убираю ее в другую комнату, так что если мне нужно еще раз на нее взглянуть, приходится покидать рабочее место.

3. Попытайтесь по памяти воссоздать нужную информацию. Постарайтесь что-нибудь записать, правильно или нет.

4. Когда вы выложили все, что запомнили, вернитесь и сравните полученный результат с исходной информацией. Отметьте все, что вы сделали неверно, затем повторите попытку.

5. Пользуясь списком ошибок, концентрируйтесь на запоминании, чтобы при следующей попытке исправить ошибки, – и повторяйте все снова.

Я обычно трачу от 2 до 15 минут на запоминание, затем от 10 до 45 минут на попытки, но вы сами поймете, когда у вас закончатся знания и вам придется идти за новыми. Вот конкретный пример, где я объясняю, как рисую (с красками или без) по памяти.

1. Я собираюсь нарисовать цветок, так что помещаю цветок в одной комнате, свой набор для рисования – в другой.

2. Я сажусь в комнате с цветком и пристально смотрю на него. Делаю зарисовки цветка. Рисую его пальцем в воздухе и пытаюсь увидеть его мысленным взором. Я создаю образ того, как рисую каждый лепесток, стебель, все. Я запоминаю пропорции. Я могу даже сделать записи о цвете и попробовать смешать краски прямо в комнате с цветком.

3. Я ухожу из комнаты с цветком. Быстро возвращаюсь в комнату для рисования и пытаюсь вызвать в памяти этот цветок, решая, что рисовать дальше. Кажется, я наконец-то ясно вижу листок. Рисую листок. Вероятно, ясно вижу горшок, рисую его. Я продолжаю закрывать глаза и пытаться вспомнить образ, а затем изобразить его.

4. Когда я захожу в тупик или у меня заканчивается время, я встаю и отношу свой небольшой холст в комнату с цветком, сравниваю с тем, что вижу. Затем я делаю записи, указывая, что сделал неправильно. Один из лепестков слишком длинный? Горшок нарисован под другим углом? Грунт слишком темный? Я набрасываю записи, исследуя, что я сделал неправильно.

5. Потом я отношу рисунок обратно в комнату для рисования, возвращаюсь в комнату с цветком и использую составленный список ошибок, чтобы продолжить обучение по памяти, готовясь к следующему «раунду».

Картины, которые у меня получаются при использовании этой методики, обычно выглядят довольно странно, но натуралистично, в зависимости от того, сколько «подходов» я делаю и как часто практикуюсь. В итоге это помогло мне стать лучше и быстрее схватывать то, что я вижу, ведь я научился сохранять в памяти больше визуальной информации на более долгие периоды времени.

Когда вы выполняете упражнения в этой книге, можете использовать тот же процесс для работы над своей способностью повторить их по требованию на собеседовании. Сначала вы должны просто сесть и реализовать их, используя всю информацию, которая вам доступна, и узнать, как они работают. Трудно запомнить то, что не понимаешь. После того, как у вас исчезнут проблемы с реализацией, вы можете начать тренировать свою память.

1. Сложите все книги, записи, диаграммы и информацию об алгоритме в одной комнате, а свой ноутбук оставьте в другой. Распечатайте свой код, если нужно.

2. Проведите добрых 15 минут в комнате алгоритма, изучая информацию, делая записи, строя дополнительные диаграммы, визуализируя поток данных и занимаясь всем остальным, что помогает выучить его.

3. Оставьте всю информацию в комнате алгоритма, пройдите в комнату с ноутбуком, сядьте и предпримите попытку реализации по памяти. Не тратьте на это больше 45 минут.

4. Вернитесь в комнату алгоритма с ноутбуком и сделайте записи о том, что вы выполнили неправильно.

5. Отложите ноутбук, вернитесь в комнату алгоритма и перейдите к следующему «раунду» запоминания и изучения, прежде чем повторить все снова. Сфокусируйтесь на том, что вы делали неправильно, это облегчит задачу.

Первые несколько раз это будет удручающе, но вскоре вы обнаружите, что работать таким образом становится проще и во многих случаях появляется ощущение спокойствия.

УПРАЖНЕНИЕ 13

# Односвязные списки

Первая структура данных, которую вы будете реализовывать, это односвязный список. Я предоставлю описание структуры данных, составлю список всех операций, которые вы должны осуществить, а также дам необходимый тест, проверяющий вашу реализацию. Сначала вы должны предпринять собственную попытку, а затем просмотреть видео моей работы и после провести аудиторскую проверку, чтобы понять процесс.

---

**Внимание!** Эти структуры данных реализованы совсем не эффективно! Они умышленно выполнены медленно и безыскусно, чтобы в главах 18 и 19 можно было поговорить об измерении и оптимизации кода. Если вы станете использовать эти структуры данных в профессиональной деятельности, то столкнетесь с проблемами производительности.

---

## Описание

При работе со многими структурами данных в языке Python и многих других объектно-ориентированных языках программирования, вы должны понимать три основные концепции.

1. «Узел». Обычно это контейнер или ячейка памяти для структуры данных. Ваши значения отправляются сюда.

2. «Ребро», но мы будем использовать термин «указатель» или «ссылка». Указывает на другие узлы. Располагается внутри каждого узла, обычно как переменная экземпляра.

3. «Контроллер». Это некий класс, знающий, как использовать указатели в узле, чтобы корректно структурировать данные.

В Python мы сопоставим эти понятия следующим образом.

1. Узлы – это просто объекты, определенные классом.

2. Указатели (ребра) – переменные экземпляра в объектах узла.

3. Контроллер – просто еще один класс, который использует узлы для хранения, а также для структурирования данных. Это сюда поступают все ваши операции (`push`, `pop`, `list` и т. д.), и обычно пользователь контроллера никогда не имеет дело с узлами или указателями.

В некоторых книгах по алгоритмам вы увидите реализации, объединяющие узел и контроллер в один класс или структуру. Это сильно запутывает и нарушает расстановку задач в замысле. Узлы лучше отделять от класса контроллера, чтобы у одной вещи было одно предназначение и вам было легче искать ошибки.

Представьте, что мы хотим упорядочить список автомобилей. У нас есть первый автомобиль, ведущий ко второму, и так далее до конца. Держа в уме этот список, мы начнем понимать соответствующую конструкцию узла/указателей/контроллера.

1. Узлы содержат описание каждого автомобиля. Возможно, это просто переменная `node.value` класса `Car`. Можно назвать его `SingleLinkedListNode` или, если вам лень, `SLLNode`.

2. В каждом `SLLNode` есть ссылка `next` на следующий узел в цепи. `node.next` перемещает вас к следующему автомобилю в цепи.

3. Далее у контроллера (`SingleLinkedList`) есть операции, такие как `push`, `pop`, `first` или `count`, принимающие `Car` и использующие узлы для внутреннего их хранения. Когда вы «толкаете» (операция `push`) `Car` на контроллер `SingleLinkedList`, это сохраняет `Car` в конце внутреннего списка связанных узлов.

---

**Внимание!** Зачем мы это делаем, если в Python есть замечательный быстрый тип данных `list` (список)? Главным образом лишь затем, чтобы освоить структуры данных. В повседневной жизни вы просто воспользовались бы списком.

---

Для реализации класса `SinglеLinkedListNode` нам понадобится простой класс вроде этого:

sllist.py

```
1    class SingleLinkedListNode(object):
2
3        def __init__(self, value, nxt):
4            self.value = value
5            self.next = nxt
6
7        def __repr__(self):
8            nval = self.next and self.next.value or None
9             return f"[{self.value}:{repr(nval)}]"
```

Приходится использовать слово `nxt`, поскольку слово `next` зарезервировано в Python. Это очень простой класс. Самое сложное в нем – функция `__repr__`. Она просто печатает вывод отладки, когда используется формат `''%r''`, или когда в узле вызывается `repr()`. Она должна возвращать строку.

---

**Внимание!** Уделите время выяснению того, как вручную построить список, используя только класс `SingleLinkedListNode`, а затем вручную же пройдитесь по нему. Это будет хорошим 45-минутным заданием на данном этапе упражнения.

---

# Контроллер

Как только мы определили наши узлы в классе `SingleLinkedListNode`, мы можем выяснить, что конкретно должен делать контроллер. У каждой структуры данных есть список распространенных операций, благодаря которым она становится полезной. Разным операциям требуется разный объем памяти (места) и разное количество времени. Соответственно, некоторые операции «легкие», другие – быстрые. Структура `SingleLinkedListNode` выполняет одни операции очень быстро, но многие другие – крайне медленно. Вы заметите это при работе над реализацией.

Простейший способ увидеть операции – взглянуть на каркас класса `SingleLinkedListNode`:

sllist.py

```
class SingleLinkedList(object):

    def __init__(self):
        self.begin = None
        self.end = None

    def push(self, obj):
        """Присоединяет новое значение к концу списка."""

    def pop(self):
        """Удаляет последний элемент и возвращает его."""

    def shift(self, obj):
        """То же, что push."""

    def unshift(self):
        """Удаляет первый элемент и возвращает его."""

    def remove(self, obj):
      """Находит совпадающий элемент и удаляет его из списка."""

    def first(self):
       """Возвращает *ссылку* на первый элемент, не удаляет."""

    def last(self):
      """Возвращает ссылку на последний элемент, не удаляет."""

    def count(self):
        """Подсчитывает количество элементов в списке."""

    def get(self, index):
        """Получает значение по индексу."""

    def dump(self, mark):
        """Функция отладки, сбрасывающая содержимое списка."""
```

В других упражнениях я просто буду сообщать вам операции и оставлять вас размышлять, но сейчас предоставлю руководство по реализации. Просмотрите список функций в классе `SingleLinkedList`, чтобы увидеть каждую операцию, и прочитайте комментарии, описывающие их работу.

# Тест

Я предоставлю тест, который вы должны пройти, реализовывая этот класс. Вы увидите, что я прошелся по каждой операции и постарался учесть большинство пограничных случаев, но когда дело дойдет до проверки, окажется, что часть из них из них я пропустил. Люди иногда забывают выполнить тест для случаев «ноль элементов» или «один элемент».

test_sllist.py

```
from sllist import *

def test_push():
    colors = SingleLinkedList()
    colors.push("Pthalo Blue")
    assert colors.count() == 1
    colors.push("Ultramarine Blue")
    assert colors.count() == 2

def test_pop():
    colors = SingleLinkedList()
    colors.push("Magenta")
    colors.push("Alizarin")
    assert colors.pop() == "Alizarin"
    assert colors.pop() == "Magenta"
    assert colors.pop() == None

def test_unshift():
    colors = SingleLinkedList()
    colors.push("Viridian")
    colors.push("Sap Green")
    colors.push("Van Dyke")
    assert colors.unshift() == "Viridian"
    assert colors.unshift() == "Sap Green"
    assert colors.unshift() == "Van Dyke"
    assert colors.unshift() == None

def test_shift():
    colors = SingleLinkedList()
    colors.shift("Cadmium Orange")
    assert colors.count() == 1

    colors.shift("Carbazole Violet")
```

```
34          assert colors.count() == 2
35
36          assert colors.pop() == "Cadmium Orange"
37          assert colors.count() == 1
38          assert colors.pop() == "Carbazole Violet"
39          assert colors.count() == 0
40
41      def test_remove():
42          colors = SingleLinkedList()
43          colors.push("Cobalt")
44          colors.push("Zinc White")
45          colors.push("Nickle Yellow")
46          colors.push("Perinone")
47          assert colors.remove("Cobalt") == 0
48          colors.dump("before perinone")
49          assert colors.remove("Perinone") == 2
50          colors.dump("after perinone")
51          assert colors.remove("Nickle Yellow") == 1
52          assert colors.remove("Zinc White") == 0
53
54      def test_first():
55          colors = SingleLinkedList()
56          colors.push("Cadmium Red Light")
57          assert colors.first() == "Cadmium Red Light"
58          colors.push("Hansa Yellow")
59          assert colors.first() == "Cadmium Red Light"
60          colors.shift("Pthalo Green")
61          assert colors.first() == "Pthalo Green"
62
63      def test_last():
64          colors = SingleLinkedList()
65          colors.push("Cadmium Red Light")
66          assert colors.last() == "Cadmium Red Light"
67          colors.push("Hansa Yellow")
68          assert colors.last() == "Hansa Yellow"
69          colors.shift("Pthalo Green")
70          assert colors.last() == "Hansa Yellow"
71
72      def test_get():
73          colors = SingleLinkedList()
74          colors.push("Vermillion")
75          assert colors.get(0) == "Vermillion"
76          colors.push("Sap Green")
```

```
77          assert colors.get(0) == "Vermillion"
78          assert colors.get(1) == "Sap Green"
79          colors.push("Cadmium Yellow Light")
80          assert colors.get(0) == "Vermillion"
81          assert colors.get(1) == "Sap Green"
82          assert colors.get(2) == "Cadmium Yellow Light"
83          assert colors.pop() == "Cadmium Yellow Light"
84          assert colors.get(0) == "Vermillion"
85          assert colors.get(1) == "Sap Green"
86          assert colors.get(2) == None
87          colors.pop()
88          assert colors.get(0) == "Vermillion"
89          colors.pop()
90          assert colors.get(0) == None
```

Внимательно изучите этот тест, чтобы получить необходимое представление о том, как должна работать каждая операция, прежде чем осуществлять реализацию. Я не стал бы писать в файл сразу весь этот код – гораздо лучше проводить один тест за раз, проверяя работу по кускам.

---

**Внимание!** Если на этом этапе вам незнакомо автоматизированное тестирование, ознакомьтесь с видео, где я его осуществляю.

---

## Вводный аудит

Выполняя тесты, вы проводите аудит (проверку) своего кода, ища ошибки. В итоге вы отследите количество брака, обнаруженного при аудите, но сейчас вы будете просто практиковать проверку своего кода после его написания. «Аудит» – это то, чем занимается налоговый агент, когда правительство подозревает вас в мошенничестве с налогами. Он перебирает каждый платеж, каждую входящую сумму, каждую исходящую и выясняет, почему вы потратили деньги так, как потратили. Аудит кода похож на этот процесс, так как вы перебираете каждую функцию и анализируете принятые параметры, а также все значения на выходе.

Для проведения базового аудита сделайте следующее.

1. Начните сверху вашего тестового случая. Для примера возьмем функцию `test_push`.

2. Взгляните на первую строку кода и определите, что она вызывает и что создает. В нашем случае это `colors = SingleLinkedList()`. То есть мы создаем переменную `colors` и вызываем функцию `SingleLinkedList.__init__`.

3. Перейдите к началу этой функции `__init__`, держа перед глазами тестовый пример и целевую функцию (`__init__`). Убедитесь, что вы все сделали правильно. Затем убедитесь, что вы вызвали эту функцию с правильным количеством и типом аргументов. В этом случае `__init__` принимает только `self`.

4. Затем вы залезаете в `__init__` и, строка за строкой, аналогичным образом подтверждаете каждый вызов функции и каждую переменную. Верное ли количество аргументов? Правильный ли тип?

5. В каждом ответвлении (конструкция `if`, циклы `for` и `while`) вы убеждаетесь, что логика верна и что все возможные условия были учтены. Есть ли в конструкциях `if` ветви `else`? Имеют ли окончание циклы `while`? Рассмотрите каждое ответвление и так же проанализируйте функции, «ныряя», проверяя переменные и возвращаясь назад, проверяя возвращаемые значения.

6. Когда вы доберетесь до конца функции или до любого выражения `return`, вернитесь к вызову `test_push` и проверьте, отвечает ли возвращаемое значение тому, чего вы ожидали, совершая этот вызов. Однако помните, что это также необходимо сделать для каждого вызова внутри `__init__`.

7. Работа сделана, когда вы достигли конца функции `test_push`. Вы рекурсивно проверили весь код внутри и за пределами всех функций, которые она вызывает.

Этот процесс поначалу кажется утомительным, и да, он таким и является, но со временем вы будете делать все быстрее. На видео вы увидите, как я делаю это перед тем, как запускаю каждый тест (или, по крайней мере, изо всех сил пытаюсь делать). Я следую такому алгоритму.

1. Пишу немного кода теста.

2. Пишу код для работы теста.

3. Провожу аудит и того, и другого.

4. Запускаю тест и узнаю, был ли я прав.

## Задача упражнения

Упражнение состоит в следующем. Первым делом пройдитесь по тесту и поймите, что он делает. Также изучите код в `sllist.py` и выясните, что требуется от вас. Я полагаю, при вашей первой попытке реализовать функцию в `SingleLinkedList` вы сначала напишете комментарии, указывая, что она будет делать, а потом уже введете код Python, приводящий эти комментарии в исполнение. На видео вы увидите, как я это делаю.

Когда вы проведете один или два 45-минутных сеанса, пытаясь выполнить задание, придет время просмотреть видео. Сначала вам нужно предпринять самостоятельную попытку, чтобы иметь лучшее представление о том, что буду делать я. Это облегчит понимание видеоматериала. На видео я не буду разговаривать, лишь писать код, но мой голос за кадром будет пояснять, что происходит. Кроме того, видео будет ускорено для экономии времени, и я вырежу все неинтересные ошибки и пустые моменты.

Сразу, как только вы увидите, как я выполняю задание, и сделаете записи (верно?), идите и предпринимайте более серьезные попытки, а также как можно внимательнее проводите аудит кода.

## Аудит

Когда вы завершили написание кода, убедитесь, проведите аудит, описанный мной во введении к части III. На случай, если вы не вполне уверены, как это делается, я также буду проводить его на видео к данному упражнению.

## Практическое задание

Вашим заданием для этого упражнения будет попытаться реализовать этот алгоритм снова, полностью из памяти – так, как описано во введении к части III. Вы также должны попытаться понять, какие операции в этой структуре данных, вероятнее всего, будут наиболее медленны. Когда вы закончите, проведите аудит того, что вы создали.

# Двусвязные списки

На выполнение предыдущего упражнения у вас могло уйти довольно много времени, ведь вам нужно было заставить работать односвязный список. К счастью, на видео представлено достаточно информации о том, как завершить упражнение и провести аудит своего кода. В этом упражнении вы будете реализовывать усовершенствованную версию связных списков под названием "двусвязный список".

В случае с односвязным списком вам необходимо было осознать, что все операции, затрагивающие конец списка, должны пройти через каждый узел, прежде чем попасть в конец. Односвязный список эффективен лишь с начала списка, где вы легко можете изменить указатель `next`. Операции `shift` и `unshift` быстрые, но `pop` и `push` замедляют работу по мере того, как список растет. Вы, возможно, ускорили бы процесс, сохранив ссылку на предпоследний элемент, но что, если нужно будет заменить этот элемент? Вам снова-таки придется проходиться по всем элементам, чтобы найти нужный. Благодаря мелким изменениям, вроде этого, можно получить небольшой прирост скорости, но лучшим решением будет просто изменить структуру, упростив работу с любого места.

Двусвязный список почти аналогичен односвязному, с той лишь разницей, что первый в дополнение содержит ссылку `prev` (от previous – предыдущий) на узел `DoubleLinkedListNode`, предшествующий данному узлу. Один дополнительный указатель на узел – и внезапно столько операций становятся значительно проще. Вы также можете легко добавить указатель в `end` в двусвязном списке и получить прямой доступ и к началу, и к концу. Благодаря этому, операции `push` и `pop` снова становятся эффективными, ведь появляется возможность получить доступ непосредственно к концу и использовать указатель `node.prev`, чтобы получить предыдущий узел.

С учетом таких изменений наш класс узла будет выглядеть следующим образом:

dllist.py

```
1    class DoubleLinkedListNode(object):
2
3        def __init__(self, value, nxt, prev):
4            self.value = value
```

```
            self.next = nxt
            self.prev = prev

        def __repr__(self):
            nval = self.next and self.next.value or None
            pval = self.prev and self.prev.value or None
            return f"[{self.value}, {repr(nval)}, {repr(pval)}]"
```

Добавилась лишь строка кода `self.prev = prev` и, соответственно, изменилась функция `__repr__`. Класс `DoubleLinkedList` использует те же самые операции, что и класс `SingleLinkedList`, лишь для конца списка добавилась еще одна переменная:

dllist.py

```
class DoubleLinkedList(object):

    def __init__(self):
        self.begin = None
        self.end = None
```

# Введение в инвариантные условия

Мы осуществляем все те же операции, что и прежде, но теперь добавляем несколько новых соображений:

dllist.py

```
    def push(self, obj):
        """Присоединяет новое значение к концу списка."""

    def pop(self):
        """Удаляет последний элемент и возвращает его."""

    def shift(self, obj):
        """На самом деле, то же, что push."""

    def unshift(self):
        """Удаляет первый элемент (с начала) и возвращает его."""
```

```
12
13          def detach_node(self, node):
14              """Вам иногда потребуется эта операция, но чаще внутри
15              remove(). Она должна принимать узел и отсоединять его
16              от списка, независимо, находится узел в начале, конце
17              или середине inside remove()"""
18
19          def remove(self, obj):
20              """Находит совпадающий элемент и удаляет его из списка."""
21
22          def first(self):
23              """Возвращает *ссылку* на первый элемент, не удаляет."""
24
25          def last(self):
26              """Возвращает ссылку на последний элемент, не удаляет."""
27
28          def count(self):
29              """Подсчитывает количество элементов в списке."""
30
31          def get(self, index):
32              """Получает значение по индексу."""
33
34          def dump(self, mark):
35              """Функция отладки, сбрасывающая содержимое списка."""
```

Имея указатель `prev`, в каждой операции мы должны обрабатывать больше условий.

1. Ноль элементов? Тогда `self.begin` и `self.end` должны быть `None`.

2. Если имеется один элемент, тогда `self.begin` и `self.end` должны быть равны (указывать на один и тот же узел).

3. Указатель `prev` первого элемента всегда должен быть `None`.

4. Указатель `next` последнего элемента всегда должен быть `None`.

Этих принципов необходимо придерживаться на протяжении работы двусвязного списка, поэтому они называются инвариантными (то есть неизменными) условиями, или просто инвариантами. Их смысл в том, что они

являются базовыми индикаторами, свидетельствующими о правильной работе структуры вне зависимости от всего остального. Все тесты и вызовы инструкции `assert`, которые вы часто осуществляете, вы можете поместить в специальную функцию под названием `_invariant`, которая будет осуществлять эти проверки. Таким образом вы снизите свой процент брака и всегда сможете сказать: «Неважно, что я делаю, раз результат истинен».

Единственная проблема с инвариантами заключается в том, что на их выполнение может уходить много времени. Если вызов каждой функции также дважды вызывает еще одну функцию, это значительно обременяет каждую из них. Если и ваша функция-инвариант делает что-то затратное, дело становится еще хуже. Представьте, будто вы добавили инвариант «Все узлы содержат указатели `next` и `prev` – кроме первого и последнего». Это значило бы, что каждый вызов функции дважды выполняет полный обход списка. Если вы желаете удостовериться, что класс работает постоянно, оно того стоит. Если не желаете, возникает проблема.

В этой книге вы будете использовать инварианты везде, где только можно, но держите в уме, что вы должны использовать их всегда. Ключ к эффективному использованию инвариантов – найти способы включить их в наборы тестов или отладку, либо использовать в процессе первоначальной разработки. Я советую вам вызывать инварианты только в верхней части функций или только внутри тестового набора. Это приемлемый компромисс.

## Задача упражнения

В данном упражнении вы будете реализовывать операции для двусвязного списка, но на этот раз также используете функцию `_invariant`, чтобы проверить, что все работает, как нужно, до и после каждой операции. Лучший способ сделать это – вызывать инвариант в верхней части функции, а затем в ключевых точках в тестовом наборе. Тестовый набор для `двусвязного списка` практически идентичен тесту для односвязного списка, за исключением вызовов функции `_invariant` в ключевых точках.

Как и с односвязным списком, вам нужно будет самостоятельно изучить эту структуру данных. Изобразите структуры узлов на бумаге и осуществите некоторые операции вручную. Затем работайте над `DoubleLinkedListNode` в своем файле `dllist.py`. После этого на протяжении одного или двух 45-минутных сеансов попытайтесь разобраться с несколькими операциями. Советую начинать с `push` и `pop`. После этого можете посмотреть видео, где

я использую сочетание аудита кода и функции `_invariant` для проверки работы.

## Практическое задание

Как и в случае с предыдущим упражнением, вам нужно снова реализовать эту структуру данных по памяти. Поместите все, что вам известно о ней, в одну комнату, а ноутбук – в другую. Это необходимо продолжать до тех пор, пока вы не сможете реализовать двусвязный список полностью по памяти.

УПРАЖНЕНИЕ 15

# Стеки и очереди

При работе со структурами данных вы будете часто сталкиваться с похожими структурами. Стек похож на односвязный список из упражнения 13, а очередь – на двусвязный список из упражнения 14. Единственное отличие состоит в том, что для упрощения работы и стек, и очередь ограничивают возможные операции. Благодаря этому снижается уровень брака, ведь теперь нельзя создать проблемы, случайно использовав стек как очередь. В стеке узлы «проталкиваются» (pushed) на «верх» (top), а затем «выталкиваются» (popped) тоже сверху. В случае с очередью узлы помещаются (shifted) в «хвост» структуры, а затем убираются (unshifted) из ее «головной» части. Обе эти операции представляют собой всего лишь упрощения односвязного и двусвязного списка, где стек позволяет осуществлять лишь `push` и `pop`, а очередь – только `shift` и `unshift`.

Визуализируя стек, представляйте себе стопку книг. Представьте настоящие увесистые тома – если бы я сложил 20 таких книг одну на другую, стопка, вероятно, весила бы больше 45 килограммов. Когда вы хотите добавить новые книги, вы же не поднимаете всю стопку и не засовываете книги снизу, верно? Нет, вы складываете их на вершину стопки. Вы опускаете их, но для этого действия мы также можем использовать слово push («толкать»). Если бы вам понадобилось взять книгу из стопки, вы подняли бы часть из них и схватили нужную, но в конечном счете вам понадобилось бы забрать книги наверху, чтобы добраться до тех, что ниже. Вы поднимали бы каждую книгу сверху, или, как мы сказали бы, «выталкивали» (pop off) ее. Так работает стек, и если захотите его представить – это просто связный список книг, держащихся вместе благодаря силе тяжести.

Очередь лучше всего представлять в виде очереди людей в банке, с «головой» и «хвостом». Обычно она огорожена веревками, вход находится в конце очереди, а выход – там, где сидят кассиры. Вы занимаете место в очереди с «хвоста», и мы называем это `shift`, принятым словом в контексте этой структуры данных. Как только вы заняли место в очереди, вы не можете просто обойти всех людей перед собой, иначе они рассердятся. Так что вы ждете, и по мере того, как каждый человек перед вами покидает очередь, вы продвигаетесь все ближе к выходу с «головы» очереди. Когда вы достигаете конца, можете выходить из очереди – мы называем это `unshift`. Таким образом, структура данных «очередь» подобна двусвязному списку, ведь вы работаете с обоими концами структуры данных.

Можно отыскать множество примеров из реальной жизни, которые помогут визуализировать работу структуры данных. Теперь уделите время изображению этих принципов на бумаге или на самом деле сложите стопку книг и воспроизведите на ней соответствующие операции. Сколько еще вы можете найти реальных ситуаций, напоминающих стек или очередь?

## Задача упражнения

Теперь я буду отучивать вас от упражнений, непосредственно основанных на коде, заставляя реализовывать структуры данных, исходя только из их описаний. В этом упражнении вы сначала должны реализовать стек, пользуясь поданным здесь начальным кодом и знаниями об односвязном списке, которые вы приобрели в упражнении 13. Как только вы закончите с этим, попытайтесь создать структуру данных «очередь» из ничего.

Класс `StackNode` практически идентичен классу `SingleLinkedListNode`. Фактически я просто перекопировал последний, изменив имя:

stack.py

```
class StackNode(object):

    def __init__(self, value, nxt):
        self.value = value
        self.next = nxt

    def __repr__(self):
        nval = self.next and self.next.value or None
        return f"[{self.value}:{repr(nval)}]"
```

Управляющий класс `Stack` также сильно напоминает `SingleLinkedList`, разве что я использую `top` для первого элемента. Это согласуется с принципом стека.

stack.py

```
class Stack(object):

    def __init__(self):
```

```
4            self.top = None
5
6        def push(self, obj):
7            """Добавляет новое значение на верх стека."""
8
9        def pop(self):
10           """Удаляет значение, находящееся на вершине стека."""
11
12       def top(self):
13           """Возвращает *ссылку* на первый элемент, не удаляет."""
14
15       def count(self):
16           """Подсчитывает количество элементов в стеке."""
17
18       def dump(self, mark="----"):
19           """Функция отладки, сбрасывающая содержимое стека."""
```

Теперь ваша задача состоит в том, чтобы реализовать стек и написать для него тест, аналогичный тем, что вы делали в упражнении 13. Убедитесь, что тест будет охватывать каждую операцию всеми возможными способами. Однако помните, что значение должно добавляться на верхушку стека, так что вам нужна именно такая ссылка.

Как только получите работающий стек, приступайте к реализации очереди, базируя ее на двусвязном списке. Из работы со стеком вы поняли, что можно оставлять все ту же внутреннюю структуру, что и в односвязном списке, изменяя лишь допустимые функции. То же самое касается и очереди. Уделите время схематическому изображению и визуализации работы очереди, а затем выясните, в чем состоят ее ограничения по сравнению с двусвязным списком. Как только поймете это, создавайте свою очередь.

## Ломаем это

Ошибки в реализации этих структур данных – всего лишь вопрос недисциплинированности. Посмотрите, что произойдет, если одна операция не сможет использовать нужный конец структуры.

Вы также могли заметить, что существует постоянный риск ошибки смещения на единицу. Я установил `self.top = None` для случая, когда структура

пуста. Это значит, что по достижении 0 элементов вам приходится «жонглировать» `self.top`. В качестве альтернативы можно заставить `self.top` всегда указывать на `StackNode` (или любой узел), и сказать, что структура пуста, когда вы добираетесь до этого последнего элемента. Разберитесь, как такие изменения влияют на вашу реализацию. Стала ли она более подвержена ошибкам или наоборот?

## Дальнейшее обучение

Множество операций для этой структуры данных чрезвычайно неэффективны. Пройдитесь по коду, написанному вами для каждой структуры данных, и попытайтесь догадаться, какие из функций наиболее медленные. Как только вам что-то придет в голову, попробуйте объяснить, почему они могут быть медленными. Кроме того, выясните, что об этих структурах данных говорят другие люди. В упражнениях 18 и 19 вы узнаете, как выполнять анализ производительности этих структур данных и настраивать их.

Наконец, действительно ли вам нужно было реализовывать новую структуру данных целиком или вы могли просто использовать наработки односвязного и двусвязного списка? Как в таком случае изменится ваша реализация?

УПРАЖНЕНИЕ 16

# Пузырьковая и быстрая сортировка, сортировка слиянием

Теперь вы попытаетесь реализовать алгоритмы сортировки для структуры данных «двусвязный список». В последующих описаниях я использую словосочетание «список чисел», подразумевая рандомизированный (случайный) список определенных вещей. Это могут быть колода карт, пронумерованные листы бумаги, перечни имен – все, что угодно, поддающееся сортировке. Обычно, выделяют три вида сортировки списков чисел.

- Пузырьковая сортировка. Скорее всего, это именно тот способ, который вы использовали бы, если бы не знали ничего о сортировке. Он предполагает обычное прохождение по списку и перестановку элементов во всех найденных неверно упорядоченных парах. Вы раз за разом проходите по списку, переставляя местами элементы внутри пар, пока не выполните проход без перестановок. Это способ прост для понимания, но чрезвычайно медленен.

- Сортировка слиянием. Данный алгоритм сортировки делит список на половины, затем на четверти и так далее, пока дальнейшее деление не станет невозможным. Затем он снова объединяет разделы, но в правильной последовательности, проверяя порядок каждого раздела. Это умный алгоритм, отлично работающий со связными списками, но похуже с массивами фиксированного размера, так как тогда нужна очередь, чтобы отслеживать разделы.

- Быстрая сортировка. Она похожа на сортировку слиянием, потому что применяется принцип «разделяй и властвуй», но в быстрой сортировке, вместо разделения и объединения списка, элементы переставляются относительно точки разделения. В простейшем случае вы выбираете возрастающий диапазон и точку разделения. Затем вы помещаете те элементы, что больше точки разделения (опорного элемента), после нее, а те, что меньше точки разделения, – перед ней. После чего снова выбираете верхнее и нижнее значение и точку

разделения внутри образовавшегося набора и повторяете операцию. Этот алгоритм разделяет список на меньшие куски, но не разбивает их на части, в отличие от сортировки слиянием.

## Задача упражнения

Предназначение данного упражнения – научиться реализовывать алгоритм, описание которого выполнено на языке псевдокода. Вы будете изучать алгоритмы по ссылкам, которые я укажу (главным образом на Википедию), а затем использовать псевдокод, чтобы их реализовать. Первые два я быстро сделаю здесь и более подробно – на видео к упражнению. Затем вы должны будете самостоятельно создать простой алгоритм сортировки. Сперва давайте взглянем на описание пузырьковой сортировки в Википедии (**en.wikipedia.org/wiki/Bubble_sort**):

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            /* если порядок в этой паре неверный */
            if A[i-1] > A[i] then
                /* переставить их местами и запомнить изменение */
                swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
```

Поскольку псевдокод – это просто неформальное описание алгоритма, он будет сильно различаться в зависимости от книг, авторов и страниц в Википедии. Предполагается, что вы сможете прочесть этот «почти язык программирования» и перевести его на тот, который для вас удобен. Иногда он будет выглядеть как старый язык программирования под названием «Алгол», в других случаях станет напоминать корявый код на JavaScript или Python. Вам просто нужно догадаться, о чем идет речь, и перевести все на нужный язык. Вот моя первоначальная реализация этого конкретного псевдокода:

sorting.py

```
    def bubble_sort(numbers):
        """Сортирует список чисел путем пузырьковой сортировки."""
        while True:
            # начать, предполагая, что он отсортирован
            is_sorted = True
            # сравниваем 2 элемента за раз, переходим к следующей паре
            node = numbers.begin.next
            while node:
                # выполняем проход, сравнивая узел со следующим
                if node.prev.value > node.value:
                    # если следующий узел больше, нужна перестановка
                    node.prev.value, node.value = node.value,
                    node.prev.value
                    # ой, кажется, нужно снова выполнить проход
                    is_sorted = False
                node = node.next

            """это сброс, но если мы не выполняли перестановку -
            список отсортирован"""
            if is_sorted: break
```

Я добавил дополнительные комментарии, чтобы вам было легче следовать за мной, сравнивая то, что я здесь сделал, с псевдокодом. Также обратите внимание, что страница в Википедии использует структуру данных, совершенно отличную от двусвязного списка. Код в Википедии предполагает некий вид структуры массива или списка. Строки вроде

```
if A[i-1] > A[i] then
```

вы должны перевести на язык Python с использованием двусвязного списка:

```
if node.prev.value > node.value:
```

Мы не можем просто получить доступ к двусвязному списку, так что нужно конвертировать эти операции с индексом массива в `.next` и `.prev`. Кроме того, нужно следить, чтобы при прохождении атрибуты `next` или `prev` были `None`. Такие превращения предполагают большой объем работы с переводом, изучением и угадыванием семантики читаемого псевдокода.

## Изучаем пузырьковую сортировку

Теперь уделите время изучению этого переведенного мной кода Python. Чтобы получить об этом более глубокое представление, посмотрите видео, где я осуществляю перевод в реальном времени. Также схематически изобразите такую сортировку списков различных типов (уже отсортированных, с элементами в случайном порядке, дубликатов и так далее). Когда у вас появится понимание того, как я это сделал, изучите фреймворк `pytest` и алгоритм сортировки слиянием:

test_sorting.py

```
import sorting
from dllist import DoubleLinkedList
from random import randint

max_numbers = 30

def random_list(count):
    numbers = DoubleLinkedList()
    for i in range(count, 0, -1):
        numbers.shift(randint(0, 10000))
    return numbers


def is_sorted(numbers):
    node = numbers.begin
    while node and node.next:
        if node.value > node.next.value:
            return False
        else:
            node = node.next

    return True


def test_bubble_sort():
    numbers = random_list(max_numbers)

    sorting.bubble_sort(numbers)

    assert is_sorted(numbers)

```

```
32
33    def test_merge_sort():
34        numbers = random_list(max_numbers)
35
36        sorting.merge_sort(numbers)
37
38        assert is_sorted(numbers)
```

Важным моментом этого тестового кода является то, что я использую функцию `random.randint`, чтобы сгенерировать случайные данные для тестирования. Такой тест не охватывает много пограничных случаев, но это лишь начало, мы будем совершенствовать его позднее. Помните, что вы еще не реализовывали сортировку слиянием `sorting.merge_sort`, так что можете либо не вписывать эту тестовую функцию, либо пока превратить ее в комментарий.

Когда напишете этот тест и код, снова изучите страницу в Википедии и попытайтесь реализовать какие-нибудь другие версии пузырьковой сортировки, прежде чем переходить к сортировке слиянием.

## Сортировка слиянием

Я пока не совсем готов оставлять вас в одиночестве. Вы должны будете повторить эту процедуру для сортировки слиянием, но сейчас я хочу, чтобы вы попытались решить алгоритм, основываясь только на псевдокоде со страницы в Википедии (**en.wikipedia.org/wiki/Merge_sort**), прежде чем посмотреть, как это сделал я. Существует несколько возможных реализаций, но я использовал вариант «сверху вниз»:

```
function merge_sort(list m)
    if length of m ≤ 1 then
        return m

    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right
```

```
    left := merge_sort(left)
    right := merge_sort(right)

    return merge(left, right)

function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

Допишите оставшуюся тестовую функцию для `test_merge_sort`, а затем попытайтесь это реализовать. Я дам вам одну подсказку – данный алгоритм лучше всего работает, если задан только первый узел. Вероятно, вам также понадобится способ подсчитать количество узлов, имея только данный узел. Двусвязный список такой возможности не предоставляет.

## Плутовство при сортировке слиянием

Если ваша попытка затянулась и вам хочется сплутовать, посмотрите, что сделал я:

sorting.py

```
1     def count(node):
2         count = 0
3
```

```
4           while node:
5               node = node.next
6               count += 1
7
8           return count
9
10
11      def merge_sort(numbers):
12          numbers.begin = merge_node(numbers.begin)
13
14          # ужасный способ получить конец
15          node = numbers.begin
16          while node.next:
17              node = node.next
18          numbers.end = node
19
20
21      def merge_node(start):
22          """Сортирует список чисел путем сортировки слиянием."""
23          if start.next == None:
24              return start
25
26          mid = count(start) // 2
27
28          # проход до середины
29          scanner = start
30          for i in range(0, mid-1):
31              scanner = scanner.next
32
33          # серединный узел сразу после точки прохода
34          mid_node = scanner.next
35          # выход в средней точке
36          scanner.next = None
37          mid_node.prev = None
38
39          merged_left = merge_node(start)
40          merged_right = merge_node(mid_node)
41
42          return merge(merged_left, merged_right)
43
44
45
46      def merge(left, right):
```

```
47          """Выполняет слияние двух списков."""
48          result = None
49
50          if left == None: return right
51          if right == None: return left
52
53          if left.value > right.value:
54              result = right
55              result.next = merge(left, right.next)
56          else:
57              result = left
58              result.next = merge(left.next, right)
59
60          result.next.prev = result
61          return result
```

Я использовал бы этот код как шпаргалку, куда можно быстро подглядывать во время реализации. На видео я пытаюсь заново реализовать этот код с чистого листа, так что вы увидите, как я борюсь с теми же проблемами, что, вероятно, возникают и у вас.

### Быстрая сортировка

Наконец пришла ваша очередь реализовать быструю сортировку и создать для нее набор тестов `test_quicksort`. Я советую сначала реализовать простую быструю сортировку при помощи обычного списка Python. Так вы лучше поймете принцип. Затем возьмите получившийся простой код Python и примените его к двусвязному списку. Помните, что заданию следует уделить достаточно времени и, разумеется, создайте много тестов и проведите тщательную отладку.

## Практические задания

1. Эти реализации – определенно не лучшие, когда дело доходит до производительности. Попытайтесь написать несколько отвратительных тестов, подтверждающих это. Вам, наверное, понадобится дать алгоритмам громадный список. Проведите исследование и выясните,

какие существуют «патологические» (наихудшие) случаи. К примеру, что произойдет, если вы передадите алгоритму быстрой сортировки уже отсортированный список?

2. Пока не вносите никаких усовершенствований, но исследуйте различные улучшения для этих алгоритмов.

3. Найдите другие алгоритмы сортировки и попытайтесь реализовать их.

4. Они работают с односвязным списком? Как насчет очереди и стека? Это полезно?

5. Почитайте о теоретической скорости этих алгоритмов. Вы встретите выражения `O(n^2)` или `O(n log n)`, говорящие о том, что в худшем случае алгоритмы будут работать настолько медленно. Определение «"О" большого» для алгоритмов выходит за рамки данной книги, но мы коротко обсудим это в упражнении 18.

6. Я реализовал эти алгоритмы как отдельный модуль, но разве не было бы проще добавить их в двусвязный список в виде функций? Если вы так и сделаете, нужно ли будет копировать этот код в другие структуры данных, с которыми он может работать? Мы не принимаем конструктивное решение о том, как заставить эти алгоритмы сортировки работать с любой «структурой данных, подобной связанному списку».

7. Никогда больше не используйте пузырьковую сортировку. Я упомянул ее, поскольку вы будете часто сталкиваться с ней в плохом коде, а также поскольку мы будем совершенствовать ее производительность в упражнении 19.

# Словарь

Вам должен быть знаком тип данных Python под названием «словарь». Каждый раз, когда вы пишете что-то, вроде этого

```
cars = {'Toyota': 4, 'BMW': 20, 'Audi': 10}
```

вы используете словарь, связывая модели автомобилей (`'Toyota'`, `'BMW'`, `'Audi'`) с их количеством (`4`, `20`, `10`). Использование этой структуры данных для вас сейчас уже должно быть обычной практикой, так что вы, вероятно, даже не задумываетесь, как она работает. В данном упражнении вы это узнаете, реализовав собственный словарь при помощи уже созданных вами структур данных. Вашей целью будет реализовать свою версию словаря, основываясь на приведенном мной коде.

## Задача упражнения

В данном упражнении вам нужно будет полностью задокументировать и понять фрагмент приведенного мной кода, а затем как можно быстрее написать по памяти собственную его версию. Предназначение упражнения – научиться анализировать и понимать сложные фрагменты кода. Кроме того, важно усвоить и запомнить, как создается простая структура данных, вроде словаря. Я считаю, что лучший способ научиться анализировать и понимать фрагмент кода – это его реализация на основе собственного исследования и запоминания.

Рассматривайте это как урок по «мастер-копии». Этот термин происходит из изобразительного искусства, когда вы пытаетесь скопировать картину кого-то более мастеровитого. Во время этого процесса вы понимаете, каким образом рисовал другой человек, и так улучшаете свои навыки. Кодинг и рисование схожи в том, что вся необходимая информация находится в открытом доступе, так что вы легко можете научиться у кого-нибудь другого, просто скопировав их работу.

## «Мастер-копия» кода

Для создания «мастер-копии» кода следуйте этой процедуре, которую я называю CASMIR.

1. Скопируйте (*Copy*) код и проверьте его работу. Ваша копия должна быть в точности такой же. Это поможет понять код и заставит вас изучить его подробнее.

2. Проаннотируйте (*Annotate*) код комментариями, составьте анализ всего кода, удостоверившись, что понимаете смысл и функцию каждой строки. Этот этап может включать переключение на другой написанный вами код для «связывания» воедино всей концепции.

3. Подведите итоги (*Summarize*) структуры при помощи кратких примечаний о том, благодаря чему этот код работает. Это будет список функций и то, что делает каждая из них.

4. Запомните (*Memorize*) это краткое описание алгоритма и ключевые фрагменты кода.

5. Реализуйте (*Implement*) по памяти все, что сможете, а когда больше нечего будет вспомнить, вернитесь к исходному коду и запискам, чтобы запомнить больше.

6. Повторите (*Repeat*) этот процесс столько раз, сколько вам нужно, чтобы воспроизвести код из памяти. Ваша сделанная по памяти копия не должна быть совершенно точной, но она должна быть близкой к оригиналу и успешно проходить тест.

Это даст вам более глубокое понимание того, как работает структура данных, но, что более важно, поможет вам усвоить и вспомнить, что эта структура данных делает. Вы сможете понять концепцию, а также реализовать структуру данных, когда понадобится. Это также натренирует ваш мозг лучше запоминать другие структуры данных и алгоритмы в будущем.

---

**Внимание!** Единственное, о чем я хочу вас предупредить, – это была очень наивная, глупая и медленная реализация словаря. Вы просто делаете копию упрощенного глупого словаря, которая содержит все основные элементы и работает, но требует значительных улучшений для запуска «в производство». Эти улучшения будут сделаны, когда мы дойдем до упражнения 19 и научимся

управлять производительностью. А сейчас реализуйте эту простую версию, чтобы понять основы структуры данных.

## Скопируйте код

Сначала взглянем на код словаря, копию которого вы будете создавать:

dictionary.py

```
from dllist import DoubleLinkedList

class Dictionary(object):
    def __init__(self, num_buckets=256):
    """Инициализирует карту (Map) с заданным количеством
    корзин (buckets)."""
    self.map = DoubleLinkedList()
    for i in range(0, num_buckets):
        self.map.push(DoubleLinkedList())

    def hash_key(self, key):
        """ С данным ключом это создаст число
        и преобразует его в индекс корзин на карте."""
        return hash(key) % self.map.count()

    def get_bucket(self, key):
       """С данным ключом найти соответствующую корзину."""
        bucket_id = self.hash_key(key)
        return self.map.get(bucket_id)

    def get_slot(self, key, default=None):
        """
        Возвращает либо корзину и узел для слота, либо
        None, None
        """
        bucket = self.get_bucket(key)

       if bucket:
          node = bucket.begin
          i = 0

          while node:
              if key == node.value[0]:
```

```
                return bucket, node
            else:
                node = node.next
                i += 1

        # не охватывается конструкциями if и while выше
        return bucket, None

    def get(self, key, default=None):
        """Получает значение в корзине для данного
        ключа, или значение по умолчанию."""
        bucket, node = self.get_slot(key, default =default)
        return node and node.value[1] or node

    def set(self, key, value):
        """Присваивает ключ значению, заменяя любое
        существующее значение."""
        bucket, slot = self.get_slot(key)

        if slot:
            # ключ существует, заменить его
            slot.value = (key, value)
        else:
            # ключ не существует, присоединить, создав
            bucket.push((key, value))

    def delete(self, key):
        """Удаляет заданный ключ с карты."""
        bucket = self.get_bucket(key)
        node = bucket.begin

    while node:
            k, v = node.value
            if key == k:
                bucket.detach_node(node)
                break

    def list(self):
        """Выводит то, что есть на карте."""
        bucket_node = self.map.begin
        while bucket_node:
            slot_node = bucket_node.value.begin
            while slot_node:
```

```
77                      print(slot_node.value)
78                      slot_node = slot_node.next
79                  bucket_node = bucket_node.next
```

Этот код реализует словарь с использованием вашего кода двусвязного списка. Если вы не полностью понимаете двусвязный список, тогда вы должны попытаться выполнить процедуру мастер-копии кода. Как только вы будете уверены, что понимаете двусвязный список, можете ввести этот код и начать работать с ним. Помните, прежде чем добавлять аннотации, убедитесь, что это идеальная копия. Худшее, что вы можете сделать, это аннотировать неправильно работающую копию моего кода.

Чтобы помочь вам лучше понять этот код, я написал быстрый и неаккуратный тестовый сценарий:

test_dictionary.py

```
1   from dictionary import Dictionary
2
3   # создать соответствие названий штатов и их аббревиатур
4   states = Dictionary()
5   states.set('Oregon', 'OR')
6   states.set('Florida', 'FL')
7   states.set('California', 'CA')
8   states.set('New York', 'NY')
9   states.set('Michigan', 'MI')
10
11  # создать базовый набор штатов и некоторых их городов
12  cities = Dictionary()
13  cities.set('CA', 'San Francisco')
14  cities.set('MI', 'Detroit')
15  cities.set('FL', 'Jacksonville')
16
17  # добавить еще немного городов
18  cities.set('NY', 'New York')
19  cities.set('OR', 'Portland')
20
21
22  # вывести некоторые города
23  print('-' * 10)
24  print("NY State has: %s" % cities.get('NY'))
25  print("OR State has: %s" % cities.get('OR'))
26
```

```
# вывести некоторые штаты
print('-' * 10)
print("Michigan's abbreviation is: %s" % states. get('Michigan'))
print("Florida's abbreviation is: %s" % states. get('Florida'))

# сделать это, используя словарь штатов, а затем городов
print('-' * 10)
print("Michigan has: %s" % cities.get(states. get('Michigan')))
print("Florida has: %s" % cities.get(states. get('Florida')))

# вывести аббревиатуру каждого штата
print('-' * 10)
states.list()

# вывести все города в штате
print('-' * 10)
cities.list()

print('-' * 10)
state = states.get('Texas')

if not state:
  print("Sorry, no Texas.")

# для значений по умолчанию ||= со значением nil
# можете сделать это одной строкой?
city = cities.get('TX', 'Does Not Exist')
print("The city for the state 'TX' is: %s" % city)
```

Я хочу, чтобы этот код вы тоже ввели точно как есть, но когда вы перейдете к следующему этапу мастер-копии, вы превратите это в официальный автоматизированный тест, который можете запустить с помощью `pytest`. Пока что просто выполните этот сценарий, чтобы заставить работать класс `Dictionary`, а затем, на следующем этапе, вы приведете все это в порядок.

## Добавьте аннотации

Убедитесь, что сделанная вами копия полностью совпадает с моим кодом и что она успешно проходит тест. Затем можете приступать к добавлению

аннотаций и изучению строк кода, выясняя, что делает каждая из них. Отличный способ осуществить это – написать «официальный» автоматизированный тест и добавлять аннотации по ходу работы. Используйте сценарий `dictionary_test.py` и конвертируйте каждый раздел в небольшую тестовую функцию, затем аннотируйте класс `Dictionary` по мере необходимости.

К примеру, первый раздел теста в `test_dictionary.py` создает словарь и совершает серию вызовов `Dictionary.set`. Я превратил бы это в функцию `test_set`, а затем добавил бы аннотацию к функции `Dictionary.set` в файле `dictionary.py`. Чтобы аннотировать функцию `Dictionary.set`, вам придется проникнуть в функцию `Dictionary.get_slot`, затем в функцию `Dictionary.get_bucket` и, в конце концов, в функцию `Dictionary.hash_key`. Так вы организованно проаннотируете и поймете значительную часть класса `Dictionary`, используя лишь один тест.

## Подведите итоги структуры данных

Теперь вы можете подвести итоги того, что вы поняли из аннотирования кода в `dictionary.py` и переписывания файла `dictionary_test.py` таким образом, чтобы превратить его в настоящий автоматизированный тест `pytest`. Ваши итоги должны представлять собой четкое и краткое описание структуры данных. Если вы можете вместить их на клочке бумаги, значит, вы делаете все правильно. Не все структуры данных можно представить в таком сжатом виде, но сохранение краткости поможет вам в процессе запоминания. Можете использовать диаграммы, рисунки, слова и все, что запомните.

Предназначение этого этапа – снабдить вас быстрыми записками, куда можно поместить больше подробностей при следующем запоминании. Итоги не должны включать в себя все, но им следует содержать отдельные кусочки информации, работающие как спусковой крючок, вызывая в памяти код из этапа аннотирования, на котором вам необходимо вызывать в памяти код из этапа запоминания. Это называется «дробление»: вы связываете более подробные воспоминания с небольшими фрагментами информации. Держите это в уме, когда подводите итоги. Чем меньше эти фрагменты, тем лучше, но следует избегать излишнего дробления.

## Запомните итоги

Вы будете запоминать итоги и осуществлять аннотацию так, как вам удобно, но я опишу базовый процесс запоминания, которым вы можете воспользоваться в первую очередь. Честно говоря, запоминание сложных вещей – это одинаковый для всех естественный процесс проб и ошибок, но некоторые приемы могут пригодиться.

1. Убедитесь, что у вас есть блокнот и распечатки итогов и кода.

2. Проведите три минуты, просто читая итоги и пытаясь их запомнить. Молча смотрите на них, прочитайте вслух, затем закройте глаза и повторите то, что прочли, и даже попробуйте запомнить «форму» слов на бумаге. Это звучит безумно, но, поверьте мне, это работает. Ваш мозг лучше запоминает фигуры, чем это может показаться.

3. Переверните лист с итогами и попробуйте записать их по памяти, а когда окажетесь в тупике, быстренько подсмотрите. Бросив на записи беглый взгляд, снова переверните их и попробуйте продолжить.

4. Как только вы напишете копию итогов по памяти (в основном по памяти), используйте их, чтобы на протяжении еще трех минут попытаться запомнить аннотированный код. Просто прочитайте часть итогов, а затем взгляните на соответствующую часть кода и попытайтесь запомнить ее. Вы даже можете тратить по три минуты на каждую маленькую функцию.

5. Когда вы попытались запомнить аннотированный код, переверните его и, пользуясь итогами, попытайтесь записать код в блокнот. Опять же, когда застреваете, быстро переверните аннотацию и подглядите.

6. Продолжайте делать это, пока не создадите достойную копию кода на бумаге. Этот ваш код не должен быть идеальным, но он должен быть очень близким к исходному коду.

Вам может показаться, что это неосуществимо, но вы будете удивлены, сколько всего вспомните, когда приступите. Как вы только завершите работу, вы удивитесь и тому, насколько хорошо понимаете концепцию словаря. Это не механическое запоминание, но, скорее, построение карты концепции, которую вы можете использовать, когда пытаетесь самостоятельно реализовать словарь.

**Внимание!** Если вы беспокоитесь о запоминании чего-либо, это упражнение здорово поможет вам в будущем. Способность следовать процедуре запоминания чего-то помогает преодолеть любые расстройства при запоминании темы. Вместо того чтобы зацикливаться на неудаче, вы будете наблюдать постепенное улучшение в ходе последовательного процесса. Во время работы вы увидите способы усовершенствования процесса вспоминания и примените эти усовершенствования. Вам просто нужно поверить мне – эта методика изучения кажется медленной, но в конце концов она оказывается куда быстрее других.

## Реализуйте по памяти

Теперь время садиться за компьютер, оставляя свои бумажные заметки в другой комнате или на полу, и пытаться осуществить первую реализацию по памяти. Ваша первая попытка может закончиться полной катастрофой, но в этом нет ничего страшного. Скорее всего, вы не привыкли реализовывать что-либо по памяти. Просто выложите все, что помните, а когда в памяти ничего не останется, вернитесь в другую комнату и запомните еще что-нибудь. После нескольких таких походов вы войдете в процесс, и воспоминания «потекут» лучше. Нет ничего страшного в том, чтобы снова и снова ходить смотреть на заметки. Все дело в попытках сохранить воспоминания о коде и улучшить свои навыки.

Я рекомендую сначала записать все, что приходит в голову, будь то тест, код или и то и другое. Затем реализуйте то, что можете вспомнить, или вспоминайте другие части кода. Если вы помните имя функции `test_set` и несколько строк кода, то запишите их. Сразу же воспользуйтесь тем, что они у вас на уме. После этого используйте этот тест, чтобы вспомнить или реализовать функцию `Dictionary.set` как можно лучше. Ваша цель – использовать любую информацию, с помощью которой вы можете реализовать другую информацию.

Чтобы реализовать код, вы также должны попытаться использовать свое понимание словаря. Не пытайтесь просто фотографически воспроизвести каждую строку. Это на самом деле невозможно, поскольку ни у кого нет фотографической памяти (поищите, ни у кого). У большинства людей нормальная память со срабатывающими механизмами вызова концептуальных понимания, которые можно использовать. Чтобы создать собственную копию, вы должны сделать то же самое и использовать ваши знания о том, как работает словарь. В случае с приведенным выше примером вы знаете, что `Dictionary.set` функционирует определенным образом и что вам понадобится

способ получить слоты и корзины... значит, вам нужны функции `get_slot` и `get_bucket`. Вы не вспоминаете каждый символ; вы запоминаете все ключевые концепции и используете их.

## Повторение

Важнейший момент этого упражнения заключается в том, чтобы рассматривать повторение процесса не как неудачу, но как способ улучшить себя. Вы будете делать так со всеми остальными структурами данных в этой книге, у вас будет много практики. Если вам нужно 100 раз возвращаться и запоминать, все в порядке. Вскоре вам станет достаточно 50 раз, а потом 10, и в конце концов вы сможете реализовать словарь по памяти, не прикладывая усилий. Просто продолжайте предпринимать попытки и воспринимайте это как медитацию, пытаясь расслабиться.

## Практические задания

1. Мои тесты очень ограниченны. Напишите более обширный тест.

2. Как алгоритмы сортировки из упражнения 16 могли бы пригодиться в случае с этой структурой данных?

3. Что происходит, когда вы производите рандомизацию ключей и значений в этой структуре данных? Помогает ли алгоритм сортировки?

4. Какое влияние на структуру данных оказывает `num_buckets`?

## Ломаем это

Возможно, вы поджарили себе мозги, но сделайте паузу, а затем попытайтесь «сломать» этот код. Эта реализация крайне ненадежна и перегружена данными. Как насчет странных пограничных случаев? Можете ли вы добавлять в качестве ключа что угодно или только строки? Вызовет ли это проблемы? Наконец, можете ли вы использовать какие-нибудь хитрости, чтобы код выглядел так, будто он работает нормально, но при этом в него особым умным способом были внесены ошибки?

# Измерение производительности

В данном упражнении вы научитесь пользоваться инструментами для анализа производительности создаваемых структур данных и алгоритмов. Чтобы это вступление было конкретным и необъемным, мы рассмотрим производительность алгоритмов `sorting.py` из упражнения 16, а затем на видео я проанализирую производительность всех структур данных, которыми мы занимались до сих пор.

Анализ производительности и соответствующая настройка – одно из моих любимых занятий в программировании. Я тот парень, который будет сидеть перед телевизором с клубком спутанной пряжи и просто распутывать ее, пока все не станет красиво и аккуратно. Мне нравится решать сложные загадки, а производительность кода – одна из самых сложных загадок. Существуют также полезные инструменты для анализа производительности кода, они делают эту процедуру намного приятнее по сравнению с отладкой.

В процессе написания кода не пытайтесь вносить улучшения производительности, если только они не очевидны. Я предпочитаю, чтобы исходная версия моего кода была очень простой и «наивной», тогда я могу гарантировать, что она работает правильно. Затем, когда код работает хорошо, но, возможно, медленно, я вытаскиваю свои инструменты для профилирования и начинаю искать способы ускорить работу, не уменьшая стабильность. Эта последняя часть является ключевой, так как многие программисты считают, что снижать стабильность и безопасность их кода – нормально, если это ускоряет его работу.

## Инструменты

В данном упражнении мы рассмотрим множество различных инструментов, полезных для Python, а также некоторые общие стратегии повышения производительности любого кода. Инструменты, которыми мы будем пользоваться, следующие:

- `timeit` (**docs.python.org/3/library/timeit.html**)

- `cProfile` и `profile` (**docs.python.org/2/library/profile.html**)

Перед продолжением убедитесь, что вы установили все необходимое. Затем сделайте копии файлов `sorting.py` и `test_sorting.py`, чтобы мы могли применить каждый из этих инструментов к данным алгоритмам.

## `timeit`

Модуль `timeit` не слишком полезен. Все, что он делает, это берет строку Python и запускает ее, подсчитывая время выполнения. Вы не можете передавать ему ссылки на функции, файлы с расширением `.py` или что-либо другое, кроме строки. Мы можем проверить, как долго работает функция `test_bubble_sort`, дописав следующий код в конце файла `test_sorting.py`:

```
if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test_bubble_sort()",
        setup="from __main__ import test_bubble_sort"))
```

Этот инструмент также не предоставляет полезных измерений или какой-либо информации о том, почему что-то может работать медленно. Нам нужен способ измерения времени выполнения фрагментов кода, так что `timeit` просто слишком неудобен, чтобы быть полезным.

## `cProfile` и `profile`

Следующие два инструмента гораздо более полезны для измерения производительности кода. Для анализа времени выполнения вашего кода я рекомендую использовать `cProfile`, а `profile` оставить до момента, когда вам понадобится больше гибкости. Чтобы запустить `cProfile` в вашем тесте, измените нижнюю часть файла `test_sorting.py` так, чтобы запускались тестовые функции:

```
if __name__ == '__main__':
    test_bubble_sort()
    test_merge_sort()
```

И выставьте значение `max_numbers` равным примерно `800` или числу, достаточно большому, чтобы вы могли оценить эффект. После этого запустите `cProfile` в своем коде:

```
$ python -m cProfile -s cumtime test_sorting.py | grep
sorting.py
```

Я использую `| grep sorting.py` просто чтобы ограничить вывод файлами, которые меня интересуют, но вы должны отбросить эту часть команды и просмотреть полный вывод. Вот результаты, полученные на моем довольно быстром компьютере для 800 чисел:

```
    ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
         1    0.000    0.000    0.145    0.145  test_sorting.py:1(<module>)
         1    0.000    0.000    0.128    0.128  test_sorting.py:25     \
                                                (test_bubble_sort)
         1    0.125    0.125    0.125    0.125  sorting.py:6(bubble_sort)
         1    0.000    0.000    0.009    0.009  sorting.py:1(<module>)
         1    0.000    0.000    0.008    0.008  test_sorting.py:33     \
                                                (test_merge_sort)
         2    0.001    0.000    0.006    0.003  test_sorting.py:7(random_list)
         1    0.000    0.000    0.005    0.005  sorting.py:37(merge_sort)
  1599/1      0.001    0.000    0.005    0.005  sorting.py:47(merge_node)
7500/799      0.004    0.000    0.004    0.000  sorting.py:72(merge)
       799    0.001    0.000    0.001    0.000  sorting.py:27(count)
         2    0.000    0.000    0.000    0.000  test_sorting.py:14     \
                                                (is_sorted)
```

Я добавил сверху заголовки, чтобы вы понимали, о чем говорит этот вывод. Каждый заголовок значит следующее:

**ncalls** – количество вызовов этой функции;

**tottime** – время, затраченное на выполнение функции;

**percall** – время, затраченное на каждый вызов этой функции;

**cumtime** – совокупное время, потраченное в этой функции;

**percall** – совокупное время на каждый вызов;

**filename:lineno(function)** – имя файла, номер строки кода и название функции.

Эти имена заголовков также являются опциями, доступными для параметра `-s`. Затем мы можем быстро проанализировать этот вывод.

- Функция `bubble_sort` вызывается один раз, но `merge_node` вызывается 1599 раз, а `merge` еще больше – 7500 раз. Это связано с тем, что функции `merge_node` и `merge` являются рекурсивными, поэтому, сортируя случайный список из 800 элементов, они будут производить огромное количество вызовов.

- Несмотря на то что `bubble_sort` вызывается гораздо меньше, чем `merge` или `merge_node`, она намного медленнее. Это соответствует ожиданиям производительности двух алгоритмов. Наихудший случай для сортировки слиянием – `O(n log n)`, но для пузырьковой сортировки это `O(n ^ 2)`. Если у вас 800 элементов, значит, `800 log 800` составляет около 5347, тогда как `800 ^ 2` равно 640 000! Эти числа не обязательно указывают на точное число секунд, за которое выполняются алгоритмы, но их можно использовать для относительного сравнения.

- Функция `count` вызывается 799 раз, что, скорее всего, значительная трата времени. Наша реализация двусвязного списка не отслеживает количество элементов, вместо этого выполняя проход по списку каждый раз, когда вы хотите узнать это количество. Мы используем тот же метод в функции `count` здесь, и это приводит к 799 проходам по всему списку из 800 элементов. Замените `max_numbers` на 600 или 500, чтобы увидеть закономерность. Обратили внимание, что `count` выполняется `n-1` раз? Это значит, что мы проходим почти все 800 элементов.

Теперь давайте также посмотрим, как числа `dllist.py` влияют на эту производительность:

```
$ python -m cProfile -s cumtime test_sorting.py | grep dllist.py
  ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
  1200    0.000    0.000    0.001    0.000    dllist.py:66(shift)
  1200    0.001    0.000    0.001    0.000    dllist.py:19(push)
  1200    0.000    0.000    0.000    0.000    dllist.py:3(__init__)
     1    0.000    0.000    0.000    0.000    dllist.py:1(<module>)
     1    0.000    0.000    0.000    0.000    dllist.py:1              \
                                               (DoubleLinkedListNode)
     2    0.000    0.000    0.000    0.000    dllist.py:15(__init__)
     1    0.000    0.000    0.000    0.000    dllist.py:13             \
                                               (DoubleLinkedList)
```

Я снова добавил заголовки, чтобы вам было понятно, что происходит. Здесь вы можете видеть, что функции `dllist.py` влияют на производительность не так уж и сильно, по сравнению с функциями `merge`, `merge_node` и `count`. Это важно, потому что большинство программистов будут оптимизировать структуру данных двусвязного списка, где можно получить больше от `merge_sort`, и проигнорируют `bubble_sort`. Всегда начинайте с того, где вы можете осуществить самое значительное улучшение с минимальными усилиями.

## Анализируем производительность

Анализ производительности сводится к выяснению, что выполняется медленно, и попыткам определить, почему это выполняется медленно. Процесс похож на отладку, за исключением того, что вы изо всех сил стараетесь не изменять поведение кода.

Когда вы закончите, код должен работать точно так же, только быстрее. Бывают случаи, когда исправление производительности помогает обнаружить ошибки, но лучше не пытаться полностью перепроектировать код, работая над его ускорением.

Еще одна важная вещь, которую нужно иметь при себе, прежде чем начинать анализировать производительность, – это показатели того, что требуется от программного обеспечения. Высокая скорость, как правило, всегда нужна, но без ориентира вы в итоге придете к совершенно ненужным решениям. Если ваша система выполняет 50 запросов в секунду, а вам в действительности нужно всего 100 запросов в секунду, тогда нет смысла предлагать полное переписывание на Haskell[5], чтобы получить 200 запросов. Этот процесс целиком сводится к «сэкономить как можно больше денег, приложив как можно меньше усилий», и вам нужно ориентироваться на какие-то измерения.

Большую часть этих измерений вы можете получить у операционного отдела компании; у них должны быть хорошие графики, показывающие использование ЦП, соотношение запросов и затраченных секунд, частоту кадров и все, что они считают важным или что считают важным клиенты. Затем вы можете поработать с ними, разрабатывая тесты, на которых будет видна потеря скорости, и затем вы сможете улучшить код для достижения желаемых целей. Возможно, вы выжмете из системы больше производительности, сэкономив деньги. После попытки сделать это, вы можете прийти к выводу, что это просто сложная задача, требующая больше ресурсов ЦП. Располагая

[5] Чистый функциональный язык программирования. – *Прим. ред.*

показателями, на которые можно ориентироваться, вы получите представление о том, когда нужно отказаться от дальнейшей работы или когда уже было сделано достаточно.

Простейший процесс, который позволит вам проанализировать производительность, заключается в следующем.

1. Запустите анализатор производительности в коде, как я делал здесь с вашими тестами. Чем больше информации вы можете получить, тем лучше. Информацию о дополнительных бесплатных инструментах см. в разделе «Дальнейшее обучение». Узнайте, какими инструментами пользуются другие люди для анализа скорости системы.

2. Определите самые медленные и наименьшие фрагменты кода. Не беритесь за анализ гигантской функции. Зачастую функции медленны, потому что используют кучу других медленных функций. Найдя сначала самую медленную и самую маленькую, вы, скорее всего, получите лучший результат при наименьших усилиях.

3. Сделайте обзор кода этих медленных фрагментов и всего, к чему они имеют отношение. Ищите возможные причины, по которым код выполняется медленно. Циклы расположены внутри циклов? Слишком много вызовов функции? Поищите что-то простое, прежде чем переходить к сложным приемам вроде кэширования.

4. Когда вы составите список всех самых медленных и самых маленьких функций, а также простых изменений, благодаря которым эти функции станут выполняться быстрее, ищите закономерности. Делаете ли вы то же самое где-нибудь еще, где этого не видно?

5. Наконец, если в маленькие функции нельзя внести простые изменения, приступайте к поиску возможных значительных улучшений. Может, действительно настало время для полного переписывания? Не делайте этого до тех пор, пока, по крайней мере, не попробуете внести простые исправления.

6. Составьте список всех изменений, что вы вносили, и всех полученных приростов производительности. Если вы этого не сделаете, то будете постоянно возвращаться к функциям, над которыми уже работали, тратя усилия впустую.

По мере такой работы ваше представление о «самой медленной и самой маленькой функции» изменится. Вы исправите и ускорите дюжину 10-строчных

функций, а это значит, что тогда вы сможете приняться за ту 100-строчную функцию, которая осталась самой медленной. Как только вы ускорите ту 100-строчную функцию, можете обратить внимание на более крупные группы функций и разрабатывать стратегии для их ускорения.

Наконец, лучший способ ускорить что-либо – не делать это вообще. Если вы несколько раз проверяете одно и то же условие, найдите способ не делать этого более одного раза. Если вы несколько раз подсчитываете один и тот же столбец в базе данных, делайте это один раз. Если вы вызываете функцию в сплошном цикле, но данные редко меняются, проверьте их сохранение в памяти или предварительно вычислите таблицу. Во многих случаях вы можете заменить хранение расчетами, просто однократно производя вычисления и сохраняя их результат.

В следующем упражнении мы будем использовать данный процесс, чтобы по-настоящему улучшить производительность этих алгоритмов.

## Задача упражнения

Ваше задание в этом упражнении – применить то, что я сделал с функциями `bubble_sort` и `merge_sort`, ко всем структурам данных и алгоритмам, которые вы создали до сих пор. Я пока не ожидаю, что вы улучшите их, но просто делайте заметки и анализируйте производительность, разрабатывая тесты, которые будут демонстрировать проблемы в этом компоненте. Не поддавайтесь искушению исправить что-либо прямо сейчас, потому что улучшать производительность мы будем в упражнении 19.

## Практические задания

1. Запустите эти инструменты профилирования для всего имеющегося у вас кода и проанализируйте его производительность.

2. Сравните полученные результаты с теоретическими результатами для алгоритмов и структур данных.

## Ломаем это

Попробуйте написать «патологические» тесты, которые сломают структуры данных. Возможно, вам придется скормить им огромные объемы данных, но используйте данные профилирования, чтобы убедиться, что вы все делаете правильно.

## Дальнейшее обучение

1. Взгляните на line_profiler (**github.com/rkern/line_profiler**), это еще один инструмент измерения производительности. Его преимущество заключается в том, что вы можете измерять только те функции, которые для вас важны. Его недостаток — для этого вы должны изменить свой исходник.

2. pyprof2calltree (**pypi.python.org/pypi/pyprof2calltree**) и KCacheGrind (**kcachegrind.github.io/html/Home.html**) являются более продвинутыми инструментами, но работают они только в Linux. На видео я показываю, как их использовать под Linux.

# Повышение производительности

В этом упражнении главное – видеоматериал, на котором я продемонстрирую улучшение производительности ранее написанного вами кода, но сначала вы сами должны попытаться сделать это. В упражнении 18 вы осуществляли анализ скорости выполнения кода, так что теперь пришло время реализовать некоторые ваши идеи. Я дам вам краткий список того, что нужно искать и что нужно изменять в процессе устранения простых проблем с производительностью.

1. Циклы внутри циклов, повторяющие вычисления, которых можно избежать. Пузырьковая сортировка в этом смысле – классический случай, и именно поэтому я ее описывал. Как только вы увидите, насколько ужасна пузырьковая сортировка в сравнении с другими методами, вы станете рассматривать ее как шаблон, которого следует избегать.

2. Повторяющиеся вычисления того, что на самом деле не меняется или может быть рассчитано один раз во время изменения. Отличным примером этого случая является использование функции `count()` из `sorting.py`. Вы можете отслеживать количество внутри функций в структуре данных. Каждый раз, когда вы добавляете элемент, увеличивайте счетчик и уменьшайте его каждый раз, когда удаляете элемент. Нет необходимости проходить весь список снова и снова. Вы также можете использовать этот способ, чтобы усовершенствовать логику других функций, проверяя условие `count == 0`.

3. Использование неправильной структуры данных. В словаре я использую двусвязный список для демонстрации этой проблемы. Словарь должен иметь произвольный доступ к элементу, по крайней мере, в списке корзин. Использование двусвязного списка с двусвязным списком внутри предполагает, что каждый раз, когда вы хотите получить доступ к *n*-му элементу, вам приходится проходить все элементы до *n*. Использование взамен списка Python значительно улучшит производительность. Это было упражнение в использовании вашего существующего кода для создания структуры данных из более простых

структур данных – не обязательно упражнение в создании самого лучшего словаря Python (таковой уже есть).

4. Использование неправильных алгоритмов в структурах данных. Пузырьковая сортировка, очевидно, плохой алгоритм (никогда не используйте ее снова), но вспомните, насколько хороши сортировка слиянием и быстрая сортировка, в зависимости от структуры данных. Сортировка слиянием отлично подходит для таких типов связанных структур данных, но она не так хороша для массивов вроде списка Python. Быстрая сортировка лучше работает со списком, но хуже – со связанными структурами данных.

5. Отсутствие оптимизации общих операций. В двусвязном списке вы часто начинаете в начале корзины и ищете слоты по значению. В текущем коде эти слоты просто добавляются по мере их поступления, и этот процесс может быть случайным или нет. Если бы вы применили сортировку этих списков при вставке, нахождение элемента стало бы проще и быстрее. Вы могли бы остановиться, когда значение слота больше, чем то, что вы ищете, ведь вы знаете, что все уже отсортировано. Это замедляет вставки, но ускоряет почти все остальные операции, поэтому следует проектировать правильно. Если вам нужно делать множество вставок, тогда это неразумно. Но если ваш анализ показывает, что вы делаете несколько вставок, но много обращений, то это способ ускорить работу.

6. Ручное написание вместо использования существующего кода. Мы выполняем упражнения, чтобы изучить структуры данных, но в реальном мире вы этого делать не будете. В Python уже есть отличные структуры данных, они встроены в язык и оптимизированы. Сначала вы должны использовать их, и только если анализ производительности свидетельствует о том, что ваша собственная структура данных будет быстрее, тогда можете писать сами. Но даже в таком случае вместо создания собственной структуры данных вы должны поискать уже существующую, чья работоспособность была проверена кем-либо. В этом упражнении напишите несколько тестов, которые сравнивают ваши словарь и списки с соответствующими встроенными типами данных Python, чтобы увидеть, насколько вы далеки от истины.

7. Использование рекурсии в языке, который для этого не годится. Код `merge_sort` можно сломать, просто передав ему список, больший, чем стек Python. Попробуйте передать ему что-то безумное, вроде 3000 элементов, а затем медленно уменьшайте это число до тех пор,

пока не найдете ту самую точку, в которой происходит переполнение стека. В Python нет определенных рекурсивных оптимизаций, поэтому не следует прибегать к рекурсии без особых причин. В этом случае переписывание `merge_sort` с целью использования цикла будет лучше (но гораздо сложнее).

Это главные проблемы, которые вы должны были обнаружить во время анализа в упражнении 18. Теперь ваша задача – попытаться произвести улучшения и повысить производительность этого кода.

## Задача упражнения

Попытайтесь использовать свой анализ и описанные выше предложения, чтобы методически повышать производительность вашего кода. «Методически» означает, что это делается согласно жесткому регламенту, с использованием данных для подтверждения того, что вы действительно улучшили ситуацию. Вот процедура, которой нужно следовать во время этого упражнения.

1. Выберите свой первый самый маленький и самый медленный фрагмент кода и убедитесь, что у вас есть тест, показывающий, насколько он медленный. Убедитесь, что у вас есть серия измерений, которые дают вам представление о скорости. Постройте соответствующий график, если сможете.

2. Внесите свои изменения по повышению скорости, затем повторите тест. Продолжайте пытаться выжать максимум производительности из этого фрагмента кода.

3. Если ваши изменения не улучшили положение вещей, тогда либо выясните, что вы сделали не так, либо отмените внесенное изменение и попробуйте что-то еще. Это важно, поскольку вы работаете над гипотезой, так что, если оставите бесполезное изменение кода, это может повлиять на производительность других функций. Откатите изменения и попробуйте другой подход или перейдите к другому фрагменту кода.

4. Повторно осуществите измерение других самых маленьких и самых медленных фрагментов кода, чтобы увидеть, изменились ли показания. Ваши исправления могли повлиять на другой код, так что перепроверьте уже известные данные.

5. Как только вы внесете все изменения, снова запустите измерение и выберите новые фрагменты кода, которые можно попытаться усовершенствовать.

6. Храните результаты тестов, начиная с шага 1 (это должны быть автоматизированные тесты), поскольку вам нужно избежать ухудшений работы. Если вы видите, что внесенное в функцию изменение приводит к замедлению работы других функций, тогда либо исправьте это изменение, либо просто отмените его и попробуйте новый подход.

## Дальнейшее обучение

Вы должны изучить исходное электронное письмо Python Timsort (**mail.python.org/pipermail/python-dev/2002-July/026837.html**) и ошибку, обнаруженную в 2015 году (**bugs.python.org/issue23515**) исследователями из EU FP7 ENVISAGE (**envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/**). Исходное письмо отправлено в 2002 году, и код был впоследствии реализован. Прошло 13 лет, прежде чем эту ошибку обнаружили. Имейте это в виду, когда станете реализовать собственные идеи для алгоритмов. Даже лучшие разработчики в больших проектах допускают ошибки в алгоритмах, остающиеся скрытыми в течение очень долгого времени. Другим примером может служить проект OpenSSL, в котором десятилетиями скрывались ошибки, просто потому что все считали, что код создали «профессиональные шифровальщики». Оказывается, даже предположительно профессиональные шифровальщики могут писать ужасный код. Правильное создание новых алгоритмов требует специальных навыков и, я полагаю, использования инструмента для автоматического доказательства теорем, чтобы подтверждать правильность. Если у вас нет этого фундамента, создавать новые алгоритмы и структуры данных вы можете лишь на свой страх и риск. То же касается криптографических алгоритмов и зашифрованных сетевых протоколов. Реализация алгоритмов, как было доказано, совершенно замечательное и полезное упражнение, пока вы полностью осведомлены о своих навыках в реализации. Но не стоит опрометчиво придумывать собственные структуры данных, не прибегая к помощи.

# Двоичные деревья поиска

В этом упражнении я научу вас превращать обычное описание структуры данных в рабочий код. Вы уже знаете, как анализировать код алгоритма или структуры данных, используя метод мастер-копии. Вы также знаете, как читать описание алгоритма в виде псевдокода. Теперь вы объедините их и научитесь использовать свободное описание двоичного дерева поиска.

Я сразу хочу предупредить вас – не заходите на страницу в Википедии во время работы над этим упражнением. Описание двоичного дерева поиска в Википедии в основном содержит рабочий код Python для структуры данных, что противоречит замыслу этого упражнении. Если вы окажетесь в тупике, можете использовать любые доступные ресурсы, но сначала попробуйте выполнить задание исходя только из моих описаний.

## Особенности ДДП

В упражнении 16 вы видели, как сортировка слиянием принимает плоский связный список и преобразует его в дерево из отсортированных частей. Она нарезает список на мелкие куски, а затем собирает их вместе, помещая меньшие куски «слева» и большие справа. В некотором смысле двоичное дерево поиска (ДДП) представляет собой структуру данных, которая сразу же выполняет такую сортировку и никогда не пытается держать элементы в списке. Главное применение ДДП в упорядочивании пар ключ=значение по принципу дерева заблаговременно, как только вы добавляете или удаляете пары.

Двоичное дерево поиска делает это, начиная с корневого узла ключ=значение и прокладывая путь вправо или влево. Если вы вставляете новую пару ключ=значение, задание дерева – начать с корня и сравнивать ключ с каждым узлом, двигаясь влево в случае, если новый ключ меньше, и вправо – если ключ больше или равен. В конце концов, ДДП находит нужную позицию. Все операции после этого делают то же самое, сравнивая ключи с каждым узлом, двигаясь влево и вправо, пока либо не будет найден узел, либо поиск не зайдет в тупик.

В этом смысле двоичное дерево поиска является альтернативой словарю из упражнения 17, следовательно, у него должны быть аналогичные операции. Для создания древовидной структуры базовому узлу ДДП нужны атрибуты

`left`, `right`, `key` и `value`. Вам также может понадобиться атрибут `parent` в зависимости от того, как вы работаете. Для работы с двоичным деревом поиска нужны следующие операции над корневым узлом.

**`get`**

С заданным ключом, обойдите дерево, чтобы найти узел, или верните `None`, если зайдете в тупик. Вы двигаетесь влево, если данный ключ меньше, чем ключ узла. Двигаетесь вправо, если ключ больше, чем ключ узла. Если вы достигли узла, после которого невозможно пойти ни вправо, ни влево, значит, на этом все – нужного узла не существует. Есть способ сделать это, используя рекурсию и цикл `while`.

**`set`**

Это почти то же самое, что и `get`, за исключением того, что когда вы достигаете этого тупикового узла, то просто присоединяете новые узлы слева или справа, тем самым добавляя снизу к дереву еще одну ветвь.

**`delete`**

Удаление узлов из двоичного дерева поиска – сложная операция, поэтому я посвятил ей целый раздел. Вкратце: у вас есть три условия: узел – лист (без детей), у узла есть один ребенок или у узла есть двое детей. Если это просто лист, удалите его. Если у него есть один ребенок, замените узел ребенком. Если же у него двое детей, тогда все становится очень сложным, поэтому прочтите раздел об удалении ниже.

**`list`**

Обойдите дерево и выведите все. Важный момент в этой операции состоит в том, что вы можете обходить дерево по-разному и осуществлять разный вывод. Если вы двигаетесь по левому пути, а затем по правому, то получите нечто отличное от того, что получили бы, если бы двигались в обратном порядке. Если вы проходите весь путь до конца и затем выводите, поднимаясь по дереву в направлении к корню, тогда получите еще один отдельный вывод. Вы также можете выводить узлы по мере продвижения вниз по дереву, от корня до «листов». Опробуйте различные способы, чтобы понять, как работает каждый из них.

# Удаление

Помните, что при удалении узла (который я буду называть D) у нас есть три возможных варианта.

1. Узел D является «листовым» узлом, поскольку он не имеет детей (ни слева, ни справа). Просто удалите его.

2. У узла D есть только один ребенок (либо слева, либо справа, но не в обоих направлениях). В этом случае вы можете просто переместить значение этого ребенка в узел D, а затем удалить ребенка. Это эффективно заменит узел D ребенком (или «передвинет ребенка вверх»).

3. У узла D есть как левый, так и правый ребенок, а это значит, что пришло время для серьезного хирургического вмешательства. Сначала найдите минимального ребенка узла справа от D (`D.right`); его называют преемником (`successor`). Присвойте ключ D (`D.key`) ключу преемника (`successor.key`), а затем аналогичным образом удалите детей этого преемника, используя его ключ.

Скорее всего, вам также понадобится операция для поиска минимального узла (`find_minimum`) и для замены родительского узла (`replace_node_in_parent`). Я упоминал, что вам может понадобиться родительский атрибут, в зависимости от того, как вы будете осуществлять реализацию. Я бы сказал, что нужно воспользоваться родительским узлом, так как в большинстве случаев это проще.

---

**Внимание!** Никто не любит удалять из дерева. Это сложная операция, и даже в моем любимом руководстве — книге «Алгоритмы. Руководство по разработке» Стивена С. Скиены (Steven S. Skiena)[6], она пропущена, поскольку реализация «выглядит немного ужасно». Не падайте духом, если освоение удаления из дерева дается вам с трудом.

---

# Задача упражнения

Вы должны реализовать свое двоичное дерево поиска, используя это умышленно расплывчатое описание. Старайтесь не использовать другие источники,

[6] На русском языке книга издана в 2017 году в издательстве БХВ-Петербург. – *Прим. ред.*

пока предпринимаете первую попытку, а затем, когда зайдете в тупик, узнайте, как с этим справились другие. Ваша цель в данном упражнении – попытаться решить сложную проблему исходя из заведомо ужасного ее описания.

Фокус в том, чтобы сначала перевести абзацы человеческого языка на грубый псевдокод. Затем этот грубый псевдокод нужно превратить в более точный псевдокод. Тогда вы можете перевести более точный псевдокод на язык Python. Обращайте особое внимание на конкретные слова, поскольку одно слово в нашем языке может значить разные понятия в Python. Иногда просто нужно сделать предположение и запустить тесты, чтобы убедиться в его правильности.

Тесты также будут очень важны, и было бы неплохо применить к этой проблеме метод «сначала тест». Вы знаете, что должна делать каждая из этих операций, так что можете написать для нее тест, а затем выполнить его.

## Практические задания

1. Можете ли вы разработать «патологический» тест, осуществляющий вставки таким образом, что двоичное дерево поиска становится не более чем причудливым связным списком?

2. Что происходит, когда вы пытаетесь удалять из такого двоичного дерева?

3. Насколько высока скорость ДДП по сравнению с вашим недавно оптимизированным словарем?

4. Насколько быстрым вы можете сделать ДДП при помощи анализа и повышения производительности?

# Двоичный поиск

Алгоритм двоичного поиска – это простой способ найти элемент в уже отсортированном списке. Проще всего описать этот алгоритм как процесс разбиения отсортированного списка на половины, который продолжается, пока либо не будет найден нужный элемент, либо список не закончится. Если вы полностью выполнили упражнение 20, тогда данное упражнение должно оказаться для вас относительно простым.

Если бы мы захотели найти число X в уже отсортированном списке чисел, алгоритм был бы следующий.

1. Возьмите число в середине списка (M) и сравните его с X.
2. Если X == M, задание выполнено.
3. Если X > M, найдите середину отрезка от M+1 до конца списка.
4. Если X < M, то найдите середину отрезка от M-1 до начала списка.
5. Повторяйте шаги, пока либо не найдете X, либо у вас не останется всего 1 элемент.

Это работает с любыми данными, которые можно сравнивать. Алгоритм будет работать со строками, числами и всем остальным, что поддается последовательному упорядочиванию.

## Задача упражнения

У вашего двоичного дерева поиска должна быть операция `get`. Она похожа на двоичный поиск. Разница в том, что ДДП уже разделено на части, поэтому больше этого делать нет необходимости. В данном упражнении вы реализуете двоичный поиск для двусвязного списка, списка Python и сравните его с производительностью операции `get` для ДДП. Ваша цель – выяснить следующее.

1. Насколько хорошо ДДП подходит для простого поиска некоторых элементов в сравнении со списком Python?

2. Насколько плохо бинарный поиск работает с двусвязным списком?

3. Могут ли ваши патологические (наихудшие) случаи для ДДП также вызвать проблемы у бинарного поиска в списке Python?

При анализе производительности не включайте время, необходимое для сортировки чисел. Оно является важным при выполнении глобальной оптимизации, но в этом случае нас волнует только то, насколько быстро работает двоичный поиск. Поскольку это не главное, для сортировки вашего списка вы также можете использовать встроенные алгоритмы сортировки Python. Это упражнение предназначено для выяснения того, насколько быстро осуществляется поиск в трех структурах данных.

## Практические задания

1. Узнайте, какое максимальное количество возможных сравнений необходимо выполнить этому алгоритму. Сначала попытайтесь разобраться в этом самостоятельно, а затем исследуйте алгоритм, чтобы узнать настоящий ответ. После этого просто запомните этот ответ.

2. Можете ли вы применить какую-либо оптимизацию к алгоритмам сортировки?

3. Попробуйте визуализировать то, как этот алгоритм работает в каждой структуре данных. Например, в случае с двусвязным списком вы фактически можете представлять, как алгоритм ходит туда-обратно, пока не найдет решение.

4. Чтобы повысить сложность, попробуйте сделать из двусвязного списка отсортированный связный список, где каждая вставка всегда осуществляется на отсортированную позицию. Теперь проведите анализ производительности, включив добавление элементов и сортировку списков чисел, чтобы увидеть, как это улучшает общую производительность.

## Дальнейшее обучение

Изучите другие алгоритмы поиска, особенно для строк. Многие из них будет трудно реализовать на Python из-за того, как здесь работают строки, но в любом случае попробуйте.

# Суффиксные массивы

Я хотел бы рассказать вам историю, связанную с суффиксными массивами. Я проходил собеседование в компании в Сиэтле в то время, когда мне было любопытно, как можно наиболее эффективно создать diff для исполняемого двоичного кода. Мои исследования привели меня к алгоритмам суффиксного массива и суффиксного дерева. Суффиксный массив – это просто массив, где вы сортируете все суффиксы строки в отсортированный список. Суффиксное дерево похоже на суффиксный массив, но больше напоминает двоичное дерево поиска, чем список. Эти алгоритмы были довольно простыми, а их производительность во время сортировки – высокой. Проблема, которую они решали, заключалась в поиске самой длинной подстроки, общей для двух строк (или в данном случае списков байтов).

На Python суффиксный массив создается очень просто:

Упражнение *22* – сеанс *Python*

```
>>> magic = "abracadabra"
>>> magic_sa = []
>>> for i in range(0, len(magic)):
...     magic_sa.append(magic[i:])
...
>>> magic_sa
['abracadabra', 'bracadabra', 'racadabra', 'acadabra',
 'cadabra', 'adabra', 'dabra', 'abra', 'bra', 'ra', 'a']
>>> magic_sa = sorted(magic_sa)
>>> magic_sa
['a', 'abra', 'abracadabra', 'acadabra', 'adabra', 'bra',
 'bracadabra', 'cadabra', 'dabra', 'ra', 'racadabra']
>>>
```

Как видите, я просто по порядку взял суффиксы строки, а затем отсортировал список. Но что мне это дало? Как только я получаю этот список, я могу найти любой желаемый суффикс, просто выполнив в списке двоичный поиск. Это очень грубый пример, но в настоящем коде это делается очень быстро, и вы можете отслеживать все исходные индексы, чтобы затем ссылаться на исходные местоположения суффиксов. Это очень быстро по сравнению с другими алгоритмами поиска и очень полезно в таких областях, как анализ ДНК.

Вернемся к собеседованию в Сиэтле. Я сижу в этой холодной комнате и прохожу собеседование с программистами C++ на должность Java. Как вы можете представить, это не слишком веселое собеседование, и я уверен, что не получу работу. Я не писал на C++ уже много лет, а должность предполагала знание языка Java, в котором я был тогда экспертом. Заходит следующий интервьюер и спрашивает меня: «Как найти подстроку в строке?»

Отлично! В свободное время я очень подробно изучал эту проблему. Сейчас я всем покажу! Я вскакиваю, подхожу к доске и объясняю тому парню, как сделать суффиксное дерево, как это улучшит производительность поиска, как измененная сортировка кучи повышает скорость работы, как работает суффиксное дерево, почему это лучше, чем троичное дерево поиска, и как это осуществить на C. Я полагаю, что если смогу продемонстрировать, как сделать это на C, то докажу, что я не просто знаток Java, не имеющий значительного опыта за плечами.

Интервьюер ошеломлен. Как будто я прямо в комнате для собеседований открыл сумку со свежими дурианами. Он смотрит на доску и бормочет: «Хм, я ожидал услышать что-то об алгоритме поиска Бойера – Мура. Вам он знаком?» Я слегка скривился и сказал: «Да, но это было где-то 10 лет назад». Он качает головой, берет свои вещи и встает, говоря: «Хорошо, я дам всем знать, что думаю».

Через несколько минут заходит следующий интервьюер. Он смотрит на доску, издевательски усмехается мне, затем задает еще один вопрос по метапрограммированию шаблонов C++, на который я не смог ответить. Работу я не получил.

## Задача упражнения

В данном упражнении вы возьмете мой небольшой пример Python и создадите собственный класс поиска суффиксного массива. Этот класс будет принимать строку, превращать ее в список суффиксов, затем разрешать над ним следующие операции.

**`find_shortest`**

> Находит кратчайшую подстроку с таким началом. В приведенном выше примере, если я ищу abra, тогда он должен возвращать abra, а не abracadabra.

**`find_longest`**

Находит самую длинную подстроку с таким началом. Если я ищу abra, он должен возвращать abracadabra.

**`find_all`**

Находит все подстроки с таким началом. Это означает, что abra возвращает как abra, так и abracadabra.

Для этого вам будет нужен хороший автоматизированный тест, а также некоторые измерения производительности. Мы будем использовать их в последующих упражнениях. Как только вы закончите, переходите к практическим заданиям для завершения этого упражнения.

# Практические задания

1. После того как вы проведете свои тесты, перепишите это, чтобы использовать ДДП для сортировки и поиска суффиксов. Вы также можете использовать значение каждого узла ДДП, чтобы отслеживать, где в исходной строке существует эта подстрока. Можете затем держать под рукой исходную строку.

2. Как ДДП меняет ваш код в зависимости от разных операций поиска? Оно делает его проще или сложнее?

# Дальнейшее обучение

Вы определенно должны изучить суффиксные массивы и их приложения. Они невероятно полезны, но не слишком хорошо известны большинству программистов.

УПРАЖНЕНИЕ 23

# Троичные деревья поиска

Последняя структура данных, которую мы рассмотрим, называется «троичное дерево поиска» (ТДП), и она полезна для быстрого поиска строки в наборе строк. Она похожа на двоичное дерево поиска, но вместо двух детей здесь их трое, и каждый ребенок представляет собой только один символ, а не целую строку. В ДДП у вас был левый и правый ребенок, которые располагались соответственно на ветвях «меньше, чем» и «больше, чем». В случае с ТДП есть левая, средняя и правая ветви, обозначающие «меньше, чем», «равен» и «больше, чем». Это позволяет взять строку, разбить ее на символы, а затем обойти ТДП по одному символу за раз, пока он не будет найден или пока дерево не закончится.

ТДП эффективно обменивает пространство на скорость, разбивая набор возможных ключей, среди которых нужно осуществить поиск, на узлы из одного символа. Каждый из этих узлов занимает больше места, чем те же ключи в ДДП, но это позволяет вам находить ключи, сравнивая символы только в желаемом ключе. В ДДП вам нужно было сравнивать большинство символов в искомом ключе и в ключе узла для каждого узла. В ТДП вы сравниваете только каждую букву искомого ключа, и когда вы добираетесь до конца, то работа на этом заканчивается.

Еще одно преимущество ТДП в том, что эта структура данных позволяет узнать, когда ключ отсутствует в наборе. Представьте, что у вас есть ключ длиной 10 символов, и вам нужно найти его в наборе других ключей, но при этом также нужно быстро остановиться, если ключа там нет. При помощи ТДП вы можете остановиться на одном или двух символах, дойти до конца дерева и понять, что этого ключа не существует. В крайнем случае, вы сравните всего 10 символов в ключе, а это намного меньше сравнений, чем в ДДП. В худшем случае ДДП (когда оно в общем представляет собой связный список) могло сравнивать все символы в 10-символьном ключе по разу для каждого отдельного узла, прежде чем решить, что ключ не существует.

## Задача упражнения

В этом упражнении вы частично выполните еще одну мастер-копию, а затем самостоятельно завершите ТДП. Вот код, который вам понадобится:

tstree.py

```
class TSTreeNode(object):

    def __init__(self, key, value, low, eq, high):
        self.key = key
        self.low = low
        self.eq = eq
        self.high = high
        self.value = value


class TSTree(object):

    def __init__(self):
        self.root = None

    def _get(self, node, keys):
        key = keys[0]
        if key < node.key:
            return self._get(node.low, keys)
        elif key == node.key:
            if len(keys) > 1:
                return self._get(node.eq, keys[1:])
            else:
                return node.value
        else:
            return self._get(node.high, keys)

    def get(self, key):
        keys = [x for x in key]
        return self._get(self.root, keys)

    def _set(self, node, keys, value):
        next_key = keys[0]

        if not node:
            # что случится, если вы добавите значение сюда?
            node = TSTreeNode(next_key, None, None,
            None, None)

        if next_key < node.key:
            node.low = self._set(node.low, keys, value)
        elif next_key == node.key:
```

```
43                  if len(keys) > 1:
44                      node.eq = self._set(node.eq, keys[1:], value)
45                  else:
46                      # а если вы НЕ ДОБАВИТЕ значение сюда?
47                      node.value = value
48              else:
49                  node.high = self._set(node.high, keys, value)
50
51              return node
52
53          def set(self, key, value):
54              keys = [x for x in key]
55              self.root = self._set(self.root, keys, value)
```

Вам нужно изучить этот код, используя знакомый вам метод мастер-копии. Обратите особое внимание на то, как обрабатывается путь `node.eq` и как устанавливается значение `node.value`. После того как вы поймете работу функций `get` и `set`, реализуйте оставшиеся функции и выполните все тесты. Следует реализовать следующие функции.

**`find_shortest`**

С данным ключом K находит кратчайшую пару ключ/значение, начинающуюся с K. Это означает, что если в вашем наборе есть apple и application, тогда вызов `find_shortest("appl")` вернет ключ apple и связанное с ним значение.

**`find_longest`**

С данным ключом K находит самую длинную пару ключ/значение, начинающуюся с K. Возвращаясь к примеру с apple и application, вызов `find_longest("appl")` вернет application и значение, связанное с ним.

**`find_all`**

С данным ключом K находит все пары ключ/значение, которые начинаются с K. Я сначала реализовал бы эту функцию, а затем перешел к `find_shortest` и `find_longest`.

**`find_part`**

С данным ключом K находит кратчайший ключ, который содержит хотя бы часть начала K. Изучите, как это работает в зависимости от того, где вы устанавливаете значение узла.

# Практические задания

1. Взгляните на комментарии в исходном коде и проверьте, куда вы помещаете значение (в `_set`). Если вы измените это, поменяется ли смысл функции `get` и почему?

2. Испытайте ТДП случайными изменениями и произведите измерения производительности.

3. В ТДП вы также можете применить частичное совпадение. Это дополнительное задание, так что попробуйте реализовать и посмотрите, что получится. Частичное совпадение – это совпадение типа a.p.e с apple, anpxe и ajpqe.

4. Как можно искать окончания строк? Подсказка: не переусердствуйте с этим.

УПРАЖНЕНИЕ 24

# Быстрый поиск по URL

Мы закончим раздел о структурах данных и алгоритмах заданием на измерение производительности, которое позволит применить ваши структуры данных к реальной проблеме. В свое время я написал несколько веб-серверов, и проблема, которая с ними постоянно возникает, – это сопоставление URL-адресов и «действий». Вы обнаружите эту проблему в каждом веб-фреймворке, веб-сервере и везде, где нужно «маршрутизировать» информацию на основе иерархического ключа. Когда ваш веб-сервер получает URL `/do/this/stuff/`, он должен выяснить, привязана ли каждая часть к какому-либо действию или конфигурации. Если вы настроили параметры конфигурации веб-приложения для `/do/`, что веб-сервер должен делать с `/this/stuff/`? Должен ли он игнорировать эту часть или передать ее веб-приложению? Что делать, если в `/do/this/` находится каталог? И как можно быстро обнаруживать неверные URL-адреса, не обрабатывая огромные несуществующие запросы?

Проблема с таким иерархическим поиском возникает достаточно часто, так что это будет достойное заключительное испытание вашей способности применять алгоритмы и структуры данных на практике, равно как и способности выполнять анализ производительности.

## Задача упражнения

Во-первых, убедитесь, что вы понимаете, что такое URL-адреса и как они используются. Если не понимаете, предлагаю вам уделить время написанию одного маленького приложения Flask, использующего сложную маршрутизацию. Эту маршрутизацию вы и будете реализовывать.

Далее вы должны сделать следующее.

1. Создайте простой базовый класс `URLRouter`, который поделите на подклассы для всех ваших реализаций. Для этого класса `URLRouter` вы должны иметь возможность сделать следующее:

   1) добавить новый URL-адрес, связанный с объектом;

2) получить точное соответствие URL-адреса. Поиск `/do/this/stuff/` возвращает только то, что находится именно по этому адресу;

3) получить лучшее соответствие для URL. Поиск `/do/this/stuff/` вернет `/do/`, если это единственное совпадение;

4) получить все объекты, начинающиеся с этого URL;

5) получить кратчайший URL, совпадающий с объектом. Поиск `/do/this/stuff/` вернет `/do/`, но не `/do/this/`;

6) получить самый длинный URL, совпадающий с объектом. Поиск `/do/this/stuff/` вернет `/do/this/`, но не `/do/`.

2. Создайте и протестируйте подкласс класса `URLRouter`, используя троичное дерево поиска, так это будет сделать проще всего. Обязательно проверьте следующее:

   1) рандомизированные URL-адреса и пути различной длины как в ТДП, так и в том, что вы ищете;

   2) поиск только частичных путей в разных ситуациях;

   3) полностью несуществующие пути;

   4) по-настоящему длинные пути, существующие и несуществующие.

3. После того как этот подкласс заработает и будет протестирован, обобщите свой тест, чтобы его можно было запускать для всех реализаций.

4. Затем попытайтесь осуществить реализации, использующие двусвязный список, двоичное дерево поиска и словарь Python. Убедитесь, что ваш обобщенный тест работает с каждой из этих структур данных.

5. После этого начните анализировать производительность каждой из этих реализаций для различных операций.

Цель состоит в том, чтобы узнать, насколько быстрым является ТДП в сравнении с другими структурами данных. Возможно, ТДП будет быстрее большинства из них но, скорее всего, словарь сэкономит больше всего времени, так как он оптимизирован для Python. Можете даже сделать ставку на то, производительность какой структуры данных окажется наивысшей для каждой операции.

## Практические задания

1. Я обошел вниманием суффиксный массив, поскольку он похож на ТДП, но для его использования вам придется добавить те же операции. Сделайте это и оцените скорость работы суффиксного массива.

2. Исследуйте, как это делает ваш любимый веб-сервер или веб-фреймворк. Вы обнаружите, что небольшая часть людей, работающих с URL-адресами, знает, что такое троичное дерево поиска – несмотря на то, насколько оно здесь полезно.

## Дальнейшее обучение

Если хотите углубиться в алгоритмы и структуры данных, настоятельно рекомендую книгу «Алгоритмы. Руководство по разработке» Стивена С. Скиены. В его книге используется язык C, поэтому вам, вероятно, вам сначала потребуется прочесть *Learn C the Hard Way*, чтобы понять ее. Кроме того, это очень хорошая книга, так как она охватывает и теорию, и практические аспекты реализации и анализа производительности алгоритмов и структур данных.

# Часть IV

## Проекты следующего уровня

В части III вы изучили основы структур данных и алгоритмов, но, что более важно, вы научились проверять (проводить аудит) и тестировать код. Вы не проверяли и не тестировали собственный код – только мой, ища ошибки или недостатки согласно моим подсказкам. Цель части IV – при помощи набора проектов повышенной сложности научиться проводить аудит собственного кода. Ваша задача на протяжении грядущих пяти проектов заключается в следующем.

1. Проведите 45-минутный сеанс программирования, создав проект и начав работу.

2. Проведите аудит выполненной работы, пользуясь тем, что вы изучили в части III, для нахождения потенциальных дефектов и проблем в своей реализации.

3. В следующие 45 минут приведите код в порядок и придайте ему официальный характер.

4. Проверите аудит этого 45-минутного сеанса, усовершенствовав код.

Единственное различие между этими 45-минутными сеансами и вашей первой партией проектов в том, что теперь вам не нужно так строго контролировать время. 45 минут – это всего лишь напоминание, существующее, чтобы убедиться, что перед проведением аудита вы не засиживаетесь слишком долго. Нет никакого смысла проверять код, прерванный посреди хорошей идеи или реализации. Очевидно, что невозможно провести нормальный аудит наполовину готового кода. Вы должны работать около 45 минут, а когда достигнете точки остановки, тогда и просмотрите, что сделали.

В этом разделе вы должны будете вернуться к контрольным спискам из части III и строго следовать им. Неплохо устраивать перерыв на 10–15 минут, прежде чем приступать к аудиту, – так вы очищаете мысли и переходите в режим критического мышления.

Во время работы над этими проектами я буду предлагать алгоритмы, которые вы можете использовать. Вам не обязательно использовать алгоритмы, которые вы реализовывали, но попытаться стоит, просто чтобы увидеть, как они работают. Очень вероятно, что они не лучше существующих структур данных Python (списков и словарей), поскольку последние были специально настроены для максимально быстрой работы. Тем не менее это хорошее упражнение, чтобы понять, когда нужно использовать алгоритмы и как их проверять.

## Отслеживание ошибок

Наконец, я попрошу вас отслеживать значения совершаемых ошибок. Аналогично тому, что вы делали в части II, когда отслеживали реализованные возможности, здесь вы будете отслеживать количество обнаруженных при аудитах дефектов и то, что это были за дефекты. Заносите в журнал все, что находите, создав таблицу с типом дефекта сверху и временем обнаружения слева. Если хотите, вы также можете сразу строить граф результатов или использовать электронную таблицу. Ваша цель при отслеживании обнаруженных дефектов – понять, какие ошибки вы чаще всего совершаете во время сеансов программирования, и попытаться предотвратить их или просто следить за их появлением во время аудитов.

# Команда `xargs`

Мы возвращаемся к упражнениям повышенной сложности. В качестве разминки предлагаю реализовать команду `xargs`. Для вас это должно быть просто, но задача может усложниться, если для работы `xargs` понадобится запуск других программ. Нужный вам модуль Python называется `subprocess`, он может запускать из Python другие программы и передавать вам их вывод. Вам нужно будет изучить этот модуль для завершения данного упражнения, а также для многих других проектов в этой книге, поэтому постарайтесь как следует.

## Задача упражнения

Реализуйте `xargs` за 45 минут, чтобы получить рабочий код, для которого можно провести аудит. Помните, что во время первого сеанса нужно просто приступить к работе, а не пытаться создать что-то совершенное. В дальнейшем вы будете последовательно улучшать этот проект. Помните, что можно ввести

```
man xargs
```

чтобы получить справку по `xargs` и узнать, как работает эта команда. Это удобный инструмент Unix, но также вы можете использовать `find` практически для тех же целей. По ходу реализации `xargs` попытайтесь выяснить, в чем она превосходит `find -exec`.

После 45-минутного сеанса необходимо сделать перерыв, а затем провести объективный аудит кода, используя контрольный список, описанный в упражнении 13. Не исправляйте код, просто напишите комментарии, указывающие, что нужно изменить и какие ошибки были допущены. Трудно оставаться объективным и одновременно пытаться исправить ситуацию, так что просто отметьте проблемы, а затем исправьте их во время следующего подхода.

Затем вы проведете серию ограниченных по времени сеансов по написанию кода и проведению аудита, чтобы привыкнуть к идее аудитов. Потратьте столько времени, сколько необходимо, чтобы реализовать как можно больше функционала `xargs`, а затем переходите к следующему проекту.

**Внимание!** Не забудьте отслеживать свои ошибки, фиксируя их в журнале, чтобы затем строить графики и отмечать тенденции.

## Практические задания

1. Во время этих сеансов написания кода и проведения аудита обнаружили ли вы что-то, в чем постоянно ошибаетесь? Запишите, над чем в будущем надо поработать.

2. Можно ли в вашем процессе кода/аудита выделить отрезки, на которых вы ошибаетесь чаще или реже? Этих отрезков в начале больше, чем после трех или четырех подходов? Какие могут быть причины этого?

3. Попробуйте написать автоматические тесты для своей реализации `xargs` и посмотреть, не снизят ли они ваш уровень ошибок. В следующем упражнении вы проведете похожее, только более контролируемое исследование, но пока попробуйте этот способ.

# Команда `hexdump`

Вы сделали разминку, реализовав `xargs`, и теперь работаете в режиме написания кода и проведения аудита. Попытайтесь выполнить следующее задание с использованием подхода «сначала тест». Он предполагает, что вы создаете тест, который описывает ожидаемое поведение вашего кода, а затем реализовываете это поведение, пока тест не будет проходиться успешно. Вы будете создавать копию инструмента `hexdump` и пытаться сделать так, чтобы вывод вашей версии этого инструмента совпадал с выводом настоящей версии. Именно здесь действительно поможет разработка с подходом «сначала тест», поскольку так вы автоматизируете процесс имитации другого программного обеспечения.

Этот метод очень полезен, когда вам нужно написать замену ужасному фрагменту программы. В области ПО часто приходится работать над проектами, целью которых является замена старой системы более современной реализацией. Примером может служить замена старой банковской системы на языке Кобол новой замечательной системой на Django. Делается это, как правило, для упрощения поддержки и распространения благодаря использованию чего-то более удобного в работе, чем старая система. Если вы можете написать набор автоматизированных тестов, которые будут проверять поведение старой системы, а затем направить этот набор тестов на новую систему, тогда у вас есть способ подтвердить, что ваша замена эффективна, по большей части. Поверьте, полноценная успешная замена практически неосуществима, но здесь выручает написание автоматизированного теста.

В этом упражнении вы добавите к своему движению следующее.

1. Напишите тестовый пример, запускающий исходный инструмент `hexdump` в сценарии, который вам нужно реализовать. Например, это может быть опция `-C`. Вам нужно будет либо использовать модуль `subprocess` для запуска, либо просто запустить его заблаговременно и сохранить результаты в файл, который вы загружаете.

2. Напишите код, который заставит этот тест работать, путем запуска теста на своей версии `hexdump`, а затем сравните результаты. Если они не идентичны, значит, вы сделали что-то неправильно.

3. Проведите аудит кода теста и собственного кода.

Я выбрал `hexdump` из-за сложности в воссоздании странного формата вывода команды. Нет ничего сложного в том, как она работает. Вам нужно понять только то, как получить правильный вывод. Это поможет вам попрактиковаться в подходе «сначала тест».

---

**Внимание!** Когда я говорю «сначала напишите тест», я не имею в виду, что нужно написать полноценный увесистый файл test.py со всеми функциями и огромным количеством комплексного кода. Я имею в виду то, чему учил вас ранее. Напишите небольшой тестовый пример – возможно, только десятую часть одной тестовой функции – а также код, необходимый для его работы, а затем переключайтесь между ними туда-обратно. Чем больше вы знаете о коде, тем больше тестовых примеров можете написать, но не стоит создавать множество тестов, не имея ничего, на чем их можно было бы опробовать. Вместо этого работайте по нарастающей.

---

# Задача упражнения

Команда `hexdump` полезна, когда вы хотите увидеть содержимое файла, не являющегося доступным для просмотра текстом. Она отображает байты в файле в различных полезных форматах, включая шестнадцатеричный, восьмеричный и ASCII в боковой части вывода. Сложность в реализации вашей собственной команды `hexdump` заключается не в чтении данных и даже не в конвертации их в другие форматы. В Python это легко сделать с помощью функций `hex`, `oct`, `int` и `ord`. Строковые операторы исходного формата также полезны, так как существуют опции для шестнадцатеричного и восьмеричного форматирования с заданной точностью.

Настоящая сложность заключается в правильном форматировании вывода для каждой из разных опций, чтобы он правильно помещался на экран. Вот первые несколько строк вывода `hexdump -C` для файла Python `.pyc`:

```
00000000  03 f3 0d 0a f0 b5 69 57  63 00 00 00 00 00 00 00  |......iWc.......|
00000010  00 03 00 00 00 40 00 00  00 73 3a 00 00 00 64 00  |.....@...s:...d.|
00000020  00 64 01 00 6c 00 00 6d  01 00 5a 01 00 01 64 00  |.d..l..m..Z...d.|
00000030  00 64 02 00 6c 02 00 6d  03 00 5a 03 00 01 64 03  |.d..l..m..Z...d.|
00000040  00 65 01 00 66 01 00 64  04 00 84 00 00 83 00 00  |.e..f..d........|
```

Страница руководства для этого «канонического» форматирования гласит, что это:

1. Отображение ввода в шестнадцатеричном формате. Таким образом, 10 на самом деле не 10 в десятеричной системе, но в шестнадцатеричной. Знакомы ли вы с шестнадцатеричной системой?

2. Шестнадцать шестнадцатеричных байтов, разделенных пробелами, помещенных в два столбца. Это каждый байт, преобразованный в шестнадцатеричный формат. Сколько столбцов представляют один байт?

3. Те же шестнадцать байтов в формате `% _p`, который выглядит как спецификатор преобразования Python, но является особенностью `hexdump`. Вам нужно будет прочесть страницу руководства, чтобы узнать, что это значит.

Далее, `hexdump` также может получать стандартный ввод `stdin`, что дает вам возможность разделить содержимое вертикальной чертой вот так:

```
echo "Hello There" | hexdump -C
```

На моей системе macOS из этого получается следующее:

```
00000000  48 65 6c 6c 6f 20 54 68  65 72 65 0a  |Hello There.|
0000000c
```

Обратили внимание на ту последнюю строку с символом `c`? Думаю, вам нужно выяснить, что это такое.

Именно это форматирование и вывод будут сложными, и ваша задача – воссоздать их как можно лучше. Вот почему в начале этого упражнения сказано, что нужно использовать подход «сначала тест». Таким образом будет намного проще создавать тесты, предоставляющие данные для команды `hexdump`, и сравнивать ее с настоящей `hexdump` до тех пор, пока все не начнет работать.

## Практическое задание

Изучите команду `od` и посмотрите, можно ли повторно использовать ваш код `hexdump` для реализации `od`. Если да, создайте библиотеку, которую используют обе эти команды.

## Дальнейшее обучение

Есть люди, которые выступают только за то, чтобы тест создавался в первую очередь, но я считаю, что ни один метод не может быть эффективным всегда. Я предпочитаю начинать с написания тестов, когда тестирую взаимодействие программного обеспечения с точки зрения пользователя. Я создаю тесты, которые описывают взаимодействие пользователя и программного обеспечения, а затем иду и воплощаю это взаимодействие в самом ПО. Это то, что вы делали в этом упражнении, когда проверяли, как пользователь видит вывод `hexdump` из командной строки.

В случае с другими программистскими заданиями смешно указывать, что именно нужно писать сначала – тест или код. Это чрезвычайно мешает решению проблемы. Автоматические тесты – всего лишь инструменты, а вы – умный человек, у которого есть право использовать эти инструменты каким угодно образом в зависимости от каждой конкретной ситуации. Любой, кто скажет вам обратное, вероятно, невоспитанный человек, который на самом деле не так уж и хорош в разработке программного обеспечения.

# Команда `tr`

Этим упражнением мы продолжаем изучение подхода TDD (test-driven development – разработка через тестирование, то есть разработка, при которой сначала пишется тест). Важно понимать, как реализовывать такой подход, поскольку он много где используется, но, как упоминалось ранее, у него есть свои ограничения. Вы выполните еще одно упражнение с использованием TDD, реализовав команду `tr`. Обязательно удостоверьтесь, что сначала вы пишете тест, затем код и только потом проводите аудит того и другого.

В предыдущем упражнении я сказал вам строить тест и код по возрастающей. Это, как правило, наименее подверженный ошибкам метод разработки, но он не помогает вам лучше анализировать собственный код. В этом упражнении вы будете работать немного по-другому. Вы полностью напишете тест, проведете аудит теста, затем полностью напишете код, проведете аудит кода и подтвердите аудит путем выполнения тестов.

Это означает, что в данном упражнении ваше движение будет следующим.

1. Напишите тестовый пример, пытаясь охватить большую его часть.

2. Проведите аудит тестового примера и убедитесь, что он написан правильно.

3. Запустите тесты, чтобы убедиться, что результата нет, но ищите любые синтаксические ошибки. На данном этапе у вас не должно быть синтаксических ошибок.

4. Напишите код для тестового примера, но не запускайте тест.

5. Проведите аудит своего кода и посмотрите, сколько найдете ошибок до того, как запустите тест.

Вы будете использовать эту процедуру в следующем упражнении, чтобы отслеживать показатели своих аудиторских навыков и навыков тестирования и чтобы сделать процесс написания кода более управляемым.

## Задача упражнения

Инструмент `tr` – это удобный способ переводить потоки символов. Несмотря на то что `tr` сам по себе очень прост, он может делать с символами довольно сложные операции. Например, вы одной строкой можете получить частоту слов, в истории работы (`history`):

```
history | tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq -c | sort -rn
```

Поразительно, но этой одной строкой Дуглас Макилрой пытался доказать, что подобная программа, написанная Дональдом Кнутом, слишком запутанная и длинная. Реализация Кнута составляла «10 страниц» и строила все с нуля. Единственная строка Дугласа добивается того же результата, используя лишь стандартные инструменты Unix. Это демонстрирует мощь использования небольших инструментов Unix и, одновременно, возможности `tr` по переводу текста.

Пользуйтесь man-страницей и любыми другими источниками, чтобы понять, что делает команда `tr`. Существует также проект Python с аналогичным именем, но пока вы не осуществили свою реализацию, лучше не изучайте его. Сравните ваши результаты с этим проектом, как только закончите работу. И не забывайте, что для этого вам понадобится целый проект, и, как я упоминал в начале, тесты должны быть написаны в первую очередь.

## Критика 45-минутного подхода

Я хочу, чтобы вы продолжали использовать 45-минутные временные блоки, но стоит упомянуть об одном существенном критическом замечании относительно такого способа работы: вы не можете попасть в «поток концентрации». Работать на протяжении коротких промежутков времени (как эти) полезно, когда вам нужно прорваться через огромный объем, сохраняя высокий темп. Так происходит, когда работа по-настоящему скучна и не увлекательна. Я заставляю вас использовать 45-минутные блоки только для того, чтобы вы могли поддерживать темп, но мы также воспользуемся ими при группировании показателей вашей работы для последующего анализа.

Но предупреждаю вас, лучшее программирование – это когда вы в ударе, в потоке, когда вы поймали кураж. Когда вы на целые часы глубоко уходите в себя и совершенно теряете счет времени, пока не опомнитесь в пять утра, осознав, что не спали всю ночь. Подобная интенсивная концентрация очень

привлекает меня в программировании, но она действительно устойчива только когда вам очень интересно то, что вы делаете. Когда вам нужно работать над чужой ужасной базой кода, кураж обычно не поймать. Для таких случаев вам нужна другая стратегия, которая задает темп работе и помогает от нее избавиться, не развалившись самому на части. Вот для этого и нужны 45-минутные блоки.

Наконец, один из способов развить способность ловить кураж и поддерживать концентрацию на протяжении часов – это начать с нескольких коротких блоков, а затем постепенно удлинять их, пока вы не сможете работать в течение более длительных периодов времени. Продолжайте использовать 45-минутные сеансы, но если вы просто забудетесь и окажется, что вы просидели над работой длительное время, наслаждайтесь. Никто не скажет, что вы что-то сделали неправильно. На самом деле это нормально.

## Практические задания

1. Как вам такой способ работать? Нравится он вам или нет? Попытайтесь сформулировать почему, а затем прочитайте некоторые современные мнения касательно TDD или его двоюродного брата BDD (behavior driven development – разработка через поведение).

2. Как думаете, вы обнаружили больше или меньше ошибок, сперва проводя аудит собственного кода, а не строя его постепенно? Сделайте предположение и запишите его.

УПРАЖНЕНИЕ 28

# Команда `sh`

Сейчас вы продолжите работать с использованием подхода TDD, но теперь начнете с небольшого программистского задания. Вообще, работая с TDD, лучше не писать тесты сначала, действуя вместо этого следующим образом.

1. Потратьте 45 минут на изучение проблемы. Этот метод называется «шип», он предназначен для сглаживания проблем, с которыми вы можете столкнуться, или исследования того, что вам нужно будет знать.

2. Спланируйте с помощью списка дел то, что вам, вероятно, потребуется реализовать.

3. Превратите этот план в тест TDD.

4. Запустите тест и убедитесь, что он не дает результата.

5. Напишите код для теста, используя то, что вы узнали после первого пункта.

6. Проведите аудит своего кода и теста для подтверждения качества.

Эта последовательность описывает то, что делают фанатики подхода TDD, когда их прижимает проблема, которую они раньше не изучали. Кроме того, гораздо более практично выработать умение быстро запускать мыслительные процессы, занимаясь изучением проблемы, а уже затем серьезно браться за работу. Если кто-нибудь скажет вам, что это не TDD, просто не говорите им, что перед этим вы использовали метод шипа. Они никогда не узнают.

## Задача упражнения

В этом упражнении вы реализуете часть инструмента Unix `sh`. Когда вы пишете код, вы постоянно используете `sh`, так как это то, что работает внутри Терминала (иначе, в PowerShell) и запускает другие ваши программы. Обычно это оболочка `bash`, но это также может быть `fish`, `csh` или `zsh`.

`sh` – слишком объемная для реализации программа, она также поддерживает полноценный язык программирования для автоматизации вашей системы. Мы не будем реализовывать язык программирования, только часть по запуску процессов командной строки.

Для выполнения этой задачи вам понадобятся следующие библиотеки.

1. `subprocess` (**docs.python.org/2/library/subprocess.html**) для запуска других программ.

2. `readline` (**docs.python.org/2/library/readline.html**) для получения ввода от пользователя и поддержки истории.

Вы не создаете Unix `sh` целиком, с пайпингом и всем остальным, но вы могли бы реализовать почти все, кроме языка программирования. Ваша реализация должна предусматривать следующее.

1. Начните с подсказки с помощью `readline` и примите от пользователя команду для запуска.

2. Разберите команду на исполняемый файл и аргументы.

3. Используйте `subprocess`, чтобы запустить команду с аргументами, и контролируйте весь вывод.

Чтобы приступить к работе, можете начать с шипа, изучив либо `readline`, либо `subprocess`, либо и то и другое, в зависимости от того, что кажется вам необходимым или незнакомым. После этого начинайте писать тесты и реализовывать систему.

## Практическое задание

Можете ли вы реализовать пайпы? Это когда вы вводите `history | grep python` и вертикальная черта `|` отправляет вывод `history` как ввод `grep`.

## Дальнейшее обучение

Вы можете ознакомиться с моим проектом python-lust (**github.com/zedshaw/python-lust**), если хотите узнать больше о Unix и управлении ресурсами. Проект не слишком большой, но содержит много маленьких хитростей.

УПРАЖНЕНИЕ 29

# Команды `diff` и `patch`

В конце части IV вы просто примените весь изученный принцип TDD к гораздо более сложному проекту, который может показаться вам непривычным. Вернитесь к упражнению 28, чтобы убедиться, что вы освоили этот принцип и строго ему следуете. При необходимости составьте контрольный список.

---

**Внимание!** Когда вы действительно погружены в работу, строгое следование процессу не очень полезно. Сейчас вы изучаете процесс и работаете над тем, чтобы усвоить его и уметь пользоваться им в реальном мире. Вот почему я строг к тому, как вам следует действовать. Это просто практика. Не будьте фанатиком какого-либо подхода, когда дело доходит до настоящей работы. Цель книги – предоставить вам набор стратегий, позволяющих справляться с заданиями, а не обучать религиозным обрядам.

---

## Задача упражнения

Команда `diff` принимает два файла и создает третий файл (вывод), в котором указываются изменения, произведенные в первом файле для получения второго. Это основа таких инструментов, как `git`, и других инструментов управления версиями. Реализация `diff` в Python довольно тривиальна, поскольку здесь есть библиотека, делающая это за вас, так что вам не нужно работать над алгоритмами (а это может быть очень сложным).

Инструмент `patch` – спутник инструмента `diff`. Он принимает файл `.diff` и применяет его к другому файлу для создания третьего файла. Это позволяет вам брать изменения между двумя файлами при помощи команды `diff`, а затем отправить кому-то этот файл `.diff`. После чего тот человек может использовать свою исходную версию файла и ваш файл `.diff`, чтобы при помощи `patch` воссоздать ваши изменения.

Вот пример для демонстрации работы `diff` и `patch`. У меня есть два файла, `A.txt` и `B.txt`. Файл `A.txt` содержит простой текст; я скопировал его и создал `B.txt` с некоторыми изменениями:

```
$ diff A.txt B.txt > AB.diff
$ cat AB.diff
2,4c2,4
< her fleece was white a mud
< and every where that marry
< her lamb would chew cud
-
> her fleece was white a snow
> and every where that marry went
> her lamb was sure to go
```

Так получается файл `AB.diff`, содержащий отличия между `A.txt` и `B.txt`, которые, как видно, заключаются в восстановлении нарушенной рифмы. Как только вы получили этот `AB.diff`, можно использовать `patch` для применения изменений:

```
$ patch A.txt AB.diff
$ diff A.txt B.txt
```

Эта последняя команда не должна иметь вывода, так как благодаря предшествующей ей команде `patch` содержимое `A.txt` стало аналогичным содержимому `B.txt`.

Реализация этих двух команд должна начинаться с `diff`, так как у вас есть инструмент `diff`, полностью реализованный с помощью Python. Вы можете найти его в конце документации difflib (**docs.python.org/2/library/difflib.html#a-command-line-interface-to-difflib**). Лучше попытайтесь реализовать свою версию и сравнить ее с официальной версией.

Самое приятное в этом упражнении – инструмент `patch`, который Python не реализовывает за вас. Вам нужно прочесть о классе `SequenceMatcher` в `difflib` и, в частности, взглянуть на функцию `SequenceMatch.get_opcodes` (**docs.python.org/2/library/difflib.html#difflib.SequenceMatcher.get_opcodes**). Это ваш единственный, но очень хороший ключ к реализации `patch`.

## Практическое задание

Как еще вы можете использовать комбинацию `diff` и `patch`? Можете ли вы объединить их в один инструмент? Можете ли вы сделать так, чтобы они работали как миниатюрный `git`?

## Дальнейшее обучение

Найдите столько алгоритмов, сколько сможете. Еще одна вещь, которую нужно исследовать, – это то, как работает инструмент `git`.

# Часть V

## Анализ текста

В этой части книги вы изучите обработку текста. В частности, мы рассмотрим начало формального анализа текста. Я не буду вдаваться в подробности теории языка программирования, поскольку это тема для целого университетского курса. Это же всего лишь начало простого и наивного разбора текста таким способом, чтобы вы смогли использовать его во многих программистских ситуациях.

У большинства программистов странные отношения с разбором текста. Ядром всего компьютерного программирования является синтаксический анализ, и это один из наиболее понятных и формализованных аспектов информатики. Анализ данных присутствует в компьютерных технологиях повсеместно. Вы найдете его в сетевых протоколах, компиляторах, электронных таблицах, серверах, текстовых редакторах, рендерах и почти всем, что должно взаимодействовать с человеком или другим компьютером. Даже когда два

компьютера отправляют фиксированный двоичный протокол, все равно существует аспект разбора, несмотря на отсутствие текста.

Я буду учить вас анализу текста по той причине, что это легкая для понимания и надежная техника, дающая достоверные результаты. Когда вы сталкиваетесь с необходимостью правильно обработать определенный ввод и выдать точные ошибки, вы обратитесь к парсеру (синтаксическому анализатору), вместо того чтобы пытаться написать его вручную. Кроме того, как только вы изучите основы синтаксического анализа, вам станет проще изучать новые языки программирования, так как вы сможете понимать их грамматику.

## Введение в покрытие кода

В этой части вы по-прежнему должны пытаться «сломать» и разобрать на части весь код, который напишете. Нововведение части V – концепция покрытия кода. Идея покрытия кода заключается в том, что вы на самом деле понятия не имеете, протестировали ли вы большинство сценариев при написании автоматизированных тестов. Вы могли бы использовать формальную логику и выдвинуть теорию, что охватили все, но, как известно, человеческий мозг просто ужасен в вопросе нахождения изъянов в собственной работе. Вот почему, занимаясь по этой книге, вы используете цикл «создай, затем критикуй». Вам просто-напросто слишком сложно анализировать свое мышление, одновременно пытаясь что-то создать.

Покрытие кода – это способ, по меньшей мере, получить представление о том, что в своем приложении вы протестировали. Он не отыщет все его недостатки, но, по крайней мере, покажет, что вы учли все ветви кода, которые только могли. Без покрытия вы на самом деле не будете знать, протестировали ли каждую ветку. Хорошим примером является обработка ошибок. Большинство автоматизированных тестов проверяют только самые надежные условия и никогда не проверяют обработку ошибок. Когда вы используете покрытие кода, то обнаруживаете все способы, которыми могли проверить обработку ошибок.

Покрытие кода также помогает избежать чрезмерного тестирования кода. Я работал над проектами с энтузиастами разработки через тестирование (TDD), которые гордились своим показателем соотношения тестов к коду в 12/1 (что означает 12 строк тестов на каждую строку кода). Обычный анализ покрытия кода показал, что они проверяли лишь 30% кода, и многие из этих строк были протестированы одним и тем же способом от 6 до 20 раз. В то же

время простые вещи, такие как исключительные ситуации в запросах базы данных, вообще не были проверены и вызывали частые ошибки. В конце концов эти наборы тестов становятся обузой, не дающей проекту расти, и просто съедают человеческие ресурсы. Неудивительно, что многие консультанты по Agile ненавидят покрытие кода.

На видео к этому упражнению показано, как я запускаю тесты и использую покрытие кода, чтобы подтвердить то, что тестирую. Вы должны сделать то же самое. Существуют инструменты, благодаря которым сделать это будет просто. Я покажу вам, как читать результаты покрытия и как убедиться, что вы эффективно проверяете все, что можете. Цель состоит в том, чтобы получить тщательный автоматизированный набор тестов, не прилагая лишних усилий и не проверяя только 30% кода 12 раз подряд.

УПРАЖНЕНИЕ 30

# Конечные автоматы

Каждый раз, когда вы читаете книгу о синтаксическом разборе, вы натыкаетесь на ту страшную главу о конечных автоматах, где подробно описываются «ребра» и «узлы». При этом каждая комбинация возможных «автоматов» преобразуется в другие автоматы, и, откровенно говоря, это немного чересчур. Существует более простое объяснение конечных автоматов (finite state machines, FSM), делающее их практичными и понятными, не нарушая при этом чистоту теории. Конечно, вас не позовут на работу в Ассоциацию вычислительной техники, поскольку вы не знаете всей математической базы конечных автоматов, но если вы просто хотите использовать их в своих приложениях, то здесь нет ничего сложного.

Конечный автомат – это способ организации событий, происходящих с набором состояний. Событие можно определить как «триггер ввода», аналогичный логическим выражениям в конструкции `if`, но обычно менее сложный. В качестве события может выступать нажатие кнопки, получение символа из потока, изменение даты или времени и многое другое, что вы хотите объявить событием. Состояние – это любая «позиция», на которой останавливается ваш конечный автомат, пока он ожидает другие события. В каждом из состояний вы переопределяете допустимые события (вводы). События временны, тогда как состояния обычно фиксированны; и то и другое – данные, которые можно сохранять. Наконец, вы можете установить код в соответствии с событиями или состояниями и даже решить, должен ли код запускаться при входе в состояние, во время состояния или при выходе из состояния.

Конечный автомат – это просто способ указать код (составить белый список), который будет запускаться в зависимости от различных событий, происходящих в разных точках выполнения. Когда вы получаете непредвиденное событие в конечных автоматах, происходит сбой, так как вам нужно четко указать, какие события (или условия) разрешены при каждом состоянии. Конструкция `if` также обрабатывает возможные варианты, но это черный список возможностей. Если вы забудете добавить ветвь `else`, то все, что не покрывается вашими условиями `if-elif`, просто не будет учтено.

Давайте тогда разберемся в принципах работы конечных автоматов.

1. У вас есть состояния, которые являются хранимым индикатором того, где в данный момент находится конечный автомат. Состояние может быть любым, вроде «начато», «нажата клавиша», «прервано» или

в виде какого-либо другого описания позиции конечного автомата в его возможных точках выполнения. Каждое состояние подразумевает, что он ждет, пока что-то произойдет, прежде чем решить, что делать дальше.

2. У вас есть события, которые могут перемещать автомат из одного состояния в другое. Событием может быть «нажмите клавишу», «ошибка сокета», «файл сохранен». Также оно может представлять какие-либо другие внешние стимулы, которые получает конечный автомат, решая, что делать и в какое состояние входить дальше. Событие может даже возвращаться в то же состояние – так можно построить цикл.

3. Переход конечных автоматов из одного состояния в другое основан на том, какие происходят события, и осуществляется только для точных событий, определенных для этого состояния (хотя одно из событий можно определить как «любое событие»). Состояние не меняется «случайно». Вы можете отслеживать, как конечные автоматы переходили из одного состояния в другое, просматривая полученные события и «посещенные» состояния. Это делает их отладку очень простой.

4. У вас есть код, который может запускаться при каждом событии до, после и во время перехода между состояниями. Это значит, что вы можете запустить код при получении события, затем, находясь в состоянии, решить, что нужно сделать, основываясь на этом событии, а затем снова запустить код до того, как вы покинете это состояние. Такая возможность выполнять код делает конечный автомат очень мощным инструментом.

5. Иногда «ничего» также является событием. Это полезно, потому что таким образом вы можете перевести автомат в новое состояние, даже если ничего не произошло. Однако в практическом плане под «ничего» обычно подразумевается событие «переходи снова» или «просыпайся». В других ситуациях «ничего» означает «пока не уверен, возможно, следующее событие укажет мне состояние».

Мощь конечных автоматов заключается в возможности явно указывать каждое событие, а также в том, что события – это просто получаемые данные. Это делает конечные автоматы невероятно простыми для отладки, тестирования и исправления, поскольку вы точно знаете каждое состояние, и все, что может произойти в каждом состоянии для каждого события. В этом упражнении вы изучите реализацию библиотеки конечного автомата (FSM) и использующий ее конечный автомат, чтобы понять, как они работают.

# Задача упражнения

Я создал модуль конечного автомата, который обрабатывает несколько простых событий для обработки соединений с веб-сервером. Этот воображаемый конечный автомат служит примером для того, как самому быстро написать его на Python. Это только каркас обрабатывающих соединений, которые считывают и записывают из сокета, и ему не хватает нескольких важных вещей. Но это всего лишь небольшой пример, которым вы можете пользоваться.

fsm.py

```
 1    def START():
 2        return LISTENING
 3
 4    def LISTENING(event):
 5        if event == "connect":
 6            return CONNECTED
 7        elif event == "error":
 8            return LISTENING
 9        else:
10            return ERROR
11
12    def CONNECTED(event):
13        if event == "accept":
14            return ACCEPTED
15        elif event == "close":
16            return CLOSED
17        else:
18            return ERROR
19
20    def ACCEPTED(event):
21        if event == "close":
22            return CLOSED
23        elif event == "read":
24            return READING(event)
25        elif event == "write":
26            return WRITING(event)
27        else:
28            return ERROR
29
30    def READING(event):
31        if event == "read":
```

```
32              return READING
33          elif event == "write":
34              return WRITING(event)
35          elif event == "close":
36              return CLOSED
37          else:
38              return ERROR
39
40      def WRITING(event):
41          if event == "read":
42              return READING(event)
43          elif event == "write":
44              return WRITING
45          elif event == "close":
46              return CLOSED
47          else:
48              return ERROR
49
50      def CLOSED(event):
51          return LISTENING(event)
52
53      def ERROR(event):
54          return ERROR
```

Вот также крошечный тест, который показывает вам, как запустить этот модуль:

test_fsm.py

```
1   import fsm
2
3   def test_basic_connection():
4       state = fsm.START()
5       script = ["connect", "accept", "read",
6                 "write", "close", "connect"]
7
8       for event in script:
9           print(event, ">>>", state)
10          state = state(event)
```

Ваша задача в данном упражнении – превратить этот пробный модуль в более надежный и обобщенный класс FSM. Используйте это как набор подсказок

относительно того, как можно обрабатывать входящие события, как состояния могут быть функциями Python и как вы можете осуществлять подразумеваемые переходы. Видите, как иногда я возвращаю функцию для следующего состояния, но в других случаях я возвращаю вызов этой функции состояния? Попытайтесь выяснить, почему я так делаю, поскольку это очень важно в конечных автоматах.

Для завершения этого задания вам нужно изучить модуль Python `inspect` (**docs.python.org/3/library/inspect.html**), чтобы узнать, что вы можете сделать с объектом и классом Python. В `inspect` есть специальные переменные, такие как `__dict__`, а также функции, которые помогают вам заглянуть в класс или объект и найти функцию.

Вы также можете решить, что хотите инвертировать эту конструкцию. События могли бы быть функциями в подклассе, а внутри функций событий вы проверяли бы текущее состояние `self.state`, чтобы определить, что делать дальше. Все зависит от того, что вы обрабатываете, чего у вас больше – событий или состояний, – и что имеет смысл в данный момент.

Наконец, вы могли бы использовать класс `FSMRunner`, который просто знает, как запускать модули, сконструированные подобным образом. У него есть некоторые преимущества перед единственным классом, знающим, как запускать собственные экземпляры, но с ним также могут возникнуть проблемы. Например, как `FSMRunner` отслеживает текущее состояние? Оно помещается в модуль или в экземпляр `FSMRunner`?

# Практические задания

1. Сделайте свой тест более подробным и создайте конечный автомат для применения в совершенно другой, но знакомой вам области.

2. Добавьте возможность включить ведение журнала запускаемых в вашей реализации событий. Одним из самых значительных преимуществ использования конечного автомата для обработки событий является возможность сохранения и регистрации всех событий и состояний, полученных автоматом. Это позволяет вам узнавать, почему он достиг нежелательного состояния.

## Дальнейшее обучение

Вы обязательно должны изучить математический базис конечного автомата. Мой маленький пример здесь представляет собой лишь частично формализованную версию этой концепции, чтобы вы смогли ее понять.

УПРАЖНЕНИЕ 31

# Регулярные выражения

Регулярное выражение – это краткий способ указать последовательность символов для поиска в строке. Регулярные выражения обычно считают «страшными», но, как известно, все, что вызывает страх, обычно просто неправильно преподается. На самом деле регулярные выражения представляют собой набор из примерно восьми символов, которые сообщают компьютеру, как искать текст по шаблону. Они просты в использовании, так что их легко понять. Люди сталкиваются с трудностями, когда пытаются использовать невероятно сложные регулярные выражения там, где был бы уместнее синтаксический анализатор (парсер). Как только вы поймете эти восемь символов и ограничения регулярных выражений, то увидите, что они вовсе не страшны.

Чтобы подготовить ваш мозг к заданию, я заставлю вас запомнить еще кое-что. Для вас важны символы, указанные ниже.

`^`

Якорь для начала строки. Он будет находить соответствие только в том случае, если оно начинается с самого начала.

`$`

Якорь для конца строки. Он будет находить соответствие, только если оно будет в конце строки.

`.`

Любой символ. Соответствие с любым единичным символом входной строки.

`?`

Необязательный предыдущий символ. Предыдущая часть регулярного выражения необязательна, поэтому `A?` означает необязательный символ «A».

`*`

*0 или более* копий предыдущего символа. Берет предыдущую часть регулярного выражения и повторяет или пропускает ее. `A*` будет находить «AAAAAAA» или «BQEFT» (поскольку здесь ноль символов A).

`+`

*1 или более* копии предыдущего символа. То же, что и `*`, но находит соответствие только в том случае, если регулярное выражение содержит

1 или более этих символов. `A+` будет находить «AAAAAAA», но не «BQEFT».

**`[X-Y]`**

Класс (диапазон) символов от X до Y. Находит любой из символов, перечисленных в этом диапазоне. Использование `[A-Z]` найдет все прописные английские буквы. Вместо этого можно использовать сокращения \ для многих распространенных диапазонов символов.

**`()`**

Захватывает эту часть регулярного выражения. Многие библиотеки регулярных выражений используются для замены, извлечения или изменения текста. Захват возьмет часть регулярного выражения внутри `()` и сохранит его для последующего использования. Многие библиотеки позволяют затем ссылаться на эти захваты. Если вы написали `([A-Z]+)`, это захватит 1 или более прописных английских букв.

В библиотеке Python `re` (**docs.python.org/3/library/re.html**) перечислены многие другие символы, но большинство из них являются модификациями этих восьми, либо представляют дополнительные функции, которые редко встречаются в библиотеках регулярных выражений. Вы начнете с создания учебных флеш-карточек для этих восьми, сосредоточив внимание на фразах, выделенных курсивом (якорь для конца, необязательный предыдущий символ), чтобы вы могли быстро вспомнить символы и объяснить, что они делают.

Как только вы запомните эти символы, возьмите следующие регулярные выражения и переведите их на человеческий язык, а также используйте библиотеку Python `re`, чтобы испробовать перечисленные строки или любые другие строки.

```
".*BC?$"     helloBC, helloB, helloA, helloBCX
"[A-Za-z][0-9]+"     A1232344, abc1234, 12345, b493034
"^[0-9]?a*b?.$"     0aaaax, aaab9, 9x, 88aabb, 9zzzz
"*_*" "___***", "_", "****", "__"
"A+B+C+[xyz]*"     AAAABBCCCCCCxyxyz, ABBBBCCCxxxx, ABABABxxxx
```

Переведя их, используйте модуль `re`, чтобы испробовать их в оболочке следующим образом:

Упражнение 31 - сеанс Python

```
>>> import re
>>> m = re.match(r".*BC?$", "helloB").span()
>>> re.match(r".*BC?$", "helloB").span()
(0, 6)
>>> re.match(r"[A-Za-z][0-9]+", "A1232344").span()
(0, 8)
>>> re.match(r"[A-Za-z][0-9]+", "abc1234").span()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
>>> re.match(r"[A-Za-z][0-9]+", "1234").span()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
>>> re.match(r"[A-Za-z][0-9]+", "b493034").span()
(0, 7)
>>>
```

Вы получите ошибку `AttributeError: 'NoneType'` при каждом отсутствии соответствия, потому что функция `re.match` возвращает `None`, когда она не соответствует вашему регулярному выражению.

# Задача упражнения

Задача состоит в том, чтобы попытаться использовать ваш модуль FSM для реализации одного простого регулярного выражения, которое выполняет по меньшей мере три из этих операций. Это сложное задание, но в качестве помощи в планировании и тестировании реализации этого регулярного выражения используйте библиотеку Python `re`. Затем, когда вы поймете, как это сделать, никогда к этому не возвращайтесь. Жизнь слишком коротка, чтобы делать то, что уже хорошо делают компьютеры.

# Практические задания

1. Расширьте набор карточек, чтобы включить все возможные символы из документации библиотеки `re`.

2. Если вы захотите найти символ *, вы можете экранировать его с помощью *. С большинством других символов можно сделать то же самое.

3. Убедитесь, что вы знаете, как использовать `re.ASCII`, так как это включено в требования синтаксического анализа.

## Дальнейшее обучение

Ознакомьтесь с библиотекой `regex` (**pypi.python.org/pypi/regex/**), которая пригодится, если вам нужна поддержка Юникода.

УПРАЖНЕНИЕ 32

# Лексические анализаторы

В упражнении 48 своей первой книги, «Легкий способ выучить *Python 3*», я очень поверхностно охватывал лексические анализаторы. Пришло время разобрать их подробнее. Я объясню концепцию анализа текста, как это относится к регулярным выражениям и как можно создать небольшой лексический анализатор для крошечного фрагмента кода Python.

Давайте используем следующий код Python в качестве примера, чтобы начать упражнение:

```
def hello(x, y):
    print(x + y)

hello(10, 20)
```

Вы уже некоторое время работаете с Python, так что ваш мозг, скорее всего, быстро обработает этот код. Но действительно ли вы понимаете, что здесь происходит? Когда я (или кто-то другой) учил вас Python, вас заставляли запоминать все «символы» (symbols). `def` и `()` являются символами, но у Python должен быть способ обрабатывать их – надежный и последовательный способ. Python также должен иметь возможность считывать `hello` и понимать, что это «имя» для чего-то, а затем понимать разницу между `def hello(x, y)` и `hello(10, 20)`. Как он это делает?

Первый шаг к этому – лексический анализ текста, в процессе которого ищутся «токены». На этом этапе такой язык, как Python, поначалу не заботится о том, что является символом (`def`), а что – именем (`hello`). Он просто попытается преобразовать входной язык в шаблоны текста, называемые «токенами». Он делает это, применяя последовательность регулярных выражений, ищущих «соответствие» с каждым из возможных вводов, понимаемых Python. Из упражнения 31 вы помните, что регулярное выражение – это способ сообщить Python о том, как искать определенный набор символов. Все, что делает интерпретатор Python, это использует много регулярных выражений для установления соответствия с каждым токеном, который он понимает.

Если вы взглянете на код выше, то, возможно, напишете набор регулярных выражений для его обработки. Вам нужно простое регулярное выражение

для `def` – просто def. С символами `()+:,` придется поработать больше. После этого останется решить, как обрабатывать `print`, `hello`, `10` и `20`.

После того как вы определили все символы в примере кода выше, вам нужно их назвать. Вы не можете просто ссылаться на них посредством соответствующих регулярных выражений, поскольку такой поиск неэффективен и запутан. Позже вы узнаете, что предоставление каждому символу собственного имени (или числа) упрощает анализ, но пока давайте разработаем некоторые имена для этих шаблонов регулярных выражений. Можно сказать, что `def` – это просто DEF, тогда символы `()+:,` могут быть LPAREN RPAREN PLUS COLON COMMA (левая скобка, правая скобка, плюс, двоеточие, запятая). После этого я могу назвать регулярное выражение для слов вроде `hello` и `print` просто NAME (имя). Таким образом я придумываю способ преобразовывать поток вводимого кем-то «сырого» текста в поток определенных токенов для использования их на более поздних этапах.

Кроме того, Python каверзен, ведь ему нужно регулярное выражение для пробельных символов, чтобы обрабатывать отступы блоков кода. Пока давайте просто глупо используем `^\s+` и притворимся, что это также предусматривает, сколько пробелов было в начале строки.

В конце концов, у вас будет набор регулярных выражений, которые могут обрабатывать предыдущий код, и он может выглядеть примерно так:

| Регулярное выражение | Токен |
|---|---|
| `def` | DEF |
| `[a-zA-Z_][a-zA-Z0-9_]*` | NAME |
| `[0-9]+` | INTEGER |
| `\(` | LPAREN |
| `\)` | RPAREN |
| `\+` | PLUS |
| `:` | COLON |

| , | COMMA |
|---|---|
| ^\s+ | INDENT |

Задача лексического анализатора состоит в том, чтобы брать эти регулярные выражения и использовать их для разбиения входного текста на поток идентифицированных символов. Если я сделаю это с примером кода, получится следующее:

```
DEF NAME(hello) LPAREN NAME(x) COMMA NAME(y) RPAREN COLON
INDENT(4) NAME(print) LPAREN NAME(x) PLUS NAME(y) RPAREN
NAME(hello) RPAREN INTEGER(10) COMMA INTEGER(20) RPAREN
```

Изучите это преобразование, сопоставляя каждую строку этого вывода лексического анализатора и сравнивая вывод с приведенным выше кодом Python, используя регулярные выражения в таблице. Вы увидите, что все сводится к тому, чтобы взять входной текст (ввод) и поставить в соответствие каждое регулярное выражение и имя токена, после чего осуществить сохранение всей необходимой информации, например `hello` или числа `10`.

# Небольшой лексический анализатор Puny

Я написал крохотный сценарий (скрипт) Puny Python, который демонстрирует сканирование этого маленького языка Python:

ex32.py

```
 1  import re
 2
 3  code = [
 4  "def hello(x, y):",
 5  " print(x + y)",
 6  "hello(10, 20)",
 7  ]
 8
 9  TOKENS = [
10  (re.compile(r"^def"),                    "DEF"),
11  (re.compile(r"^[a-zA-Z_][a-zA-Z0-9_]*"), "NAME"),
12  (re.compile(r"^[0-9]+"),                 "INTEGER"),
13  (re.compile(r"^\("),                     "LPAREN"),
```

```
    (re.compile(r"^\)"),                                    "RPAREN"),
    (re.compile(r"^\+"),                                    "PLUS"),
    (re.compile(r"^:"),                                     "COLON"),
    (re.compile(r"^,"),                                     "COMMA"),
    (re.compile(r"^\s+"),                                   "INDENT"),
    ]

    def match(i, line):
        start = line[i:]
        for regex, token in TOKENS:
            match = regex.match(start)
            if match:
                begin, end = match.span()
                return token, start[:end], end
        return None, start, None

    script = []

    for line in code:
        i = 0
        while i < len(line):
            token, string, end = match(i, line)
            assert token, "Failed to match line %s" % string
            if token:
                i += end
                script.append((token, string, i, end))

    print(script)
```

Когда вы запускаете этот сценарий, вы получаете список кортежей, в которых находятся ТОКЕНЫ, соответствие в строке, начало и конец, следующим образом:

```
[('DEF', 'def', 3, 3), ('INDENT', ' ', 4, 1), ('NAME', 'hello', 9, 5),
('LPAREN', '(', 10, 1), ('NAME', 'x', 11, 1), ('COMMA', ',', 12, 1),
('INDENT', ' ', 13, 1), ('NAME', 'y', 14, 1), ('RPAREN', ')', 15, 1),
('COLON', ':', 16, 1), ('INDENT', ' ', 4, 4), ('NAME', 'print', 9, 5),
('LPAREN', '(', 10, 1), ('NAME', 'x', 11, 1), ('INDENT', ' ', 12, 1),
('PLUS', '+', 13, 1), ('INDENT', ' ', 14, 1), ('NAME', 'y', 15, 1),
('RPAREN', ')', 16, 1), ('NAME', 'hello', 5, 5), ('LPAREN', '(', 6, 1),
```

```
('INTEGER', '10', 8, 2), ('COMMA', ',', 9, 1), ('INDENT', ' ', 10, 1),
('INTEGER', '20', 12, 2), ('RPAREN', ')', 13, 1)]
```

Этот код, безусловно, не будет самым быстрым или самым точным анализатором из тех, что вы можете создать. Это простой сценарий, демонстрирующий основы работы лексического анализатора. Для выполнения реального анализа вы должны использовать инструмент для создания лексического анализатора, что гораздо более эффективно. Я расскажу об этом в разделе «Дальнейшее обучение».

## Задача упражнения

Ваша задача – взять этот образец кода лексического анализатора и превратить его в общий класс `Scanner`. Он будет использоваться позже. Цель этого класса `Scanner` – взять входной файл, просканировать его, поместив токены в этот список, а затем позволить «вытаскивать» токены по порядку. API должен иметь следующие возможности.

**`__init__`**

Принимает похожий список кортежей (без метода `re.compile`) и настраивает анализатор.

**`scan`**

Принимает строку и запускает ее анализ, создавая список токенов на будущее. Вы должны держать эту строку при себе, чтобы позже к ней можно было получить доступ.

**`match`**

Имея список возможных токенов, возвращает первый, соответствующий первому токену в списке, и удаляет его.

**`peek`**

Имея список возможных токенов, возвращает те, что могут соответствовать, но не удаляет их из списка.

**`push`**

Возвращает токен обратно в поток, чтобы позже его вернули `peek` или `match`.

Вы также должны создать общий класс `Token`, который заменяет используемые мной кортежи. Он должен отслеживать найденный токен, соответствие в строке, а также начало и конец области соответствия в исходной строке.

## Практические задания

1. Установите библиотеку `pytest-cov` и используйте ее для измерения покрытия ваших автоматизированных тестов.

2. Используйте результаты `pytest-cov` для улучшения ваших автоматизированных тестов.

## Дальнейшее обучение

Чтобы создать лучший лексический анализатор, полезно знать следующие три факта о регулярных выражениях.

1. Регулярные выражения являются конечными автоматами.

2. Вы можете правильно комбинировать небольшие конечные автоматы с более сложными.

3. Комбинированный конечный автомат, соответствующий многим маленьким регулярным выражениям, будет работать так же, как и каждое регулярное выражение в отдельности, и он будет более эффективным.

Существует множество инструментов, которые используют этот факт, чтобы принять определение анализатора, превратить каждое небольшое регулярное выражение в конечный автомат, а затем объединить их для создания одного быстрого фрагмента кода, который может надежно находить соответствия всем токенам. Преимущество этого заключается в том, что вы можете оперативно передавать этим сгенерированным лексическим анализаторам отдельные символы и быстро идентифицировать

токены. Это предпочтительнее, чем то, что я делаю здесь, разбивая строку и по порядку пробуя последовательность регулярных выражений, пока не найду токен.

Изучите работу генераторов лексических анализаторов и сравните их с тем, что вы написали.

# Синтаксические анализаторы

Представьте, что вам дали огромный список чисел и сказали ввести их в электронную таблицу. Сначала этот огромный список представляет собой всего лишь необработанный поток цифр, разделенных пробелами. Ваш мозг автоматически разбивает поток цифр по пробелам и создает числа, действуя как лексический анализатор. Затем вы берете числа и вводите их в строки и столбцы, имеющие определенное значение. Здесь ваш мозг действует как синтаксический анализатор (парсер), беря плоский поток чисел (токенов) и превращая их в двумерную сетку более осмысленных строк и столбцов. Правила, которым вы следуете, когда определяете, как именно числа попадают в разные строки и столбцы, являются вашей «грамматикой», и работа синтаксического анализатора заключается в обеспечении соблюдения грамматики точно так же, как и в случае с электронной таблицей.

Давайте еще раз рассмотрим пример кода Puny Python из упражнения 32 и обсудим синтаксические анализаторы с трех разных точек зрения:

```
def hello(x, y):
    print(x + y)

hello(10, 20)
```

Что вы видите, когда смотрите на этот код? Я вижу дерево, похожее на ДДП или ТДП, которые мы создавали ранее. А вы видите дерево? Давайте начнем с самого начала этого файла и узнаем, как перейти от символов к дереву.

Вначале, когда мы загружаем файл `.pyy`, это просто поток «символов» – фактически байтов, но Python использует Юникод, так что «символы» подойдут. Эти символы расположены в виде строки и не структурированы. Задача нашего лексического анализатора заключается в том, чтобы добавить начальный уровень осмысленности. Лексический анализатор делает это, используя регулярные выражения, чтобы извлечь этот смысл из потока символов, создав список токенов. Мы перешли от списка символов к списку токенов, но давайте взглянем на `def hello(x, y):`. Это функция с блоком кода внутри. Это означает определенную форму структурирования по типу «содержания» или «вещи внутри вещи».

Очень простой способ представления содержания – это дерево. Мы могли бы использовать таблицу, подобную вашей электронной таблице, но работать с ней не так просто, как с деревом. Далее посмотрите на часть `hello(x, y)`. У нас есть токен `NAME (hello)`, но нам нужно взять содержимое части `(...)` и знать, что оно находится внутри круглых скобок. Опять же, мы можем использовать дерево, и мы вкладываем часть `x, y` внутрь части `(...)` как дочерний элемент или ветвь дерева. В конце концов, у нас будет дерево, начинающееся с корня этого кода Python. Каждый блок, вывод, определение функции и вызов функции будут представлять собой ветви, отходящие от корня, у которых будут сыновья и т. д.

Зачем нам это делать? Нам нужно знать структуру кода Python, основанную на ее грамматике, чтобы позже мы могли ее проанализировать. Если мы не преобразуем линейный список токенов в древовидную структуру, тогда мы не будем иметь понятия, где находятся границы функций, блоков, ветвей или выражений. Нам придется определять границы на лету по «прямолинейному» принципу, что не так просто сделать, сохраняя качество. Множество ранних ужасных языков программирования – прямолинейные, и теперь мы знаем, что такими они быть не должны. Мы можем использовать синтаксический анализатор, чтобы построить древовидную структуру.

Задача парсера – принимать список токенов от лексического анализатора и переводить их в более осмысленное дерево грамматики. Вы можете рассматривать синтаксический анализатор как применение к потоку токенов еще одного регулярного выражения. Регулярное выражение лексического анализатора запаковывает куски символов в токены. Регулярное выражение синтаксического анализатора помещает эти токены в ящики, внутри которых есть другие ящики, и так далее, пока токены не перестанут быть линейными.

Синтаксический анализатор также добавляет смысл этим ящикам. Он просто отбрасывает токены скобок и создает специальный список параметров `parameters` для возможного класса `Function` (функции). Это отбрасывает двоеточия, бесполезные пробелы, запятые и любые токены, фактически не добавляющие смысла, и переведет их во вложенные структуры, которые легче обрабатывать. Конечный результат может выглядеть как это ненастоящее дерево для вышеуказанного образца кода:

```
* root
  * Function
    - name = hello
    - parameters = x, y
    - code:
      * Call
```

```
      - name = print
      - parameters =
        * Expression
          - Add
            - a = x
            - b = y
* Call
  - name = hello
  - parameters = 10, 20
```

Прежде чем двигаться дальше, разберитесь, как я перешел от кода Python к этому выдуманному дереву, которое представляет его. Это несложно понять, но крайне важно, чтобы вы могли взглянуть на код Python и создать соответствующую древовидную структуру.

## Синтаксический анализ методом рекурсивного спуска

Существует несколько проверенных способов создания синтаксического анализатора для такого вида грамматики. Наиболее простым из них является метод «рекурсивного спуска» (MPC). На самом деле я охватывал эту тему в упражнении 49 книги «Легкий способ выучить *Python 3*». Для обработки языка своей небольшой игры вы создали простой синтаксический анализатор MPC, даже не подозревая об этом. В этом упражнении я предоставлю более формальное описание того, как писать синтаксический анализатор MPC, а затем вы попробуете применить его к нашему фрагменту кода Python выше.

Метод рекурсивного спуска использует взаимно рекурсивные вызовы функций, которые реализуют древовидную структуру данной грамматики. Код для синтаксического анализатора MPC выглядит как фактическая грамматика, обрабатываемая вами, и его легко написать, если следовать определенным правилам. Синтаксические анализаторы MPC имеют два недостатка: они могут быть недостаточно эффективными, а также их обычно пишут вручную. Поэтому они содержат больше ошибок, чем сгенерированные парсеры. Существуют также некоторые теоретические ограничения в отношении того, что синтаксический анализатор MPC может анализировать, но поскольку вы будете писать его вручную, то сможете обойти многие из этих ограничений.

Чтобы написать парсер MPC, вам необходимо обработать ваши токены лексического анализатора, используя три основные операции.

**peek**

Вернуть, если следующий токен может соответствовать, но не убирать его из потока.

**match**

Найти соответствие следующему токену, убрав его из потока.

**skip**

Пропустить следующий токен, так как он не нужен, и убрать его из потока.

Как вы заметили, это те же самые три операции, что вы создавали для класса `Scanner` в упражнении 33, и вот почему. Они нужны вам для создания синтаксического анализатора MPC.

Вы используете эти три функции для написания функций разбора грамматики, которые принимают токены от лексического анализатора. Коротким примером может служить синтаксический анализ следующей простой функции:

```
def function_definition(tokens):
    skip(tokens) # отбрасываем def
    name = match(tokens, 'NAME')
    match(tokens, 'LPAREN')
    params = parameters(tokens)
    match(tokens, 'RPAREN')
    match(tokens, 'COLON')
    return {'type': 'FUNCDEF', 'name': name, 'params': params}
```

Как видите, я просто беру токены и использую для их обработки операции `match` и `skip`. Вы также заметили, что у меня есть функция `parameters`, представляющая «рекурсивный» аспект метода рекурсивного спуска. `function_definition` вызывает `parameters`, когда необходимо проанализировать параметры для функции.

# Грамматика формы Бэкуса – Наура

Трудновато пытаться написать парсер MPC с нуля без какой-либо спецификации грамматики. Помните, как я попросил вас преобразовать одно регулярное выражение в конечный автомат? Это было трудно, правда? Для этого требовалось гораздо больше кода, чем несколько символов регулярного выражения. Когда в этом упражнении вы станете писать синтаксический анализатор MPC, вы сделаете аналогичную вещь, так что будет удобно располагать языком, являющимся «регулярным выражением для грамматик».

Наиболее распространенным «регулярным выражением для грамматик» является форма Бэкуса – Наура (БНФ), названная в честь создателей — ученых Джона Бэкуса и Петера Наура. БНФ описывает требуемые токены и то, как эти токены повторяются при формировании грамматики языка. БНФ использует те же символы, что и регулярные выражения, поэтому *, + и ? имеют сходные значения.

Для этого упражнения я собираюсь использовать синтаксис расширенной БНФ (**tools.ietf.org/html/rfc5234**), чтобы определить грамматику для фрагмента Puny Python. Операторы РБНФ в основном такие же, как у регулярных выражений, за тем исключением, что по какой-то странной причине они указывают повторение перед повторяющимся элементом, а не после. В остальном прочтите спецификацию и попытайтесь выяснить, что означает следующее:

```
root = *(funccal / funcdef)
funcdef = DEF name LPAREN params RPAREN COLON body
funccall = name LPAREN params RPAREN
params = expression *(COMMA expression)
expression = name / plus / integer
plus = expression PLUS expression
PLUS = "+"
LPAREN = "("
RPAREN = ")"
COLON = ":"
COMMA = ","
DEF = "def"
```

Давайте взглянем только на строку `funcdef` и по соответствующим частям сравним ее с `function_definition` из указанного выше кода Python.

**funcdef =**

Это мы воссоздаем с помощью `def function_definition(tokens);` этим начинается данная часть нашей грамматики.

**DEF**

В грамматике это указано как DEF = "`def`", а в коде Python я это просто пропускаю с помощью `skip(tokens)`.

**name**

Мне это нужно, поэтому я устанавливаю соответствие при помощи `name = match(tokens, 'NAME')`. Я использую прописные буквы, чтобы указать то, что в БНФ я, скорее всего, пропущу.

**LPAREN**

Я предположил, что получил `def`, но теперь хочу применить `(`, так что устанавливаю соответствие, но игнорирую результат, используя `match(tokens, 'LPAREN')`. Это вроде как «обязательно, но пропущено».

**params**

В БНФ я определяю `params` как новую «продукцию грамматики» или новое «правило грамматики». Это означает, что в коде Python мне нужна новая функция. В этой функции я могу вызвать другую функцию с помощью `params = parameters(tokens)`. Позже я определяю функцию `parameters` для обработки параметров, разделенных запятыми.

**RPAREN**

Я отбрасываю это, но снова требую при помощи `match(tokens, 'RPAREN')`.

**COLON**

Опять-таки, отбрасываю соответствие `match(tokens, 'COLON')`.

**body**

На самом деле я здесь пропускаю тело, потому что синтаксис отступа в Python слишком сложен для этого простого примера. Вам нужно будет справиться с этой проблемой в упражнении, если не хотите сделать это прямо сейчас.

Вот основа того, как нужно читать спецификацию РБНФ и систематически переводить ее в код. Вы начинаете с корня, реализуете каждую продукцию грамматики как функцию и позволяете лексическому анализатору обрабатывать простые токены (которые я обозначаю ПРОПИСНЫМИ буквами).

## Быстрый демонстрационный синтаксический анализатор

Вот мой быстрый синтаксический анализатор MPC, который вы можете использовать в качестве основы для создания более формального и изящного парсера:

ex33.py

```
from scanner import *
from pprint import pprint

def root(tokens):
    """root = *(funccal / funcdef)"""
    first = peek(tokens)

    if first == 'DEF':
        return function_definition(tokens)
    elif first == 'NAME':
        name = match(tokens, 'NAME')
        second = peek(tokens)

        if second == 'LPAREN':
            return function_call(tokens, name)
        else:
            assert False, "Not a FUNCDEF or FUNCCALL"

def function_definition(tokens):
```

```
20          """
21          funcdef = DEF name LPAREN params RPAREN COLON body
22          Я игнорирую тело, поскольку это слишком сложно.
23          В смысле, можете сами в этом разобраться.
24          """
25          skip(tokens) # отбрасываем def
26          name = match(tokens, 'NAME')
27          match(tokens, 'LPAREN')
28          params = parameters(tokens)
29          match(tokens, 'RPAREN')
30          match(tokens, 'COLON')
31          return {'type': 'FUNCDEF', 'name': name, 'params': params}
32
33      def parameters(tokens):
34          """params = expression *(COMMA expression)"""
35          params = []
36          start = peek(tokens)
37          while start != 'RPAREN':
38              params.append(expression(tokens))
39              start = peek(tokens)
40              if start != 'RPAREN':
41                  assert match(tokens, 'COMMA')
42          return params
43
44      def function_call(tokens, name):
45          """funccall = name LPAREN params RPAREN"""
46          match(tokens, 'LPAREN')
47          params = parameters(tokens)
48          match(tokens, 'RPAREN')
49          return {'type': 'FUNCCALL', 'name': name, 'params': params}
50
51      def expression(tokens):
52          """expression = name / plus / integer"""
53          start = peek(tokens)
54
55          if start == 'NAME':
56              name = match(tokens, 'NAME')
57              if peek(tokens) == 'PLUS':
58                  return plus(tokens, name)
59              else:
60                  return name
61          elif start == 'INTEGER':
62              number = match(tokens, 'INTEGER')
```

```
63              if peek(tokens) == 'PLUS':
64                  return plus(tokens, number)
65              else:
66                  return number
67          else:
68              assert False, "Syntax error %r" % start
69
70      def plus(tokens, left):
71          """plus = expression PLUS expression"""
72          match(tokens, 'PLUS')
73          right = expression(tokens)
74          return {'type': 'PLUS', 'left': left, 'right': right}
75
76
77      def main(tokens):
78          results = []
79          while tokens:
80              results.append(root(tokens))
81          return results
82
83      parsed = main(scan(code))
84      pprint(parsed)
```

Обратите внимание, что я использую написанный мной модуль `scanner`, который содержит функции `match`, `peek`, `skip` и `scan`. Я использую `from scanner import *` только для того, чтобы сделать этот пример более понятным. Вы должны использовать свой класс `Scanner`.

Также заметьте, что внутри этого маленького синтаксического анализатора я поместил РБНФ в комментарии внутри каждой функции. Это помогло мне писать код, и позже я смогу использовать это для отчетов об ошибках. Прежде чем переходить к разделу «Задача упражнения», вы должны изучить этот анализатор, возможно даже с использованием метода мастер-копии.

## Задача упражнения

Ваша следующая задача состоит в том, чтобы объединить свой класс `Scanner` с недавно написанным классом `Parser`, в котором вы можете выделить подкласс. Также его можно повторно реализовать с помощью моего простого

синтаксического анализатора. Ваш базовый класс `Parser` должен иметь следующие возможности.

1. Брать объект класса `Scanner` и обрабатывать его. Вы можете предположить, что любая функция по умолчанию представляет начало грамматики.

2. Предполагать код обработки ошибок, более совершенный, чем простое использование `assert`.

Затем вы должны реализовать `PunyPythonPython`, который может анализировать только этот крошечный язык Python, и сделать следующее.

1. Вместо того чтобы просто создавать список словарей, следует создать классы для каждого результата продукций грамматики. Затем эти классы становятся объектами внутри списка.

2. Классам нужно хранить только те токены, которые анализируются; но позже их возможности будут расширены.

3. Вам нужно проанализировать только этот маленький язык, но попытайтесь решить проблему с отступами в Python. Возможно, вам придется изменить `Scanner` так, чтобы он разбирался только с соответствием токена отступа в начале строки и игнорировал отступы в других местах. При этом вам нужно будет отслеживать, сколько вы отступили, а также отметить нулевой отступ, чтобы можно было убирать отступы между блоками.

Обширный набор тестов включал бы передачу парсеру большего количества образцов этого небольшого кода, но пока просто проведите синтаксический анализ одного этого маленького файла. Постарайтесь сделать так, чтобы ваш тест имел хорошее покрытие и обнаруживал как можно больше ошибок.

# Практическое задание

Это упражнение огромно, поэтому просто выполните его. Не торопитесь и выполняйте маленькими порциями. Я крайне рекомендую изучать мой маленький образец, пока полностью не разберетесь, а также выводить обрабатываемые токены в ключевых точках.

## Дальнейшее обучение

Ознакомьтесь с генератором парсеров SLY Дэвида Бизли (**sly.readthedocs.io/en/latest/**), который может создать синтаксический и лексический анализатор (или лексер) за вас. Не стесняйтесь пытаться воссоздать это упражнение при помощи `SLY` для сравнения.

УПРАЖНЕНИЕ 34

# Семантические анализаторы

Теперь у вас есть синтаксический анализатор, который должен создавать дерево объектов продукций грамматики. Я назову его вашим «деревом разбора грамматики» (синтаксическим деревом). Это означает, что вы можете проанализировать всю программу, начиная с вершины дерева разбора, проходя по нему, пока не обойдете каждый узел. Вы делали что-то подобное, когда изучали структуры данных ДДП и ТДП. Вы начинали сверху и посещали каждый узел. При этом порядок, в котором вы их посещали (методы «в глубину», «в ширину», по порядку и т. д.), определял, как обрабатывались узлы. Ваше дерево разбора имеет такую же возможность. Следующим шагом в написании маленького интерпретатора Python будет обход дерева и его анализ.

Задача семантического анализатора – найти семантические ошибки и исправить или добавить информацию, необходимую для следующего этапа. Семантические ошибки – это ошибки в программе Python, которые при правильной грамматике не имеют смысла. Это может быть что угодно, начиная с переменной, которая еще не определена, и заканчивая нелогичным кодом, не имеющим никакого смысла. Грамматика некоторых языков крайне несовершенна, поэтому семантическому анализатору нужно проделать большую работу, чтобы исправить дерево разбора. Другие языки настолько легко анализируются и обрабатываются, что им даже не нужен шаг с семантическим анализатором.

Чтобы написать семантический анализатор, вам нужно найти способ посетить каждый узел в дереве разбора, проанализировать его на наличие ошибок и добавить любую недостающую информацию. Есть три общих способа это сделать.

1. Вы создаете анализатор, который знает, как обновлять каждую продукцию грамматики. Он будет обходить дерево разбора аналогично тому, как это делал ваш класс `Parser`, с функцией для каждого типа продукции. Но задача семантического анализатора – изменять, обновлять и проверять продукции грамматики.

2. Вы изменяете свои продукции грамматики так, чтобы **они сами** знали, как анализировать собственное состояние. Тогда ваш семантический анализатор – это просто механизм, который обходит дерево разбора грамматики, вызывая метод `analyze()` каждой продукции. Для

этого способа вам понадобится определенное состояние, которое передается каждому классу продукции, и это должен быть третий класс.

3. Вы создаете отдельный набор классов, реализующих окончательное анализируемое дерево, которое вы можете передать интерпретатору. Во многом вы отразите продукции грамматики синтаксического анализатора набором новых классов, которые принимают глобальное состояние и продукцию грамматики и настраивают их `__init__` так, чтобы они были результатом анализа.

В «Задаче упражнения» я рекомендую использовать либо второй, либо третий способ.

## Шаблон Посетитель

Шаблон Посетитель – очень распространенная методика в объектно-ориентированных языках программирования. Она предполагает создание классов, которые знают, что они должны делать, когда их «посетят». Это позволяет свести код для обработки класса к этому классу. Преимущество этой методики в том, что вам не нужны огромные конструкции `if`, которые проверяют типы в классах, чтобы узнать, что делать. Вместо этого вы просто создаете класс, подобный этому:

```
class Foo(object):
    def visit(self, data):
        # сделать что-то с self для foo
```

Как только у вас есть этот класс (`visit` можно назвать как угодно), вы просто проходите по списку и вызываете его:

```
for action in list_of_actions:
    action.visit(data)
```

Этот шаблон будет использоваться для семантических анализаторов из способов 2 и 3; единственное различие заключается в следующем.

1. Если вы решите, что ваши продукции грамматики также будут результатами анализа, тогда ваша функция `analyze()` (это наш метод

`visit()`) просто сохраняет эти данные в классе продукции или в состоянии, которое ей дано.

2. Если вы решите, что ваши продукции грамматики будут производить другой набор классов для интерпретатора (см. упражнение 35), тогда каждый вызов `analyze` будет возвращать новый объект, который вы поместите в список «на потом» или присоедините, как дочерний объект, к текущему объекту.

Я расскажу о первой ситуации, когда ваши продукции грамматики также являются результатами работы вашего семантического анализатора. Это работает для нашего простого маленького сценария Puny Python, и сейчас вы должны следовать этому стилю. Если захотите, можете позже попробовать применить другое проектирование.

# Короткий семантический анализатор Puny Python

---

**Внимание!** Остановитесь здесь, если хотите самостоятельно попытаться реализовать шаблон Посетитель для продукций грамматики. Я предоставляю полноценный, но простой пример, который полон «спойлеров».

---

Концепция шаблона Посетитель кажется странной, тем не менее она очень логична. Каждая продукция грамматики знает, что она должна делать на различных этапах, поэтому вы можете также сохранить код для этого этапа рядом с данными, которые ему нужны. Чтобы продемонстрировать это, я написал небольшую модель `PunyPyAnalyzer`, которая просто выводит синтаксический анализ, используя шаблон Посетитель. Я выполняю только одну пробную продукцию грамматики, чтобы вы могли понять, как это делается. Не хочу давать вам слишком много подсказок.

Первое, что я делаю, это определяю класс `Production`, от которого будут наследовать все мои продукции грамматики.

ex34a.py

```
1    class Production(object):
```

```
2         def analyze(self, world):
3             """Реализуйте свой анализатор здесь."""
```

Здесь мой первоначальный метод `analyze()`, и мы берем `PunyPyWorld`, который будем использовать позже. Первая продукция грамматики – продукция `FuncCall`:

ex34a.py

```
1     class FuncCall(Production):
2
3         def __init__(self, name, params):
4             self.name = name
5             self.params = params
6
7         def analyze(self, world):
8             print("> FuncCall: ", self.name)
9             self.params.analyze(world)
```

Вызовы функций содержат имя (name) и `params` – класс продукции `Parameters` для параметров вызова функции. Взгляните на метод `analyze()`, и вы увидите первую функцию Посетителя. Когда вы дойдете до `PunyPyAnalyzer`, вы увидите, как она выполняется, но обратите внимание, что эта функция затем вызывает `param.analyze(world)` для каждого из параметров этой функции:

ex34a.py

```
1     class Parameters(Production):
2
3         def __init__(self, expressions):
4             self.expressions = expressions
5
6         def analyze(self, world):
7             print(">> Parameters: ")
8             for expr in self.expressions:
9                 expr.analyze(world)
```

Это приводит к классу `Parameters`, который содержит каждое из выражений, составляющих параметры для функции. `Parameters.analyze` просто проходит через свой список выражений, которых у нас два:

ex34a.py

```
class Expr(Production): pass

class IntExpr(Expr):
    def __init__(self, integer):
        self.integer = integer

    def analyze(self, world):
        print(">>>> IntExpr: ", self.integer)

class AddExpr(Expr):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def analyze(self, world):
        print(">>> AddExpr: ")
        self.left.analyze(world)
        self.right.analyze(world)
```

В этом примере я прибавляю только два числа. Для этого я создаю базовый класс `Expr`, а затем классы `IntExpr` и `AddExpr`. У каждого из них просто есть методы `analyze()`, которые выводят их содержимое.

Таким образом, у нас есть классы для наших синтаксических деревьев, и мы можем провести определенный анализ. Первое, что нам нужно, – это мир (world), который может отслеживать определения переменных, функции и другие данные, которые нужны нашим методам `Production.analyze()`.

ex34a.py

```
class PunyPyWorld(object):

    def __init__(self, variables):
        self.variables = variables
        self.functions = {}
```

Когда вызывается какой-либо метод `Production.analyze()`, ему передается объект `PunyPyWorld`, поэтому метод `analysis()` знает состояние world. Он может обновлять переменные, искать функции и проводить анализ везде в world.

Затем нам нужен `PunyPyAnalyzer`, который может принять дерево разбора и world и выполнить все грамматические продукции:

ex34a.py

```
1    class PunyPyAnalyzer(object):
2        def __init__(self, parse_tree, world):
3            self.parse_tree = parse_tree
4            self.world = world
5
6        def analyze(self):
7            for node in self.parse_tree:
8                node.analyze(self.world)
```

Это достаточно просто, чтобы настроить простой вызов функции `hello (10 + 20)`:

ex34a.py

```
1    variables = {}
2    world = PunyPyWorld(variables)
3    # смоделировать hello(10 + 20)
4    script = [FuncCall("hello",
5                Parameters(
6                    [AddExpr(IntExpr(10), IntExpr(20))])
7                )]
8    analyzer = PunyPyAnalyzer(script, world)
9    analyzer.analyze()
```

Убедитесь, что вы понимаете, как я структурировал эту переменную `script`. Обратите внимание, что в первую очередь там используется список.

## Синтаксический анализатор против семантического анализатора

В этом примере я предполагаю, что `PunyPyParser` преобразовал токены `NUMBER` в целые числа. В других языках вы можете оставить только токены и заставить `PunyPyAnalyzer` выполнить преобразование. Все зависит от того, где вы хотите получить ошибки, и где вы можете провести наиболее полезный анализ. Если вы возложите работу на синтаксический анализатор, то можете сразу же получить ошибки при форматировании. Если воспользуетесь семантическим анализатором, можете получить ошибки, которые будут затрагивать весь синтаксически проанализированный файл.

## Задача упражнения

Предназначение всех этих методов `analyze()` заключается не в том, чтобы просто осуществлять вывод, но изменить внутреннее состояние каждого подкласса `Production`, чтобы интерпретатор мог запустить его как сценарий. Ваша задача в этом упражнении – взять ваши классы продукций грамматики (которые могут отличаться от моих) и заставить их провести анализ.

Не стесняйтесь использовать мою отправную точку. Если нужно, можете взять мой анализатор и мой world, но сначала вы должны попытаться написать собственные. Вы также должны сравнить свои классы продукций из упражнения 33 с моими. Ваши лучше? Поддерживают ли они это проектирование? Если нет, измените их.

Ваш семантический анализатор должен будет делать несколько вещей, чтобы интерпретатор работал правильно.

1. Отслеживать определения переменных. В настоящем языке это потребовало бы очень сложных вложенных таблиц, но для Puny Python просто предположим, что есть одна гигантская таблица (ТДП или словарь), в которой находятся все переменные. Это означает, что параметры `x` и `y` функции `hello(x, y)` на самом деле глобальные переменные.

2. Отслеживать местоположение функций, чтобы их можно было запустить позже. В нашем Puny Python будут простые функции, которые можно запустить, но когда работает интерпретатор, ему нужно

«перепрыгивать» к ним и запускать их. Лучший способ обеспечить такую возможность – хранить функции в себе.

3. Проверьте наличие любых приходящих в голову ошибок, например отсутствующих переменных. Это сложно, потому что языки, подобные Python, выполняют большую часть проверки ошибок на этапе интерпретации. Вы должны решить, какие ошибки возможны во время анализа, и реализовать их проверку. Например, что произойдет, если я попытаюсь использовать переменную, которая не была определена?

4. Если вы правильно использовали синтаксис Python `INDENT`, тогда у ваших продукций `FuncCall` должен быть соответствующий прикрепленный код. Он потребуется интерпретатору для запуска, поэтому убедитесь в его наличии.

# Практические задания

1. Это упражнение и так довольно сложное, но как вы создали бы лучший способ хранения переменных, реализующий хотя бы еще один уровень области видимости? Помните, что идея «область видимости» заключается в том, что параметры `x`, `y` в `hello(x, y)` не влияют на переменные `x` или `y`, определенные за пределами функции `hello`.

2. Реализуйте присваивание в своем лексическом, синтаксическом и семантическом анализаторе. Это значит, у меня должна быть возможность написать `x = 10 + 14`, а вы должны суметь обработать это.

# Дальнейшее обучение

Изучите разницу между языками программирования «основанными на выражениях» (expression-based) и «основанными на инструкциях» (statement-based). Говоря коротко, в некоторых языках есть только выражения, так что у всего есть какое-то подобие связанного возвращаемого значения. В других языках есть выражения, имеющие значение, и инструкции, которые не имеют значения, поэтому присвоение им переменных должно завершиться неудачей. К какому типу языков относится Python?

УПРАЖНЕНИЕ 35

# Интерпретаторы

Это заключительное упражнение на тему анализа текста должно быть одновременно сложным и веселым. Вы наконец-то увидите, как ваш сценарий Puny Python выполняется и что-то делает. Нет ничего страшного, если вы испытываете трудности с этим разделом и концепцией синтаксического анализа. Если, достигнув этого момента, вы не совсем понимаете, что происходит, сделайте шаг назад и подумайте о том, чтобы заново выполнить упражнения из данной части. Прежде чем продолжать, повторите этот раздел несколько раз – это поможет вам справиться с последними двумя упражнениях части V, в которых вы создадите собственные небольшие языки.

Я специально не включил в это упражнение код, чтобы вы попытались сами его написать, основываясь на описании того, как работает интерпретатор. У вас уже есть Python в качестве справки о том, как должны работать эти маленькие инструкции в нашем примере Puny Python. Вы знаете, как обходить дерево разбора с помощью шаблона Посетителя. Все, что вам осталось, – это написать интерпретатор, который сможет склеить все это вместе и запустить ваш маленький сценарий.

## Интерпретаторы против компиляторов

Среди языков программирования различают языки для компиляции и для интерпретации. Компилируемый язык принимает ваши входные данные и поэтапно проводит лексический, синтаксический и семантический анализ, как это делали вы. Затем компилятор «выпускает» машинный код на основе этого анализа, обходя его и записывая байты, которые требуются реальному (или фиктивному) компьютеру для запуска процессора. В некоторых компиляторах добавлен дополнительный шаг перевода исходного кода на «промежуточный язык», который является достаточно обобщенным, чтобы его потом можно было превратить в байты для распознавания машиной. Компиляторы легко узнать, потому что вы не можете просто запустить их, сначала вам нужно пропустить исходный код через компилятор, а затем выполнить результат. C – классический компилятор. Программа на C запускается следующим образом:

```
$ cc ex1.c -o ex1
$ ./ex1
```

Команда `cc` является «компилятором C». Мы указываем, что нужно взять файл `ex1.c`, осуществить его лексический, синтаксический и семантический анализ, а затем вывести выполняемые байты в файл `ex1`. После этого его можно будет запустить, как любую другую программу.

Интерпретатор не создает скомпилированный байт-код, который можно запускать, но вместо этого он просто запускает результат анализа напрямую. Он «интерпретирует» язык ввода так же, как человек, владеющий двумя языками, может перевести с английского на русский. Он загружает исходный файл, затем проводит его лексический, синтаксический и семантический анализ так же, как компилятор. Затем он просто использует собственный язык интерпретатора (в данном случае Python) для запуска файла на основе анализа.

Если бы нам нужно было реализовать интерпретатор JavaScript в Python, мы бы «интерпретировали JavaScript с помощью Python». JavaScript – мой «английский», а интерпретатор переводит его на Python («русский») на лету. Если бы я хотел интерпретировать `1 + 2` из JavaScript с помощью Python, я мог бы сделать это следующим образом.

1. Провести лексический анализ `1 + 2` и создать токены `INT(1) PLUS INT(2)`.

2. Провести синтаксический анализ и создать выражение `AddExpr(IntExpr (1), IntExpr(2))`.

3. Провести семантический анализ, чтобы преобразовать текст `1` и `2` в настоящие целые числа Python.

4. Интерпретировать это, получив код Python `result = 1 + 2`, который я могу передать остальному дереву разбора грамматики.

Для сравнения, компилятор сделал бы все то же, что сделал я в пунктах 1–3, но затем в четвертом пункте он бы записал байт-код (машинный код) в другой файл, который я затем запустил бы на ЦП.

## Python – это и то и другое

Python немного современнее и использует преимущество более быстрых компьютеров, фактически осуществляя как компиляцию, так и интерпретацию. Он работает как интерпретатор, поэтому вам не нужно проходить этап компиляции. Но интерпретаторы медленны, поэтому у Python есть своего

рода встроенная виртуальная машина. При запуске сценария, вроде `python ex1.py`, Python действительно запускает его и компилирует в файл `ex1.cpython-36.pyc` в каталоге `__pycache__`. Этот файл является байт-кодом, программа `python` знает, как его загружать и запускать, и он работает как фиктивный машинный код.

Ваш интерпретатор никогда не будет и никогда не должен быть настолько элегантным. Он просто должен проводить лексический, синтаксический, семантический анализ и интерпретировать сценарий Puny Python.

## Как написать интерпретатор

Когда вы пишете интерпретатор, вам нужно будет переключаться между всеми тремя этапами, чтобы исправлять ошибки или восполнять пробелы. Предлагаю вам сначала добавить числа, а затем работать над более сложными выражениями, пока ваш сценарий не запустится. Я использовал бы такой подход.

1. Добавьте свой первый метод `interpret` в класс `AddExpr`, и пусть он просто выводит сообщение.

2. Убедитесь, что интерпретатор как следует «посетил» этот класс, передав нужный ему `PunyPyWorld`.

3. После этого вы можете поручить `AddExpr.interpret` задачу вычислить сумму его двух выражений и вернуть результат.

4. Затем вам нужно выяснить, куда должны идти результаты этого метода `interpret`. Чтобы не усложнять, давайте предположим, что Puny Python – это язык, основанный на выражениях, поэтому все возвращает значение. В этом случае вызовы одного интерпретатора всегда возвращают что-то, что могут использовать родительские вызовы.

5. Наконец, поскольку Puny Python основан на выражениях, ваш интерпретатор может вывести окончательный результат вызова своего `interpret`.

Если вы с этим справитесь, то получите основу своего интерпретатора и сможете приступить к реализации всех остальных необходимых для работы методов `interpret`.

## Задача упражнения

Написание интерпретатора для Puny Python не должно включать в себя ничего, кроме написания еще одного шаблона Посетителя, который проходит через анализируемое дерево разбора, выполняя то, что говорит это дерево. Ваша единственная цель – запустить этот один крошечный сценарий. Она кажется простой, ведь это всего лишь три строки кода, но эти три строки охватывают широкий спектр тем в программировании: переменные, прибавление, выражение, определения функций и вызовы функций. Если бы вы реализовали конструкции `if`, у вас был бы практически рабочий язык программирования.

Вашей задачей является написать класс `PunyPyInterpreter`, который принимает `PunyPyWorld` и результаты запуска `PunyPyAnalyzer` для выполнения сценария. Вам придется реализовать команду `print` как нечто, просто выводящее свои переменные, но остальная часть кода должна запускаться вами при прохождении каждого класса продукции.

## Практические задания

1. Как только у вас будет `PunyPyInterpreter`, вы должны реализовать инструкции `if` и логические выражения, а затем расширить свой набор тестов, чтобы убедиться, что они работают. Постарайтесь усовершенствовать этот маленький интерпретатор Python, насколько сможете.

2. Что нужно сделать, чтобы у Puny Python тоже были инструкции?

## Дальнейшее обучение

Теперь вы должны уметь изучать грамматики и спецификации для каких угодно языков. Давайте, выбирайте языки и учите их, и делайте это, используя исходный код. Вы также должны более подробно изучить спецификации РФБН на странице **tools.ietf.org/html/rfc5234**, чтобы подготовиться к следующим двум упражнениям.

УПРАЖНЕНИЕ 36

# Простой калькулятор

Задание данного упражнения состоит в том, чтобы создать простой алгебраический калькулятор, используя все, что вы узнали об анализе. Вам нужно будет разработать язык для выполнения базовых математических операций с переменными, создать РФБН для языка и написать для него лексический, синтаксический и семантический анализаторы, а также интерпретатор. На самом деле это может быть излишним для простого языка калькулятора, поскольку в нем не будет никаких вложенных структур, таких как функции, но все равно выполните задание, чтобы понять весь процесс.

## Задача упражнения

Простой алгебраический язык для разных людей может означать множество вещей, поэтому я хочу, чтобы вы поиграли с командой Unix под названием `bc`. Вот пример моего запуска команды `bc`:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
x = 10
y = 11
j = x * y
j
110
```

Вам нужна возможность создавать переменные, вводить числа (как целые, так и числа с плавающей точкой) и использовать столько операторов, сколько вы сможете внедрить. Можете играть с `bc`, или даже с оболочкой Python, и разрабатывать РФБН в процессе. Помните, что ваша РФБН – практически псевдокод и не должна быть идеально грамотной, но просто достаточно правильной для того, чтобы вы могли создать свои лексический и синтаксический анализаторы.

Как только у вас будет «эскиз» грамматики в форме РФБН, можете приступать к созданию лексического и синтаксического анализаторов. Я написал бы набор простых сценариев, которые будут осуществлять то, что, по вашему мнению, должен делать язык, а затем при помощи тестового набора на каждом этапе прогонял бы их через калькулятор. Это упрощает его проверку.

После того как вы создали парсер, нужно написать семантический анализатор, чтобы добавить основательности, и проверить семантическое значение ввода. Для простого языка, вроде нашего, это больше, чем нужно, но на небольшом игрушечном языке вы упражняетесь в выполнении всего процесса. Помните, что важная часть работы семантического анализатора заключается в отслеживании определений переменных в разных местах сценария, чтобы интерпретатор мог получить к ним доступ во время выполнения.

Если ваш семантический анализатор способен создавать выполняемое дерево разбора, можете написать интерпретатор, который выполняет его. Как упоминалось в упражнении 35, есть два способа написать интерпретатор. Первый – вы можете создать «машину», которая умеет выполнять продукции грамматики как последовательности входов. Тогда ваши классы продукции грамматики (`Expression`, `Assignment` и т. д.) как будто представляют собой машинный код и просто выполняют то, что сами содержат. Другой способ, применимый к языку ООП, коим является Python, заключается в том, чтобы каждый класс продукции сам знал, как выполнять себя. При использовании такого стиля классы «умны» и с данным окружением просто делают то, что нужно. Затем вы просто «обходите» список продукций грамматики, вызывая `run` до тех пор, пока они не закончатся.

Выбранный вами способ определит, где вы должны хранить состояние для вашего маленького интерпретатора. Если вы создаете класс `Interpreter`, который просто выполняет объекты продукций, тогда `Interpreter` может отслеживать все состояния и выступать в роли компьютера. Но в этом случае язык сложнее расширить, поскольку вам нужно совершенствовать `Interpreter` для каждого класса продукции. Если ваши классы продукции знают, как выполнять свой код, тогда язык расширить несложно, но нужно найти способ передавать состояние компьютера между каждой продукцией.

Я предлагаю начать работу над упражнением с крошечного выражения, например, прибавления. Начните с него, а затем разработайте всю систему, от лексического анализатора до выполнения простого прибавления. Затем, если вам не понравится этот вариант разработки, вы можете удалить его и начать по-другому. Когда ваше проектирование будет работать, можете добавить в язык большее количество функций.

## Практические задания

1. Лучшее практическое задание для этого – создание функций для выполнения вычислений и возврата результатов. Если вы можете это реализовать, значит, ваше проектирование, вероятно, будет работать с более крупным языком.

2. Следующее, что нужно попробовать, – это реализовать управление потоком с помощью конструкций `if` и логических проверок. Ничего страшного, если это покажется слишком сложным, просто попробуйте.

## Дальнейшее обучение

Узнайте как можно больше о команде `bc` или языке Python. Попробуйте найти и изучить другие файлы грамматики, особенно описания протоколов IETF. По увлекательности спецификации IETF могут сравниться с туалетной бумагой, но читать их — полезная практика.

# Немного Бейсика

Теперь вы отправитесь в прошлое во времена моего детства и реализуете интерпретатор Бейсик (BASIC). Под Бейсиком я подразумеваю не «действительно простой базовый интерпретатор». Я имею в виду язык программирования Бейсик. Это был один из самых первых языков программирования, первоначально созданный Джоном Кемени и Томасом Курцем в Дартмуте, США. Эта версия называется Dartmouth BASIC, и код выглядит точно так же, как на соответствующей странице в Википедии (**en.wikipedia.org/wiki/Dartmouth_BASIC**):

```
5 LET S = 0
10 MAT INPUT V
20 LET N = NUM
30 IF N = 0 THEN 99
40 FOR I = 1 TO N
45 LET S = S + V(I)
50 NEXT I
60 PRINT S/N
70 GO TO 5
99 END
```

Эти цифры слева – на самом деле номера строк, введенные вручную. Вы указывали Бейсику номер для каждой строки, и тогда можно было создать цикл, просто сказав «перейти на» (GO TO) эту строку. Позже, в других версиях Бейсика, это выражение превратилось в оператор `GOTO` и стало своего рода символом той компьютерной эпохи.

Более поздние версии Бейсика задокументированы на странице Бейсика в Википедии (**ru.wikipedia.org/wiki/Бейсик**), где показана длительная эволюция языка с переходом к все более и более, скажем так, современным формам. Через некоторое время он стал структурированным языком, как C и Алгол, затем – объектно-ориентированным, и сегодня вы можете найти довольно продвинутые версии Бейсика. Ознакомьтесь с Gambas по адресу **gambas.sourceforge.net/en/main.html**, если хотите увидеть современный бесплатный Бейсик.

# Задача упражнения

Вашей задачей будет реализовать оригинальный интерпретатор Бейсик – тот самый, с ручной нумерацией и текстом, состоящим из прописных букв. Вам нужно будет поискать на основной странице в Википедии возможные токены и пример кода, а также прочесть страницу Dartmouth BASIC для получения дополнительных сведений. Ваш интерпретатор должен иметь возможность обрабатывать как можно больше исходного Бейсика, а также осуществлять допустимый вывод.

Я предлагаю попробовать простые математические операции, вывод и отслеживание номеров строк. После этого я бы поработал над тем, чтобы переход `GOTO` функционировал корректно. Если вы с этим разобрались, можете закончить все остальное и не спеша разработать набор тестовых программ, чтобы убедиться в корректной работе вашего интерпретатора.

Удачи! Это задание может отнять у вас много времени, но оно должно быть очень веселым. Будучи ребенком, я мог тратить месяцы на что-то подобное, просто добавляя наивные возможности, вроде графики, чтобы создавать все эти небольшие бестолковые программы. Я написал невероятно много кода на Бейсике, и подсчет номеров строк определенно покоробил мне мозг. Наверное, поэтому мне так нравится редактор Vim.

# Практические задания

Это упражнение и так непростое, но если вам нужны дополнительные задания, сделайте следующее.

1. Создайте альтернативный интерпретатор, который использует генератор парсеров вроде SLY (**github.com/dabeaz/sly**). Когда у вас будет готова РФБН, это может стать намного проще, но с языком вроде Бейсика все сложнее. Вам придется сделать все самостоятельно.

2. Попробуйте сделать версию «структурированного Бейсика», в которой есть функции, циклы, конструкции `if` и все, что можно найти в более старом, не объектно-ориентированном языке, например C или Паскале. Это сложнейшая задача, поэтому я предлагаю вам не пытаться написать парсер MPC вручную. Используйте инструмент SLY, чтобы создать парсер и сохранить свой мозг для более важных вещей.

# Часть VI

## SQL и объектно-реляционное отображение

В этой части мы рассмотрим очень важную тему, необходимую для начинающих разработчиков, хоть и не совсем вписывающуюся в общую структуру книги. Понимание того, как структурировать данные в базе данных SQL (произносится «эс-кью-эл»), позволит вам логически подходить к вопросу требований к хранилищу данных. Существует давно устоявшийся метод деконструкции данных, эффективного хранения и доступа к ним. В последние годы разработка баз данных NoSQL («не только SQL») изменила этот аспект, но базовые понятия, лежащие в основе проектирования реляционной базы данных, по-прежнему полезны. Везде, где вам приходится хранить данные, их необходимо правильно структурировать и понимать.

В большинстве этих упражнений вы будете использовать базу данных SQL, поэтому для работы я рекомендую загрузить двоичный файл `sqlite3` из базы данных SQLite3 (**www.sqlite.org/download.html**), если у вас его еще нет. Мы используем Python, поэтому в большинстве случаев он уже установлен, но иногда оказывается недоступен. Попытайтесь запустить этот код в своей оболочке Python:

```
>>> import sqlite3
```

Если вы получили ошибку, значит `sqlite3` не встроен в ваш Python по умолчанию. Нужно выяснить, почему он отсутствует, и тогда, скорее всего, вам придется установить дополнительный пакет, прежде чем вы сможете использовать `sqlite3` внутри Python.

## Понимать SQL – это понимать таблицы

Прежде чем приступить к упражнениям этой части, вам нужно хорошо понять одну концепцию, которая вызывает множество проблем у начинающих изучать SQL:

```
КАЖДЫЙ ЭЛЕМЕНТ В БАЗЕ ДАННЫХ SQL – ТАБЛИЦА.
```

Заучите это! Под «таблицей» я имею в виду обычную таблицу, где слева направо расположены строки, и сверху вниз – столбцы. Обычно вы называете столбцы по типу данных, которые находятся там. Затем каждая строка представляет сущность, которую нужно поместить в таблицу. Это может быть учетная запись, список людей и информация о них, кулинарные рецепты или даже автомобили. Каждая строка – это автомобиль, а каждый столбец – некое свойство этого автомобиля, которое вы хотите отследить.

У большинства программистов с этим возникают проблемы, потому что мы мыслим понятиями древовидных структур. Объект и другой объект внутри него, хранящий список, содержащий словарь, заключающий строки, сопоставленные с данными. Мы вкладываем вещи внутрь вещей, и этот стиль структуры данных не согласуется с таблицей. Большинству программистов кажется, что эти две структуры (таблицы и деревья) не могут сосуществовать, но на самом деле дерево и таблица очень похожи. Вы можете взять почти любую древовидную структуру и сопоставить ее практически с любой матрицей, но для этого вам нужно изучить еще один аспект баз данных SQL: отношение.

Отношение – это то, благодаря чему база данных SQL имеет гораздо больше возможностей, чем таблица. Таблица позволяет создать полный набор листов и поместить в них различные типы данных, но потом эти листы трудно скомпоновать. Все предназначение базы данных SQL сводится к тому, чтобы связывать таблицы вместе при помощи либо столбцов, либо других таблиц. Гениальность этой концепции в том, что она использует одну структуру (таблицу), чтобы затем построить практически любую вообразимую структуру данных, позволяя связать их вместе.

Мы изучим отношения в базе данных SQL, но краткий ответ заключается в том, что если вы способны создать дерево данных, вы можете поместить это дерево в одну или несколько таблиц. На данном этапе возможно сократить процесс преобразования набора взаимосвязанных классов Python в таблицы SQL следующим образом.

1. Создайте таблицы для всех классов.

2. Установите `id` строки в дочерних элементах, указывающие на родителей.

3. Создайте связывающие таблицы «между» любыми двумя классами, которые связаны посредством списка.

На самом деле все намного сложнее, но это суть ваших действий при конвертации набора классов в SQL. Фактически большинство из того, чем занимаются системы вроде Django является усложненной версией трех вышеупомянутых действий.

## Что вы изучите

Цель этого раздела не в том, чтобы обучить вас делу системного администратора SQL. Если вы хотите выполнять эту работу, я предлагаю узнать о Unix все, что сможете, а затем пройти обучение в крупной компании, предлагающей сертификацию в популярной технологии. Имейте в виду, что это не самая увлекательная работа – она похожа на уход за огромным питомником кошек. Кошек, не котят.

К концу части VI вы будете на базовом уровне понимать принцип работы SQL. Это интенсивный экспресс-курс по SQL, который заканчивается тем, что вы создаете собственный объектно-реляционный проектор (англ. object-relational mapper, ORM), подобный тому, что использует Django. Данный раздел

является лишь отправной точкой в осознании того, как работает SQL. Цель раздела – предоставить вам достаточно информации для понимания, что происходит в системах вроде Django.

Если вы хотите как следует изучить SQL, я рекомендую воспользоваться книгой Джо Селко «*SQL для профессионалов*»[7]. Книга Джо огромная, но она охватывает все необходимое, а сам автор – мастер в SQL; прочтение этой книги значительно улучшит ваши навыки.

[7] Книга издана на русском языке в 2009 году в издательстве Лори. – *Прим. ред.*

# Введение в SQL

Лучший способ научиться моделировать и проектировать надежные данные – начать с базовых строительных блоков. «Стиль» базы данных SQL был стандартом для моделирования и хранения данных в течение многих десятилетий. Если вы знаете базовый SQL, вы можете легко использовать практически любую из существующих систем NoSQL или систем объектно-реляционного отображения (ORO). SQL – это крайне формальный способ хранения, управления и доступа к данным. Именно так к нему и следует относиться. Он не очень сложен, так как не является полным по Тьюрингу, в отличие от полноценных языков программирования.

SQL встречается повсюду, и я говорю об этом не потому, что хочу, чтобы вы его использовали. Это просто факт. Держу пари, что SQL прямо сейчас «лежит» у вас в кармане. У всех iPhone и телефонов на Android есть простой доступ к базе данных SQL под названием SQLite, и многие приложения на вашем телефоне используют ее напрямую. SQL используется банками, больницами, университетами, правительствами, предприятиями малого и крупного бизнеса; почти каждый компьютер и каждый человек на планете прикасается к чему-то, работающему с SQL. SQL – невероятно успешная и надежная технология.

Проблема с SQL заключается в том, что, кажется, никто не переносит его внутренностей. Это странный бестолковый тип «не языка», который большинство программистов терпеть не может. Он разработан задолго до того, как вообще появились все эти современные проблемы вроде Web-Scale или объектно-ориентированного программирования. Несмотря на то что он имеет математическую основу, слишком много в нем устроено неправильно и легко вызывает раздражение. Деревья? Вложенные объекты и взаимосвязь «родитель – дочерний элемент»? SQL просто смеется вам в лицо и выдает массивную плоскую таблицу, говоря: «Сам разберешься, дружище».

Почему же вы должны изучать SQL? Движение NoSQL частично является реакцией на устаревшие серверы баз данных, а также ответом на страх перед SQL, проистекающий из незнания того, как он работает. Изучая SQL, вы на самом деле изучите важные теоретические концепции, которые могут быть применены практически к каждой системе хранения данных прошлого и настоящего.

Независимо от того, что утверждают противники SQL, вы должны изучить его, потому что он повсюду и в действительности не так уж сложен. Когда вы станете компетентным пользователем SQL, это поможет вам принимать обоснованные решения о том, какие базы данных использовать, пользоваться ли SQL или нет. У вас появится более глубокое понимание многих систем, с которыми вы работаете как программист.

## Что такое SQL?

SQL расшифровывается как structured query language – язык структурированных запросов. Однако это значение неинтересно, так как оно было всего лишь маркетинговой уловкой. На самом деле SQL предоставляет вам язык для взаимодействия с информацией в базе данных. Его преимущество состоит в том, что он точно соответствует установленному принципу, определяющему свойства хорошо структурированных данных. SQL отвечает ему не полностью (на что сетуют некоторые недоброжелатели), но достаточно полно, чтобы быть полезным инструментом.

Работа SQL основана на понимании полей в таблицах и на принципах поисках данных в соответствии с содержимым этих полей. Все операции SQL принадлежат к одной из четырех общих операций, совершаемых с таблицами.

**Create** (Создать)

Помещение данных в таблицу.

**Read** (Прочесть)

Запрос данных из таблицы.

**Update** (Обновить)

Изменение данных, уже находящихся в таблице.

**Delete** (Удалить)

Удаление данных из таблицы.

Это считают фундаментальным набором функций, которыми должна располагать каждая система хранения данных; их обозначают акронимом CRUD.

Вообще, если вы не способны выполнить одну из этих четырех операций, у вас должна быть очень уважительная причина.

Мне нравится объяснять принцип функционирования SQL, сравнивая его с программным обеспечением для работы с электронными таблицами, таким как Excel:

- База данных представляет собой весь файл электронной таблицы.
- Таблица представляет собой вкладку или лист электронной таблицы, каждому из которых дается имя.
- Столбец – это столбец и там, и там.
- Строка – это строка и там, и там.
- Затем SQL дает вам язык для выполнения над ними операций CRUD, чтобы создавать новые таблицы и изменять существующие.

Последний пункт крайне важен, из-за его непонимания могут возникнуть проблемы. SQL работает только с таблицами, и каждая операция создает таблицы. SQL либо «создает» таблицу, изменяя существующую, либо возвращает новую временную таблицу в качестве набора ваших данных.

В процессе чтения этой книги вы начнете сознавать значимость такого подхода. К примеру, одна из причин, по которой объектно-ориентированные языки несовместимы с базами данных SQL, заключается в том, что языки ООП организованы вокруг графов, а SQL хочет возвращать только таблицы. Так как почти любой граф можно сопоставить с таблицей, и наоборот, все работает, но на язык ООП взваливается бремя по осуществлению перевода. Если бы SQL возвращал вложенную структуру данных, это не было бы проблемой.

# Настройка

В этом разделе мы будем использовать SQLite3 в качестве учебного инструмента. SQLite3 – это полноценная система баз данных, преимущество которой заключается в том, что она практически не требует установки. Вы просто загружаете двоичный файл и работаете с ним, как с большинством других сценарных языков. Благодаря этому вы сможете изучить SQL, не погрязнув в болоте администрирования сервера баз данных.

Установка SQLite3 проста. Можете воспользоваться одним из следующих решений.

- Перейдите на страницу загрузки SQLite3 (**www.sqlite.org/download.html**) и выберите двоичный файл для своей платформы. Ищите пункт Precompiled Binaries for X (Предварительно скомпилированные бинарные файлы), где X – ваша операционная система.

- Используйте диспетчер пакетов вашей операционной системы для установки. Если у вас Linux, вы знаете, что это значит. Если у вас macOS, сначала обзаведитесь диспетчером пакетов, а затем используйте его для установки SQLite3.

Когда вы выполнили установку, убедитесь, что можете запустить командную строку и использовать SQLite3. Вот вам быстрый тест:

```
$ sqlite3 test.db
SQLite version 3.7.8 2011-09-19 14:49:19
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table test (id);
sqlite> .quit
```

Посмотрите, есть ли здесь файл `test.db`. Если все в порядке, значит, настройка завершена. Вы также должны убедиться, что ваша версия SQLite3 не ниже моей версии 3.7.8. Иногда на старых версиях не все работает корректно.

# Изучаем словарь SQL

Чтобы начать изучение SQL, вам нужно создать флеш-карточки (или использовать программу Anki) для запоминания этих терминов SQL. В последующих упражнениях вы изучите и примените каждое из этих понятий к различным задачам. Изучая язык SQL, лучше всего представлять, что все в нем сводится к операциям `Create`, `Read`, `Update` или `Delete`. Даже если это слово `INSERT`, представляйте его в виде операции `Create`, поскольку оно создаст данные. Поначалу просто потратьте некоторое время на запоминание этих слов и держите их в уме в процессе выполнения упражнений этой главы.

**CREATE**

Создает таблицу базы данных, которая может хранить столбцы данных.

**INSERT**

Добавляет строки в таблицу базы данных, заполняя столбцы данных.

**UPDATE**

Изменяет один или несколько столбцов в таблице.

**DELETE**

Удаляет строку из таблицы.

**SELECT**

Делает запрос к таблице или набору таблиц и возвращает временную таблицу с результатами.

**DROP**

Уничтожает таблицу.

**FROM**

Часть запроса SQL, определяющая, какие таблицы или столбцы следует использовать.

**IN**

Используется для обозначения набора элементов.

**WHERE**

Используется с запросами, чтобы указать, откуда что-то нужно получить.

**SET**

Используется с обновлением, чтобы указать значения столбцов.

# Грамматика SQL

Вам понадобится еще один набор карточек для важных грамматических конструктов SQL. Их не так уж много, но запишите их (или используйте Anki) и начните зубрить прямо сейчас, чтобы быстрее изучить язык. Изучаемая вами грамматика предназначена для базы данных SQLite3, которую мы будем использовать в этой книге. Это довольно распространенная грамматика SQL, но у каждой базы данных есть свой странный «диалект», в котором вам нужно разбираться. После этого вам будет легко понять, что использует другая база данных.

Чтобы создать нужные вам карточки, вы должны будете посетить страницу о SQLite 3 (**sqlite.org/lang.html**). На этой странице перечислено все, что понимает SQLite. Заострите внимание на основных позициях, перечисленных выше. Добавьте любые другие слова, которые не понимаете. Их диаграммы немного запутанны, но это просто графические представления ФБН SQL, которую вы изучали в части V. Если вы не помните РФБН, вернитесь к части V и восстановите ее в памяти.

# Дальнейшее обучение

1. Перейдите к списку грамматики SQLite3 и просмотрите все доступные команды. Большинство из них будут бессмысленными, но если какие-нибудь вас заинтересуют, сделайте флеш-карточки и для них.

2. Тщательно изучайте свои карточки в течение всего времени, что выполняете оставшиеся упражнения.

# Создание в SQL

В акрониме CRUD буква C означает create (создать), но это не просто создание таблиц. Здесь также имеется в виду вставка данных в таблицы и использование таблиц и вставок для связывания (компоновки) таблиц. Поскольку нам нужны таблицы и определенные данные для остальной части CRUD (операций read, update, delete), мы начнем с изучения того, как выполнять базовые операции создания в SQL.

## Создание таблиц

Во вступлении в упражнении 38 я упомянул, что вы можете выполнять операции CRUD с данными внутри таблиц. Как нам создать таблицу? Путем выполнения CRUD со схемой базы данных, и первая инструкция SQL, которую мы рассмотрим, это `CREATE`:

ex1.sql

```
1    CREATE TABLE person (
2        id INTEGER PRIMARY KEY,
3        first_name TEXT,
4        last_name TEXT,
5        age INTEGER
6    );
```

Можно вместить все это в одну строку, но я разделил ее на несколько, потому что хочу рассмотреть каждую из них.

Строка 1

> Начало `CREATE TABLE` (создать таблицу), дающее нам имя таблицы – `person`. Далее вы помещаете в скобки нужные поля.

Строка 2

> Столбец `id`, который будет использоваться для точной идентификации каждой строки. Формат столбца – `NAME TYPE` (имя, тип), и в этом случае

я указываю, что мне нужно `INTEGER` (целое число), а также `PRIMARY KEY` (первичный ключ). Это означает, что SQLite3 должна относиться к этому столбцу по-особенному.

Строки 3–4

Столбцы `first_name` и `last_name`, тип обоих – `TEXT` (текст).

Строка 5

Столбец `age`, это просто `INTEGER`.

Строка 6

Завершение списка столбцов закрывающей скобкой и точкой с запятой.

## Создание многотабличной базы данных

Создание одной таблицы не слишком полезно. Теперь я хочу, чтобы вы создали три таблицы, в которых можно хранить данные.

ex2.sql

```
 1  CREATE TABLE person (
 2      id INTEGER PRIMARY KEY,
 3      first_name TEXT,
 4      last_name TEXT,
 5      age INTEGER
 6  );
 7
 8  CREATE TABLE pet (
 9      id INTEGER PRIMARY KEY,
10      name TEXT,
11      breed TEXT,
12      age INTEGER,
13      dead INTEGER
14  );
15
16  CREATE TABLE person_pet (
17      person_id INTEGER,
18      pet_id INTEGER
```

```
19    );
```

В этом файле вы создаете таблицы для двух типов данных, а затем «связываете» их вместе с помощью третьей таблицы. Такие «связи» таблиц принято называть «отношениями». Слишком педантичные люди без какой-либо личной жизни называют «отношениями» все таблицы, чем путают людей, которые просто хотят выполнить свою работу. В этой книге таблицы, содержащие данные, называются «таблицами», а таблицы, которые связывают таблицы, называются «отношениями».

Здесь нет ничего нового, кроме того, что когда вы взглянете на `person_pet`, то увидите, что я создал два столбца: `person_id` и `pet_id`. Две таблицы можно связать вместе, просто вставив в `person_pet` строку, содержащую значения столбцов `id` двух строк, которые вы хотели бы соединить. Например, если бы таблица `person` («человек») содержала строку с `id=20`, а `pet` («питомец») содержала строку с `id=98`, тогда чтобы сказать, что человек владеет этим питомцем, вы бы должны были вставить `person_id=20`, `pet_id=98` в отношение (таблицу) `person_pet`.

Подобным образом мы будем вставлять данные в следующих нескольких упражнениях.

# Вставка данных

У вас есть несколько таблиц, с которыми можно работать, поэтому теперь я хочу, чтобы вы поместили в них данные при помощи команды `INSERT`:

ex3.sql

```
1   INSERT INTO person (id, first_name, last_name, age)
2       VALUES (0, "Zed", "Shaw", 37);
3
4   INSERT INTO pet (id, name, breed, age, dead)
5       VALUES (0, "Fluffy", "Unicorn", 1000, 0);
6
7   INSERT INTO pet VALUES (1, "Gigantor", "Robot", 1, 1);
```

В этом файле я использую две разные формы команды `INSERT`. Первая форма более явная и, наверное, ею вам и следует пользоваться. Она указывает

столбцы, которые будут вставлены, затем VALUES (значения) и данные для включения. Оба этих списка (имена столбцов и значения) помещаются в круглые скобки и разделяются запятыми.

Вторая версия команды (строка 7) – сокращенная. Она не указывает столбцы и вместо этого полагается на неявный порядок в таблице. Эта форма ненадежна, так как вы не знаете, к какому столбцу на самом деле обращается ваша инструкция, а некоторые базы данных не предполагают надежного упорядочивания столбцов. Используйте эту форму, только если вам чрезвычайно лень.

# Вставка ссылочных данных

В предыдущем разделе вы заполнили таблицы людьми и питомцами. Единственное, чего не хватает, это информации о том, кому принадлежат какие питомцы – эти данные помещаются в таблицу person_pet следующим образом:

ex4.sql

```
1   INSERT INTO person_pet (person_id, pet_id) VALUES (0, 0);
2   INSERT INTO person_pet VALUES (0, 1);
```

Здесь я опять сначала использую явный формат, затем неявный. Я использую нужные значения id из строки человека (в данном случае, 0) и нужные id из строк питомцев (0 для Unicorn («единорог») и 1 для dead Robot («мертвый робот»)). Затем я вставляю в таблицу отношения person_pet по одной строке для каждой «связи» между человеком и питомцем.

# Задача упражнения

1. Создайте еще одну базу данных, содержащую другие поля с типами INTEGER и TEXT для чего-то, чем может владеть человек.

2. В этих таблицах я сделал третью таблицу отношения, чтобы связать их. Как бы вы избавились от этой таблицы отношения person_pet и поместили эту информацию прямо в person? Каково следствие этого изменения?

3. Если вы можете поместить одну строку в `person_pet`, можете ли вы поместить несколько строк? Как бы вы создали сумасшедшего любителя кошек с 50 кошками?

4. Создайте еще одну таблицу для автомобилей, которыми могут владеть люди, и создайте соответствующую таблицу отношения.

5. Введите в свой любимый поисковик «типы данных sqlite3» и ознакомьтесь с документом Datatypes In SQLite Version 3 (Типы данных в SQLite версии 3) по адресу **sqlite.org/datatype3.html**. Делайте заметки о том, какие типы вы можете использовать, и о других вещах, которые вам кажутся важными. Позже мы углубимся в эту тему.

6. Вставьте (внесите в таблицу) себя и своих питомцев (или воображаемых питомцев, как я).

7. Если во втором задании вы изменили базу данных, чтобы обойтись без таблицы `person_pet`, тогда создайте новую базу данных с этой схемой и вставьте в нее ту же информацию.

8. Вернитесь к списку типов данных и сделайте заметки о том, какие форматы нужны для разных типов. Например, обратите внимание на то, сколькими способами можно указывать данные типа TEXT.

9. Добавьте отношения для вас и ваших питомцев.

10. Может ли питомец, основываясь на этой таблице, принадлежать более чем одному человеку? Возможно ли это логически? А как насчет собаки в семье? Разве технически не все в семье владеют этой собакой?

11. Учитывая вышеизложенное и то, что у вас есть свой проект, в котором `pet_id` помещается в таблицу `person`, какая схема БД лучше подходит в этой ситуации?

# Дальнейшее обучение

Прочтите документацию для команды SQLite3 `CREATE` по адресу **sqlite.org/lang_createtable.html**, а затем просмотрите как можно больше других инструкций `CREATE`. Вы также должны прочесть документацию по `INSERT` на странице **sqlite.org/lang_insert.html**, которая подведет вас к прочтению множества других страниц.

УПРАЖНЕНИЕ 40

# Чтение в SQL

Из операций CRUD вам знакомо только создание. Вы можете создавать таблицы и создавать строки в этих таблицах. Теперь я покажу вам, как можно прочесть, или в случае с SQL, `SELECT` («выбрать»):

ex5.sql

```
1    SELECT * FROM person;
2
3    SELECT name, age FROM pet;
4
5    SELECT name, age FROM pet WHERE dead = 0;
6
7    SELECT * FROM person WHERE first_name != "Zed";
```

Вот что делает каждая из этих строк.

Строка 1

> Она говорит: «Выберите все столбцы таблицы `person` и верните все строки». Структура инструкции `SELECT` такова: `SELECT что_выбрать FROM таблица(ы) WHERE (тесты)`. Конструкция `WHERE` необязательна. Символ * (астериск) означает, что вы хотите выбрать все столбцы.

Строка 3

> Здесь я прошу только два столбца, `name` и `age`, из таблицы `pet`. Это вернет все строки.

Строка 5

> Теперь я ищу те же столбцы из таблицы `pet`, но прошу только те строки, где `dead = 0`. Это дает мне всех питомцев, которые живы.

Строка 7

Наконец я выбираю все столбцы `person`, как в первой строке, но теперь я прошу только те, что не равны `Zed`. Конструкция `WHERE` определяет, какие строки нужно возвращать, а какие нет.

## Выбор среди множества таблиц

Надеюсь, вы постигаете принцип составления выборки данных из таблиц. Всегда помните о следующем: SQL РАБОТАЕТ ТОЛЬКО С ТАБЛИЦАМИ. SQL ЛЮБИТ ТАБЛИЦЫ. SQL ВЫЗВРАЩАЕТ ТОЛЬКО ТАБЛИЦЫ. ТАБЛИЦЫ. ТАБЛИЦЫ. ТАБЛИЦЫ. ТАБЛИЦЫ![8]

Я повторяю это в такой ненормальной манере, чтобы донести до вас идею: то, что вы уже понимаете в программировании, вам не поможет. В программировании вы имеете дело с графами, а в SQL – с таблицами. Это связанные понятия, но концептуально модели разные.

Вот пример того, где появляются различия. Представьте, что вам нужно узнать, какие у Зеда (`Zed`) есть питомцы. Вам нужно написать инструкцию `SELECT`, которая посмотрит в таблице `person`, а затем «каким-то образом» найдет моих питомцев. Для этого вам нужно запросить таблицу `person_pet`, чтобы получить нужные столбцы `id`. Вот как я это сделал бы.

ex6.sql

```
1    SELECT pet.id, pet.name, pet.age, pet.dead
2        FROM pet, person_pet, person
3        WHERE
4        pet.id = person_pet.pet_id AND
5        person_pet.person_id = person.id AND
6        person.first_name = "Zed";
```

Так, этот фрагмент выглядит огромным, но я разобью его, чтобы вы увидели, что он просто создает новую таблицу на основе данных в трех таблицах и конструкции `WHERE`.

[8] Аллюзия на известную речь Стивена Балмера, в которой он повторял: «Разработчики!». – *Прим. ред.*

Строка 1

Мне нужны только определенные столбцы из `pet`, поэтому я перечисляю их имена в запросе. В последнем упражнении вы использовали символ *, чтобы сказать «все столбцы», но здесь следует поступить иначе. Вам нужно точно указать, какой столбец из каждой таблицы вам нужен, и вы делаете это при помощи синтаксиса `таблица.столбец`, например `pet.name`.

Строка 2

Чтобы связать `pet` с `person`, мне нужно пройти по таблице отношения `person_pet`. В SQL это значит, что мне нужно перечислить все три таблицы после оператора `FROM`.

Строка 3

Начинаю конструкцию `WHERE`.

Строка 4

Сначала я связываю `pet` с `person_pet` соответствующими столбцами `id` `pet.id` и `person_pet.id`.

Строка 5

И (*AND*) аналогичным образом мне нужно связать `person` с `person_pet`. Теперь база данных может искать только те строки, где совпадают все столбцы id, и те, которые являются связанными.

Строка 6

И (*AND*) наконец я прошу только тех питомцев, которые мне принадлежат, добавляя тест (условие) `person.first_name` для моего имени.

# Задача упражнения

1. Напишите запрос, который находит всех питомцев старше 10 лет.

2. Напишите запрос, чтобы найти всех людей моложе вас. Найдите всех, кто старше.

3. Напишите запрос, который использует более одного теста в конструкции `WHERE` (используйте `AND`). Например, `WHERE first_name = "Zed" AND age > 30`.

4. Создайте еще один запрос, который ищет строки при помощи 3 столбцов и использует как инструкцию `AND`, так и `OR`.

5. Если вы уже знаете язык вроде Python или Ruby, этот взгляд на данные может казаться вам странным и даже шокирующим. Уделите время моделированию тех же отношений с помощью классов и объектов, а затем сопоставьте их с данной моделью.

6. Сделайте запрос, находящий ваших питомцев, которых вы добавили до настоящего времени.

7. Измените запросы так, чтобы использовать ваш `person.id` вместо `person.name`, как делал я.

8. Просмотрите свой вывод и убедитесь, что знаете, какая таблица создана какими командами SQL, и как они произвели этот вывод.

## Дальнейшее обучение

Продолжайте изучать SQLite3, прочтя документацию для команды `SELECT` по адресу **sqlite.org/lang_select.html**, а также прочтите документацию по возможности `EXPLAIN QUERY PLAN` по адресу **sqlite.org/eqp.html**. Если вы когда-нибудь задавались вопросом, почему SQLite3 достигла столь широкого распространения, `EXPLAIN` – ваш ответ на этот вопрос.

УПРАЖНЕНИЕ 41

# Обновление в SQL

Теперь в CRUD вам знакома часть CR, так что остаются операции обновления и удаления. Подобно остальным командам SQL, команда `UPDATE` использует формат, подобный `DELETE`, но она изменяет столбцы в строках вместо того, чтобы удалять их.

ex9.sql

```
1   UPDATE person SET first_name = "Hilarious Guy"
2       WHERE first_name = "Zed";
3
4   UPDATE pet SET name = "Fancy Pants"
5       WHERE id=0;
6
7   SELECT * FROM person;
8   SELECT * FROM pet;
```

В приведенном выше коде я меняю свое имя на Hilarious Guy («Весельчак»), так как оно мне больше подходит. И чтобы представить свое новое прозвище, я переименовал своего Unicorn (Единорога) в Fancy Pants («Модные брюки»). Ему понравилось.

Это несложно, но, на всякий случай, я объясню подробнее.

1. Начните с инструкции `UPDATE` и таблицы, которую собираетесь обновлять, в нашем случае с `person`.

2. Затем используйте `SET`, чтобы указать, каким столбцам должны быть присвоены какие значения. Вы можете изменить столько столбцов, сколько хотите; не забудьте разделять их запятыми, например: `first_name = "Zed", last_name = "Shaw"`.

3. Затем укажите условие для конструкции `WHERE`, которая предоставляет инструкции `SELECT` набор тестов для каждой строки. Когда `UPDATE` находит совпадение, она осуществляет обновление и устанавливает (`SET`) столбцы в соответствии с указанными вами параметрами.

## Обновление комплексных данных

В последнем примере вы делали подзапрос в запросе UPDATE. Теперь воспользуемся им, чтобы заменить имена всех питомцев, которые мне принадлежат, названием Zed's Pet.

ex10.sql

```
SELECT * FROM pet;

UPDATE pet SET name = "Zed's Pet" WHERE id IN (
    SELECT pet.id
    FROM pet, person_pet, person
    WHERE
    person.id = person_pet.person_id AND
    pet.id = person_pet.pet_id AND
    person.first_name = "Zed"
);

SELECT * FROM pet;
```

Вот так одна таблица обновляется на основе информации из другой таблицы. Существуют и другие способы сделать то же самое, но прямо сейчас данный способ вам будет понять проще всего.

## Замена данных

Я продемонстрирую вам альтернативный способ вставки данных, который помогает с заменой отдельных строк. Вряд ли вам это будет нужно слишком часто, но это действительно полезно в случае, если нужно заменить целые записи и вы не хотите выполнять более сложное обновление (UPDATE), не прибегая к транзакциям.

В такой ситуации я хочу заменить себя другим парнем, но сохранить уникальный идентификатор (id). Проблема в том, что мне придется либо выполнить DELETE/INSERT в транзакции, чтобы сделать ее атомарной, либо мне нужно будет сделать полное обновление (UPDATE).

Существует более простой способ сделать это – использовать команду REPLACE или добавить ее в качестве модификатора в INSERT. Ниже код SQL,

где сначала мне не удается вставить новую запись, и я использую запрос REPLACE.

ex11.sql

```
/* Так сделать не получится, потому что 0 уже занят. */
INSERT INTO person (id, first_name, last_name, age)
    VALUES (0, 'Frank', 'Smith', 100);

/* Мы можем добиться результата, используя INSERT OR REPLACE. */
INSERT OR REPLACE INTO person (id, first_name, last_name, age)
    VALUES (0, 'Frank', 'Smith', 100);

SELECT * FROM person;

/* Для краткости можем использовать просто REPLACE. */
REPLACE INTO person (id, first_name, last_name, age)
    VALUES (0, 'Zed', 'Shaw', 37);

/* Как видите, теперь я вернулся. */
SELECT * FROM person;
```

# Задача упражнения

1. Используйте UPDATE, чтобы вернуть мне имя Zed с помощью моего person.id.

2. Напишите UPDATE, который переименовывает любых мертвых (dead) животных в DECEASED (скончавшихся). Если вы попытаетесь указать, что они DEAD, ничего не выйдет, потому что SQL подумает, что вы имеете в виду «установите его столбцу с именем DEAD», а это не то, что вам нужно.

3. Попробуйте использовать подзапрос, как в примере с DELETE.

4. Перейдите на страницу SQL As Understood By SQLite (**www.sqlite.org/lang.html**) и начните чтение документации по CREATE TABLE, DROP TABLE, INSERT, DELETE, SELECT и UPDATE.

5. Испробуйте что-нибудь интересное из того, что найдете в этих документах, а также делайте заметки о том, чего вы не понимаете, чтобы исследовать это позже.

## Дальнейшее обучение

Продолжайте изучение языка SQLite3 – прочтите документацию по `UPDATE` на **sqlite.org/lang_update.html** и связанных страницах.

УПРАЖНЕНИЕ 42

# Удаление в SQL

Это самое простое упражнение, но я хочу, чтобы вы на секунду задумались, прежде чем вводить код. Если для SELECT у вас есть «SELECT * FROM», а для INSERT «INSERT INTO», тогда какой будет структура DELETE? Наверное, вы можете просто подсмотреть ответ внизу, но попытайтесь догадаться самостоятельно, а затем посмотрите.

ex7.sql

```
/* убедитесь, что здесь есть мертвые питомцы */
SELECT name, age FROM pet WHERE dead = 1;

/* ах, бедный робот */
DELETE FROM pet WHERE dead = 1;

/* убедитесь, что робот умер */
SELECT * FROM pet;

/* давайте воскресим робота */
INSERT INTO pet VALUES (1, "Gigantor", "Robot", 1, 0);

/* робот ЖИВ! */
SELECT * FROM pet;
```

Я просто реализую очень сложное обновление робота, удаляя его, а затем возвращая запись, но с dead=0. В последующих упражнениях я покажу вам, как для этого использовать UPDATE, поэтому не считайте это настоящим способом делать обновление.

Большинство строк в этом сценарии уже знакомы вам, за исключением строки 5. Здесь у нас инструкция DELETE, формат которой похож на уже виденное нами ранее. Вы пишете DELETE FROM таблица WHERE тесты, и можете смотреть на это как на инструкцию SELECT, которая удаляет строки. Будет удалено все, что соответствует условиям конструкции WHERE.

# Удаление с использованием других таблиц

Помните, я сказал «DELETE похож на SELECT, но он удаляет строки из таблицы». Ограничение состоит в том, что одним запросом вы можете удалять только из одной таблицы. Это означает, что для удаления всех питомцев вам необходимо выполнить некоторые дополнительные запросы, а затем осуществлять удаление, основываясь на них.

Один из способов сделать это – использовать подзапрос, выбирающий идентификаторы, которые вы хотите удалить, на основе уже написанного запроса. Существуют другие решения, но данным способом вы можете воспользоваться прямо сейчас, основываясь на том, что уже знаете.

ex8.sql

```
1    DELETE FROM pet WHERE id IN (
2        SELECT pet.id
3        FROM pet, person_pet, person
4        WHERE
5        person.id = person_pet.person_id AND
6        pet.id = person_pet.pet_id AND
7        person.first_name = "Zed"
8    );
9
10   SELECT * FROM pet;
11   SELECT * FROM person_pet;
12
13   DELETE FROM person_pet
14       WHERE pet_id NOT IN (
15           SELECT id FROM pet
16       );
17
18    SELECT * FROM person_pet;
```

Строки с 1 по 8 – это команда DELETE, которая начинается стандартно, но затем конструкция WHERE использует IN для сопоставления столбцов id в pet с таблицей, возвращаемой в подзапросе. Подзапрос (также называемый подвыборкой) является запросом SELECT, и он выглядит очень похоже на те, что вы делали раньше, когда пытались найти питомцев, принадлежащих людям.

Затем в строках 13–16 я использую подзапрос, чтобы очистить таблицу `person_pet` от любых питомцев, которых больше не существует, используя `NOT IN`, «не `IN`».

SQL делает это следующим образом.

1. Запускает подзапрос в скобках в конце и строит таблицу со всеми столбцами, точно как обычный `SELECT`.

2. Рассматривает эту таблицу как подобие временной таблицы для сопоставления со столбцами `pet.id`.

3. Просматривает таблицу `pet` и удаляет все строки с идентификатором в (`IN`) этой временной таблице.

# Задача упражнения

1. Объедините все файлы, начиная с `ex2.sql`, и по `ex7.sql` в один файл и переделайте описанный выше сценарий, чтобы можно просто запустить этот новый файл для воссоздания базы данных.

2. Добавьте в сценарий возможность удаления других домашних животных и их повторной вставки с новыми значениями. Помните, что обычно записи обновляются по-другому, а этот способ – только для упражнения.

3. Попрактикуйтесь в написании команд `SELECT`, а также помещайте их внутрь `DELETE WHERE IN`, чтобы удалять найденные записи. Попробуйте удалить всех принадлежащих вам мертвых питомцев.

4. Сделайте наоборот: удалите людей, у которых есть мертвые питомцы.

5. Действительно ли вам нужно удалять мертвых питомцев? Почему бы просто не удалить их отношения в `person_pet` и отметить их мертвыми? Напишите запрос, который убирает мертвых питомцев из `person_pet`.

## Дальнейшее обучение

Для полноты картины вам нужно прочесть документацию о команде `DELETE` по адресу **sqlite.org/lang_delete.html**.

УПРАЖНЕНИЕ 43

# Администрирование SQL

В базах данных под словом «администрирование» имеют в виду много различных функций. Оно может означать «проверка, что сервер PostgreSQL продолжает работать корректно», оно также может означать «изменение и миграция таблиц для развертывания нового программного обеспечения». В этом упражнении я расскажу только о том, как выполнять простые изменения и миграцию схемы баз данных. Управление полноценным сервером базы данных выходит за рамки этой книги.

## Уничтожение и изменение таблиц

Вам уже знакома инструкция `DROP TABLE` — с ее помощью можно избавиться от созданной таблицы. Я покажу вам другой способ использования, а также то, как добавлять и удалять столбцы из таблицы с помощью команды `ALTER TABLE`.

ex12.sql

```
/* Удалить таблицу, только если она существует. */
DROP TABLE IF EXISTS person;

/* Снова создать ее, чтобы работать с ней. */
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);

/* Переименовать таблицу в peoples. */
ALTER TABLE person RENAME TO peoples;

/* Добавить в таблицу peoples столбец hatred. */
ALTER TABLE peoples ADD COLUMN hatred INTEGER;

/* Переименовать peoples обратно в person. */
ALTER TABLE peoples RENAME TO person;
```

```
20
21    .schema person
22
23    /* Нам это не нужно. */
24    DROP TABLE person;
```

Я вношу некоторые фиктивные изменения в таблицы, чтобы продемонстрировать работу команд, но это все, что вы можете сделать в SQLite3 с инструкциями `ALTER TABLE` и `DROP TABLE`. Давайте построчно разберем все SQL запросы.

Строка 2

Я использую модификатор `IF EXISTS`, чтобы удалить таблицу, только если она уже существует. Это предотвращает ошибку, возникающую при запуске сценария .sql в новой базе данных, в которой нет таблиц.

Строка 5

Просто заново создаю таблицу, чтобы работать с ней.

Строка 13

Использую `ALTER TABLE`, чтобы переименовать ее в `peoples`.

Строка 16

Добавляю новый столбец `hatred`, это `INTEGER` в только что переименованной таблице `peoples`.

Строка 19

Переименовываю `peoples` обратно в `person`, потому что это дурацкое название для таблицы.

Строка 21

Отображаю схему для `person`, чтобы вы увидели, что в этой таблице есть новый столбец `hatred`.

Строка 24

Удаляю таблицу, прибирая за собой после упражнения.

# Миграция и развитие данных

Давайте применим некоторые из приобретенных вами навыков. Я попрошу вас взять вашу базу данных и «развить» схему до другой формы. Вам нужно убедиться, что вы хорошо разобрались в предыдущем упражнении, и ваш файл `code.sql` работает как надо. Если какое-то из этих условий не было выполнено, вернитесь и исправьте ситуацию.

Чтобы убедиться, что вы готовы попытаться выполнить это упражнение, запустите свой `code.sql`. Ваша `.schema` должна запускаться следующим образом:

Exercise 13 Session

```
$ sqlite3 ex13.db < code.sql
$ sqlite3 ex13.db .schema
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);
CREATE TABLE person_pet (
    person_id INTEGER,
    pet_id INTEGER
);
CREATE TABLE pet (
    id INTEGER PRIMARY KEY,
    name TEXT,
    breed TEXT,
    age INTEGER,
    dead INTEGER,
    dob DATETIME
);
```

Убедитесь, что ваши таблицы выглядят так же, как мои. Если они отличаются, вернитесь и удалите из последнего упражнения любые команды, касающиеся `ALTER TABLE` или других изменений.

# Задача упражнения

Вам нужно внести в базу данных следующие изменения.

1. Добавьте в таблицу `person` столбец `dead`, аналогичный столбцу в таблице `pet`.

2. Добавьте в `person` столбец `phone_number`.

3. Добавьте в `person` столбец `salary` с типом `float`.

4. Добавьте столбец `dob` (тип `DATETIME`) и в `person`, и в `pet`.

5. Добавьте столбец `purchased_on` (тип `DATETIME`) в `person_pet`.

6. Добавьте столбец `parent` в `pet`, который является `INTEGER` и содержит `id` родителя этого питомца.

7. Обновите существующие записи базы данных с учетом данных новых столбцов при помощи инструкций `UPDATE`. Не забудьте о столбце `purchased_on` («приобретен в») в таблице отношения `person_pet`, чтобы указать, когда человек купил питомца.

8. Добавьте еще четверых человек и пятерых питомцев, назначьте владение, а также укажите, какие питомцы являются родителями. В этой последней части помните, что вы получаете идентификатор родителя, а затем устанавливаете его в столбце `parent`.

9. Напишите запрос, который сможет найти все имена домашних питомцев, купленных после 2004 года, и их владельцев. Ключом к этому является сопоставление (мапирование) `person_pet`, основанной на столбце `purchased_on`, с `pet` и `parent`.

10. Напишите запрос, который сможет найти питомцев, являющихся детьми данного питомца. Для этого снова посмотрите на `pet.parent`. Это на самом деле легко, так что не стоит размышлять об этом слишком долго.

11. Обновите файл `code.sql`, в котором у вас содержится весь код, так, чтобы был использован синтаксис `DROP TABLE IF EXISTS`.

12. Используйте `ALTER TABLE`, чтобы добавить столбцы `height` и `weight` в `person`, и поместите код запросов в свой файл `code.sql`.

13. Выполните обновленный сценарий `code.sql`, чтобы с нуля построить базу данных. У вас не должно быть ошибок.

Вы должны сделать это путем написания файла `ex13.sql` с вышеперечисленными нововведениями. Затем проверьте его путем сброса базы данных с помощью `code.sql` и последующего выполнения `ex13.sql` для изменения базы данных. Выполните запросы `SELECT`, подтвердив, что вы правильно внесли изменения.

# Дальнейшее обучение

Продолжите чтение документации по `DROP TABLE` (**sqlite.org/lang_droptable.html**) и `ALTER TABLE` (**sqlite.org/lang_altertable.html**), а затем перейдите на страницу языка SQLite3 (**sqlite.org/lang.html**) и прочтите документацию для оставшихся инструкций – `CREATE` и `DROP`.

# Использование API баз данных Python

Python имеет стандартизованный API баз данных, который позволяет использовать один и тот же код для доступа к различным базам данных. У каждой базы данных, к которой вы хотите подключиться, есть свой модуль, позволяющий работать с ней, и соответствующий стандарту PEP 249 (**www.python.org/dev/peps/pep-0249/**). Это упрощает работу с базами данных с разными API для доступа. Для работы с SQL в этом упражнении вы будете использовать модуль `sqlite3`, описание которого находится по адресу **docs.python.org/2/library/sqlite3.html**.

## Изучение API

Чем вы наверняка будете постоянно заниматься как программист, так это изучать API, написанные другими людьми. Я специально не описывал наиболее эффективный способ делать это, потому что большинство программистов просто находят его в процессе изучения языка. Язык Python и его модули взаимосвязаны настолько тесно, что при изучении Python просто невозможно не изучать API в этих модулях. В то же время существует эффективный способ изучения API, который использую я, и в этом упражнении вы узнаете о нем.

Чтобы изучить API, вроде модуля `sqlite3`, я бы сделал следующее.

1. Нашел бы всю документацию для API, а если документации не существует, нашел бы код.

2. Просмотрел бы образцы кода или тестовый код и воссоздал их в собственных файлах. Одного чтения обычно недостаточно. Я действительно заставляю код работать, потому что – угадайте-ка – часто документация не работает с текущей версией API. Реализация всего, что указано в документации, помогает мне отыскать то, о чем там не было упомянуто.

3. Запуская образцы кода на своем компьютере, я делал бы заметки по любым ситуациям типа «у меня работает». Так называются ситуации,

которые возникают, когда человек, писавший документацию, не указал важные шаги по настройке, поскольку его компьютер уже был настроен. Большинство программистов, пишущих документацию, не работают на «чистых» компьютерах, поэтому забывают, что библиотеки и программное обеспечение, которые установлено у них, часто отсутствует у остальных. Эти пробелы доставят мне неприятности позже, когда я попытаюсь отправить API в производство, так что я отмечаю их на будущее.

4. Сделал бы флеш-карточки или заметки для всех основных точек входа API и того, что они делают.

5. Попытался бы написать небольшой тестовый «шип», использующий API, но пользуясь только собственными заметками. Если я сталкиваюсь с фрагментом API, который не помню, я возвращаюсь к документации и обновляю свои заметки.

6. Наконец, если API сложно использовать, я перешел бы к простому API, который делает только то, что мне необходимо.

Если этот способ не поможет вам изучить API, найдите другой. Если автор API говорит вам «прочитать код», скорее всего, есть другой проект, с документацией. Используйте тот проект. Если вы обязаны использовать конкретный API, подумайте над тем, чтобы сделать заметки на основе его кода и написать книгу, которую затем продать и заработать на лени автора.

## Задача упражнения

Вы должны изучить API `sqlite3` описанным образом, а затем попытаться написать собственный API для упрощения доступа к базе данных. Имейте в виду, что DB API 2.0 – уже существующий отличный простой API для доступа к базе данных, поэтому вы просто практикуетесь, пытаясь создать свой. Ваша цель должна состоять в том, чтобы изучить API `sqlite3` достаточно хорошо, чтобы затем разработать более простой способ получения доступа.

Иногда понятие «более простой» чисто субъективно или основано на текущих потребностях. Вероятно, вам нужно упростить не то, как общаться с базой данных SQL, а скорее то, как конкретно вам нужно общаться с базой данных SQL. Если вашему приложению нужно иметь дело только с людьми и питомцами, тогда упрощением просто будет создание API, предназначенного только для вас.

## Дальнейшее обучение

Прочтите документацию для других API баз данных в Python. Можете ознакомиться с Pyscopg PostgreSQL (**initd.org/psycopg/docs/**), а также с драйвером Python MySQL (**dev.mysql.com/doc/connectorpython/en/**).

УПРАЖНЕНИЕ 45

# Создание объектно-реляционного менеджера

Это последнее упражнение в части книги, посвященной SQL. Оно станет большим скачком вперед. Вы знаете основы языка SQL, у вас также должен быть опыт объектно-ориентированного программирования на Python. Теперь пришло время объединить это и создать объектно-реляционный менеджер (ORM). Задача ORM состоит в том, чтобы брать классы Python и переводить их в строки, сохраненные в таблицах базы данных. Если вы когда-либо пользовались Django, вы использовали его ORM для хранения своих данных. В этом упражнении вы попытаетесь воспроизвести эту технологию.

## Задача упражнения

В реальном мире, если бы работающий на меня программист попросил создать свой собственный ORM, я ответил бы ему: «Ни за что, используй существующий». Рабочие ситуации отличаются от образовательного процесса, поскольку за работу вам платят деньги. Вы не можете оправдать трату своего рабочего времени на создание вещей, которые, честно говоря, не приносят пользу вашему работодателю. Тем не менее ваше личное время находится полностью в вашем распоряжении и, будучи новичком, вы должны пытаться воссоздать как можно больше классических ситуаций.

Создание ORM укажет вам на множество проблем, связанных с несоответствием между концепциями объектно-ориентированного программирования и SQL. Существует много задач, решение которых может легко смоделировать SQL, но на которых классы обычно спотыкаются. Также существует проблема с тем, что в SQL все является таблицей. Попытка создать собственный ORM позволит узнать столько всего о SQL и ООП. Я рекомендую потратить побольше времени и разработать свой лучший вариант.

Вот некоторые ключевые функции, которые должен иметь ваш ORM.

1. В ваш ORM должно быть возможно безопасно передать строку извне. Используя f-строки для создания SQL, вы поступаете неправильно. Причина в том, что если вы используете `f "SELECT * FROM`

{имя_таблицы}», то кто-то может извне установить имя_таблицы SQL, вроде `person; DROP TABLE`. В таком случае ваша база данных, скорее всего, исполнит запрос и уничтожит все данные, или случится что-нибудь похуже. Некоторые базы данных даже позволяют запускать системные команды внутри SQL. Это называется «инъекцией (внедрением) SQL». В вашем ORM этого быть не должно.

2. Все операции CRUD, но в Python. Я рекомендую вам пропустить часть `CREATE TABLE`, пока не будет реализовано все остальное. Простые `INSERT`, `SELECT`, `UPDATE` и `DELETE` легко создать, но генерация схемы базы данных из определений классов требует определенного шаманства с Python. Используйте созданный вручную файл `.sql` для создания своей базы данных, а затем, когда все остальное будет работать, можете попытаться использовать схему для замены файла `.sql`.

3. Согласование типов Python с типами SQL, а также новые типы для обработки типов SQL. Вам придется немного извернуться, чтобы типы данных Python подошли вашим таблицам SQL. Возможно, это доставит вам слишком много головной боли и в итоге вы создадите собственные типы данных. Это то, что сделал Django.

4. Транзакции – это дополнительная тема, но попытайтесь внедрить их, если они вам понятны.

Стоит также упомянуть, что в этом упражнении вы можете красть функции из скольких угодно проектов. Не стесняйтесь изучать ORM Django при работе над своим проектом. Наконец, я настоятельно рекомендую вам начать с ORM, который работает с небольшой базой данных, созданной вами в этой части книги. Как только у вас получится что-то, работающее с этой базой данных, можете затем переходить к обобщению ORM для работы с любой базой данных.

## Дальнейшее обучение

Как уже упоминалось в начале этой части, книга Джо Селко «*SQL* для профессионалов» обучит вас всему, что нужно знать о SQL. Книга Джо превосходна и выведет вас далеко за пределы этого крошечного экспресс-курса.

# Часть VII

## Финальные проекты

В заключительной части книги вы подойдете к более продвинутым проектам и попытаетесь урегулировать свое персональное движение. Это проекты разного уровня сложности, но они призваны помочь вам формализовать собственное движение и выяснить, что вам подходит лучше всего. Главное – идти по пути анализа того, как вы работаете, и узнавать, что лучше лично для вас. Возможно, вы не следовали всем моим советам по личному развитию, но я надеюсь, что вы продолжите разбирать эту книгу и найдете способы изучить самого себя. Это эффективный способ расти и совершенствоваться как программист.

Давайте рассмотрим, что вы уже изучили до настоящего момента, так как от вас потребуется применить как можно больше в дальнейших упражнениях.

- В первой части вы начали обучение, используя вводный материал.
- В части II вы узнали, как выполнять короткие задания и как лучше всего приступать к работе.

- В части III вы узнали о структурах данных и алгоритмах, а также научились фокусироваться на качестве и писать хорошие тесты.

- В части IV вы применили свои навыки тестирования и качества к проектам, используя разработку через тестирование (TDD) и аудит кода.

- В части V вы узнали об анализе текста и о том, как измерять свое качество в работе и создавать эффективные тесты.

- В части VI вы изучили базы данных SQL и узнали о новом процессе анализа данных и должного их структурирования.

В этой части вы примените все эти знания к набору проектов, сосредоточиваясь на трех областях роста:

1) движении – пытаясь определить свое движение и действовать соответствующим образом;

2) качестве – заострив внимание на автоматизированном тестировании, инструментах тестирования и отслеживании улучшений качества своего труда;

3) креативности – пытаясь решить проблемы, определенные недостаточно четко, и начиная работу с какого-то несерьезного веселого задания.

## Каково ваше движение?

На протяжении всей книги я указывал вам, какие нужно использовать инструменты по улучшению движения. В каждом разделе я давал вам различные задания, посвященные движению, качеству или креативности, а затем – упражнения для работы над этими компонентами. Вы отслеживали качество своей работы и изучали графики, узнавая, что вам помогает, а что нет. Теперь пришло время разработать собственное движение для завершения проекта, а затем применить его к проектам в этом разделе книги.

Уделите время тому, чтобы придумать свою методику движения. Это выполнение задания, затем TDD? Или это постоянная разработка через тестирование и много аудита? Это просто выполнение задания и аудит? Я не имею в виду, что вам необходимо выбрать всего две вещи. Разработайте собственную методику. Думайте об этом, как о личных вкусах и предпочтениях. Мне нравятся шляпы и красные рубашки. Не спрашивайте почему, они мне просто

нравятся. Таким для вас должно выглядеть описание движения – это ваши платья с узором в горошек и желтые туфли в летний день. В программировании я обычно следую методике «выполнить задание, уточнить, проверить, сломать».

После того как вы составили простую формулировку методики, наступает время разработать конкретные шаги. Запишите их на карточке, чтобы им можно было следовать. Чем проще, тем лучше. Над сложным движением тяжело работать. Ваше движение также должно учитывать креативность и качество. Мои способы различны для разных проектов, но в этой книге я описал их все. Используйте то, чему я до сих пор вас научил, чтобы придумать собственные приемы.

Когда вы разработаете свое движение, вернитесь к заметкам и поищите показатели, оправдывающие ваш выбор. Возможно, вы выбрали TDD, потому что так вы чувствовали, что пишете более качественный код, но затем ваши показатели качества в части V были не слишком хороши. Неплохо использовать движение, которое вам нравится, но, если оно не работает, нужно отказаться от него.

Определившись со своим движением, начните работать над некоторыми проектами и проверьте его в деле. Не бойтесь ошибиться. Иногда нам кажется, что наше решение – лучшее, а затем в разгаре битвы оно испаряется, как от взрыва атомной бомбы. Для вас это научный эксперимент, поэтому, если что-то совсем не получается, отследите показатели и узнайте, почему это не получается. Затем внесите изменения и попробуйте снова.

# Инструмент `blog`

Вы должны были определиться со своей методикой движения, как мы обсудили в начале этой части. Ваше движение должно быть расписано по шагам и готово к использованию. Для начала, в качестве разминки перед остальной частью этого раздела, мы создадим совершенно новый инструмент под названием `blog`.

Вы должны выполнять этот проект постепенно, стараясь не спешить. Ваша цель не в том, чтобы быть быстрым программистом. Лучше строить работу плавно, вдумчиво и методично, пока то, как вы работаете, не станет вашей второй натурой. Если вы постоянно спешите, результат будет выглядеть небрежно.

Убедитесь, что ваши заметки под рукой. Отслеживайте данные и показатели своей работы. Пытайтесь найти какой-то элемент движения, который позже можно будет использовать в работе как отдельный метод. Не все методы работают постоянно, поэтому я и пытался научить вас различным тактикам и стратегиям работы, популярным среди программистов. Если вы выполните этот проект и узнаете, что какое-то ваше действие не принесло ожидаемого результата, заметки помогут вам понять причину этого. В следующем проекте, возможно, вы предпочтете другой метод.

## Задача упражнения

Я поручаю вам написать простой инструмент командной строки для блогинга под названием `blog`. Это очень креативное имя для очень креативного проекта. Блоги – это одни из первых проектов, которыми занимаются начинающие программисты, но ваш проект будет генерировать блог локально, а затем отправлять его на сервер с помощью другого инструмента — `rsync`. Требования к этому упражнению следующие.

1. Если вы вообще не знаете, что такое блог, то вам, вероятно, стоит ненадолго попробовать начать вести свой. Для этого существует множество платформ, но, скорее всего, вам подойдет Wordpress или Tumblr. Просто попользуйтесь ими какое-то время и делайте заметки о возможностях, которые вы , вероятно, захотите скопировать. Но не увлекайтесь!

2. Вам захочется узнать, как использовать шаблонизатор для стилизации своих страниц HTML. Я рекомендую использовать шаблонизаторы Mako (**www.makotemplates.org/**) или Jinja (**jinja.pocoo.org/**). Эти системы позволяют создавать файлы шаблонов HTML, в которые затем можно поместить настоящий контент на основе текстовых файлов, размещенных пользователем в каталоге.

3. Вам нужно будет использовать язык разметки Markdown (**pypi.python.org/pypi/Markdown**) для ведения блога, поэтому установите библиотеку `markdown` для этого проекта.

4. Ваш блог будет блогом со статическими файлами, поэтому вам нужно будет выполнить команду `python -m SimpleHTTPServer 8000`, как показано в инструкциях к модулю SimpleHTTPServer (**docs.python.org/2/library/simplehttpserver.html**). Этот сервер будет передавать файлы браузеру из каталога.

5. Для работы с блогом вам понадобится инструмент командной строки под названием `blog`.

6. Прежде чем вы приступите к работе, продумайте все, что должен делать ваш инструмент `blog`, а затем разработайте все необходимые команды и их аргументы. Затем ознакомьтесь с проектом docopt (**github.com/docopt/docopt**) – способом реализации этих команд.

7. Используйте библиотеку `mock` (**pypi.python.org/pypi/mock**), для замены mock-объектами всего, что вам нужно протестировать (особенно на ошибочные условия). Просмотрите видео для данного упражнения, в котором показано использование `mock`.

В остальном вы можете свободно разрабатывать инструмент `blog` так, как сочтете нужным. Используйте творческий подход. Все, что должно получиться, – это блог, который можно разместить на сервере.

Наконец, разместить этот блог онлайн можно при помощи `rsync`, используя следующую команду:

```
rsync -azv dist/* myserver.com:/var/www/myblog/
```

Сейчас вам не помешает научиться помещать на сервер статические файлы. В одном из практических заданий говорится о том, как использовать для этого веб-службу Amazon S3.

## рактические задания

1. Статические файлы на вашем собственном сервере – это прекрасно, но разве не было бы лучше, если бы `blog` работал с облаком Amazon S3? Есть проект под названием Boto3 (**aws.amazon.com/sdk-for-python/**), в котором найдется все необходимое для реализации.

2. Напишите команду `blog serve`, которая использует класс `SimpleHTTPServer`, чтобы просто разместить блог на сервере напрямую и не генерировать его отдельно.

УПРАЖНЕНИЕ 47

# Язык bc

Новый проект в этом упражнении будет непрост. Я предполагаю, что вы будете выполнять его в течение одного или двух дней в виде 2–3-часовых сеансов. Но вы можете тратить столько времени, сколько вам нужно, чтобы реализовать его настолько полно, насколько сможете.

Этот проект посвящен использованию материала, изученного вами в части V, для создания языка для программы `bc`. В упражнении 36 мы уже выполняли простые математические расчеты с использованием команды `bc`, но теперь вам нужно реализовать как можно больше функций языка `bc`. У `bc` есть множество операторов, функций и управляющих структур. Ваша цель – реализовать их поэтапно, используя то, что вы узнали о парсинге методом рекурсивного спуска.

Я бы подошел к созданию парсера поэтапно, начиная с лексического анализа, затем перейдя к синтаксическому анализу, потом к семантическому и использованию образца кода `bc` для его проверки. Этот проект может быть громоздким, так как вы реализуете язык вручную. Тем не менее постарайтесь внедрить как можно больше синтаксиса.

## Задача упражнения

Язык `bc` может больше, чем просто осуществлять математические расчеты. Я никогда не пользуюсь чем-то, кроме простых математических операций, но полноценный язык куда мощнее. У вас есть возможность определять функции, использовать инструкции `if` и реализовывать многие другие распространенные конструкты программирования. В этом упражнении вы не сможете реализовать весь язык `bc`, просто потому что он огромен. Вместо этого вы должны реализовать только следующее.

1. Все математические операторы.
2. Переменные.
3. Функции.
4. Инструкции `if`.

На самом деле это как раз тот порядок, в котором вы, вероятно, и должны реализовывать язык. В первую очередь, реализуйте операторы. Для начала работы можете воспользоваться собственной минималистской реализацией из упражнения 35. После этого реализуйте переменные – для этого потребуется, чтобы ваш лексический анализатор правильно обрабатывал хранение и извлечение значений переменных. Наконец, вы можете реализовать функции, а затем инструкции `if`.

Вам нужно будет раскопать любую документацию по версии GNU `bc`, там вы найдете хорошее полное описание языка, которое поможет его реализовать. В этом нет ничего необычного, так как все в основном скопировано из C.

Когда вы работаете над этой задачей, не спешите и выполняйте все шаги по порядку. Изящность реализации языка заключается в том, что вы можете работать в логическом порядке, переходя от лексического анализа к синтаксическому и далее к семантическому анализу, не «прыгая» между этими тремя этапами туда-обратно.

Наконец, помните, что вы реализуете парсер MPC, который, честно говоря, является лишь детской версией настоящего масштабного анализа, применяемого в IT-индустрии. Если вы желаете провести серьезный анализ, используйте генератор парсеров вместо того, чтобы писать их вручную. Написание парсеров вручную – не более чем забавное упражнение, а также способ изучить то, как парсер логически структурирует обработку текста.

## Практическое задание

Чтобы изучить язык `bc`, вы должны изучить исходный код для него на **ftp.gnu.org/gnu/bc/** и найти файлы `bc.y`, `sbc.y` и `scan.l`. Это может сбивать с толку, так что исследуйте инструмент под названием `lex`, и еще один под названием `yacc`.

УПРАЖНЕНИЕ 48

# Команда ed

Если у вас получается следовать движению, значит, вы должны быть в состоянии сосредоточиться на одном большом проекте в течение нескольких недель подряд. В этом проекте вашей задачей будет создание как можно более точной копии команды ed. Цель этого упражнения состоит в том, чтобы вообще не использовать креативность, вместо этого методически создавая точную копию фрагмента программного обеспечения. Смотрите на это как на подлог. Вам нужно что-то настолько качественное, что, если бы им заменили настоящую команду ed, никто не заметил бы разницы.

Это упражнение будет заключаться в том, чтобы методом мастер-копии создать команду ed, настолько близкую к оригиналу, насколько это возможно. Это значит, что ваш тестовый набор должен проверять настоящую версию и созданную вами версию на одних и тех же сценариях, сравнивая вывод. Это похоже на упражнение с использованием мастер-копии, которое вы выполняли при изучении алгоритмов, за исключением того, что вы копируете поведение существующего программного обеспечения, а не пытаетесь запомнить его. Процесс аналогичен, но вы теперь можете использовать тестовые наборы, чтобы ускорить работу.

## Задача упражнения

Инструмент ed является одним из самых первых текстовых редакторов Unix, и, честно говоря, он никуда не годится. Я даже не могу себе представить, как кто-то вообще использовал ed для редактирования текста, ведь это пример совершенно недружелюбного к пользователю программного обеспечения. Если вы не в курсе, каким мучением была работа с компьютерами в старые недобрые времена Unix, то «подделывание» ed даст вам необходимое представление.

Вам нужно знать кое-что об ed: несмотря на то, что редактор поддерживает сценарии, он изначально использовался в интерактивном режиме. Это было похоже на игру MUD[9] (или МПМ – сокращение от «многопользовательский мир»), только для текстовых файлов. Сначала вы запускали ed в командном режиме, это позволяло вам изменять текст с помощью команд. Когда выполнялась команда, требующая ввода, ed переходил в режим ввода и оставался

[9] Текстовая многопользовательская компьютерная игра. – *Прим. ред.*

в нем до тех пор, пока работа команды не заканчивалась. Для редактирования вы также должны были знать адреса строк. Это может показаться крайне неудобным, но, по сравнению с другими текстовыми редакторами того времени, это было чудом прогресса.

При создании копии `ed` вам придется в значительной степени положиться на библиотеку `re` в Python (**docs.python.org/2/library/re.html**), которая облегчит работу с регулярными выражениями. Мы использовали эту библиотеку в упражнении 31, поэтому вы должны быть знакомы с ней и с регулярными выражениями в целом.

Я также предлагаю вам попытаться использовать `ed` в течение одного 45-минутного сеанса, чтобы написать немного кода для проекта `ed`. Тяготы этого процесса помогут вам в дальнейшем создании копии.

Помимо этого, вам нужно прочесть man-страницу `ed` для изучения основ, а также посмотреть уроки по использованию этой команды. Верным начальным шагом было бы отыскать в интернете различные сценарии и попытаться выполнить их в качестве первого тестового примера.

---

**Внимание!** Подсказка: вам понадобится конечный автомат, чтобы справиться с модальной природой команды `ed`.

---

# Практические задания

1. Найдите исходный код GNU `ed` и просмотрите его, даже если вы не знаете C.

2. Сделайте из своей реализации `ed` модуль, который затем можно использовать в других проектах. Он понадобится вам для последующих упражнений.

3. Никогда больше не создавайте подобное программное обеспечение – если только вы не скучаете.

УПРАЖНЕНИЕ 49

# Команда `sed`

В упражнении 9 вы реализовали «детскую» версию команды `sed`, когда учились выполнять задания быстро и грязно. В этом упражнении вы попытаетесь создать еще одну, на этот раз точную копию команды. В разделе «Практические задания» упражнения 48 вам предлагалось создать модуль из вашей реализации. Если вы этого не сделали, вернитесь и создайте модуль – он понадобится для вашей команды `sed`, она должна будет его использовать.

Как обстоят дела с вашим движением? Помогает ли оно в работе над этими долгими проектами? Есть ли что-либо что, по вашему мнению, нужно изменить? Вы еще записываете свои показатели или вам кажется, что этот метод для вас уже в прошлом? Прежде чем начинать данное упражнение, потратьте время на то, чтобы просмотреть свой журнал и узнать, насколько вы стали лучше с того момента, как приступили к чтению книги.

Задача этого упражнения – взять код из проекта по `ed` в упражнении 48 и повторно использовать его в данном проекте. Концепция «повторное использование» имеет решающее значение для работы программного обеспечения, но во многих случаях планирование повторного использования приводит к катастрофическим последствиям. Слишком часто люди разрабатывают программное обеспечение таким образом, чтобы каждый компонент можно было использовать в другой программе, но из-за такого подхода они чрезмерно усложняют проектирование, не имея реального плана по использованию наработок в другом проекте. Лучше создавать дискретное программное обеспечение, способное работать самостоятельно, а затем, начиная другой проект, извлекать фрагменты, которые можно будет в нем использовать.

Обычно я пишу программное обеспечение, совершенно не заботясь о повторном использовании. Мне все равно, будут ли части моего проекта использоваться в других проектах. Меня волнует лишь то, что эта часть программного обеспечения работает хорошо и имеет высокое качество. Начиная новый проект, я просматриваю остальные свои работы, чтобы узнать, есть ли там фрагменты, которые можно использовать снова. Если есть, я извлекаю из старого проекта необходимые мне компоненты. Таким образом, мой алгоритм повторного использования выглядит следующим образом.

1. Реализовать полностью работоспособное высококачественное программное обеспечение с автоматизированными тестами. Не

заботиться о том, чтобы какая-то его часть была пригодной для использования в другом программном обеспечении.

2. Начать новый проект, который может использовать код из первого проекта.

3. Вернуться к первому проекту и оформить его код в отдельном модуле, который может быть использован этим первым проектом, и больше вообще ничего не менять.

4. Если все исходные автоматизированные тесты показывают требуемые результаты с модулем на месте первого проекта, использовать этот модуль в новом проекте.

5. Попытка использования модуля в новом проекте позволит понять, какие в этот модуль нужно внести изменения. После внесения изменений удостовериться, что они также работают с исходным программным обеспечением.

Вы не сможете осуществить это без автоматизированных тестов, поэтому, если в вашем проекте по `ed` нет тестов, я не уверен, что вы вообще читали эту книгу. Вернитесь и убедитесь, что ваши тесты полностью покрывают код проекта `ed`.

## Задача упражнения

Сначала вам нужно извлечь фрагменты проекта `ed`, которые обрабатывают команды, и поместить их в модуль, используемый `ed`, не нарушив работу тестов. Честно говоря, это будет одна из самых сложных частей данного проекта, поскольку `sed` в основном использует то же самое, но без модального характера интерактивного интерфейса `ed`.

Затем вам нужно либо взять свой старый код из упражнения 9 и сдуть с него пыль, либо начать этот новый проект с нуля. После того как вы примете соответствующее решение, реализуйте как можно больше функций `sed` с помощью модуля `ed`. Креативность в данном упражнении заключается в том, чтобы определить, что именно необходимо для обоих проектов, а затем поместить это в модуль.

Ваша цель при реализации состоит в том, чтобы сделать максимально точную копию команды `sed`. В этой части упражнения нет места креативности.

Просто старайтесь быть настолько педантичными, насколько возможно, а также используйте автоматическое тестирование, чтобы убедиться, что ваша и исходная команда `sed` работают одинаково.

Наконец, при работе над `sed` вы найдете в модуле необходимые вам компоненты. Вам нужно будет внести изменения в модуль, заставить их работать в `sed`, а затем вернуться к `ed` и заставить их работать там. Этот процесс метания между тремя проектами будет нелегким, поэтому я предлагаю вам придерживаться 45-минутного подхода, чтобы не выдохнуться из-за переключений контекста.

## Практическое задание

Когда вы работали над модулем, обнаружили ли вы у себя какие-либо привычки, которые затрудняли извлечение кода? Что это за привычки?

# Текстовый редактор vi

За это упражнение меня посадят в тюрьму. У вас есть модуль, который реализует возможности, используемые в ed и sed. Очевидно, что следующим шагом будет реализация наиболее ненавистного и полезного текстового редактора в истории человечества — vi. Если бы вы знали язык Лисп, вы могли бы реализовать Emacs, но ни у кого нет времени создавать целую новую операционную систему, притворяющуюся текстовым редактором. Жизнь слишком коротка, чтобы целый день удерживать три клавиши и жать tab.

Цель этого упражнения состоит не в том, чтобы создать идеально точную копию vi. Это огромный проект, но, если вам хочется попробовать, тогда желаю вам удачи. Ваша цель в этом проекте – еще раз повторно использовать ваш модуль ed и поиграть с библиотекой Python curses (**docs.python.org/2/howto/curses.html**). Модуль curses позволяет работать со старомодным окном текстового Терминала и осуществлять манипуляции с графикой. Вообще-то слово «графика» должно быть в кавычках, ведь в curses очень мало настоящей графики.

Вы будете использовать curses, чтобы создать легкую версию редактора vi, которая позволяет открывать файлы, запускать команды ed и sed при помощи вашего модуля и использовать curses для отображения их на экране Терминала. Вы также поймете, что создать для этого автоматизированный тест будет очень сложно. Вы заработаете дополнительные баллы, если разберетесь, как сделать фиктивный фреймворк curses для тестирования, но для этого потребуются навыки волшебства и псевдотерминал Unix (как мне кажется).

Лучший способ сделать реализацию пригодной для тестирования – поместить как можно большую часть вашего vi в модули Python, чтобы можно было тестировать код без необходимости запуска экранной системы curses. Когда я говорю «модули», я не имею в виду полностью готовую библиотеку Python, которую вы устанавливаете с помощью pip, как в случае с модулем ed. Я имею в виду модули, что находятся прямо в коде для vi, а затем импортируются в ваш проект.

При работе над этим проектом стоит разграничивать код, который контролирует представление (curses), от остальной части кода, чтобы для тестирования вы могли подключать собственное представление. Тогда останется

небольшая часть функциональности, которую можно протестировать, вручную запустив `vi`.

## Задача упражнения

Мы не будем реализовывать весь `vi`. Нужно, чтобы вы это хорошо поняли, поскольку настоящий `vi` – устаревший и крайне запутанный инструмент, поэтому создание его полноценной копии займет много времени. Вам необходимо сделать только следующее.

1. Взять свой модуль `ed`.
2. Создать для него пользовательский интерфейс `curses`.
3. Проверить работу на нескольких файлах.

В общем и целом, это все, что вам нужно сделать. Первое, на что вы должны обратить внимание, это то, как работает `curses`. Для этого прочтите соответствующую документацию и выполните как можно больше пробных заданий.

Когда у вас появится понимание `curses`, вам нужно будет научиться пользоваться `vi`. К этому упражнению я приложил видео с кратким курсом по `vi`. Кроме того, в интернете есть несколько шпаргалок, которые вы можете изучить. Предлагаю просмотреть мой урок по `vi` и во время этого сеанса попытаться использовать настоящий `vi` для редактирования кода. На самом деле ваша реализация `ed` и `sed` даст хорошее представление о том, как работает `vi`. Теоретически `vi` – это просто «графический `ed`», поэтому вы, грубо говоря, просто улучшаете интерфейс `ed`.

## Практические задания

1. Как конечный автомат из вашей реализации `ed` соотносился с тем, который вы использовали в этой реализации `vi` (если вы использовали этот подход к проектированию)?
2. Насколько сложно было бы сделать версию с графическим интерфейсом пользователя вместо `curses`? Я не советую вам этого делать, но изучите задачу и прикиньте, что для этого потребовалось бы.

# Создание веб-сервера (lessweb)

Мы приближаемся к концу книги, так что я дам вам один проект на последние два упражнения: вы создадите веб-сервер. В этом упражнении вы просто узнаете о библиотеке Python `http.server` и о том, как с ее помощью создать простой веб-сервер. Я дам вам инструкции и ожидаю, что вы прочтете документацию, чтобы выяснить, как это сделать. Здесь не будет слишком много указаний, поскольку к настоящему моменту вы должны уметь работать самостоятельно.

После того как вы создадите свой веб-сервер, вы напишете набор тестов, чтобы попытаться взломать его. Я расскажу об этом в разделе «Ломаем это», но сейчас вам должно быть легко находить недостатки в своем коде.

## Задача упражнения

Для начала вам нужно будет прочитать документацию по `http.server` (**docs.python.org/3/library/http.server.html**). Кроме того, следует ознакомиться с документацией по `http.client` (**docs.python.org/3/library/http.client.html**), а также документацией по библиотеке `requests` (**docs.python-requests.org/en/master/**). Чтобы писать тесты для создаваемого вами `http.server`, вы будете использовать либо `requests`, либо `http.client`.

Далее ваша задача – при помощи `http.server` создать веб-сервер, который может осуществлять следующее.

1. Производить настройку из файла конфигурации.
2. Непрерывно работать и обрабатывать получаемые запросы.
3. Предоставлять файлы из сконфигурированных каталогов.
4. Отвечать на запросы по сайтам и выдавать правильный контент.

5. Записывать все поступающие запросы в файл, который затем можно будет прочесть.

Если вы ознакомитесь с примером в документации, у вас, вероятно, все это будет работать на базовом уровне. Часть данного упражнения заключается в том, как взломать безыскусный веб-сервер, так что вам нужно просто сделать так, чтобы он кое-как функционировал, и затем мы перейдем к следующей части.

## Ломаем это

В этом разделе ваше задание – атаковать свой веб-сервер любым доступным способом. Можете начать со списка 10 главных уязвимостей OWASP, открытого проекта обеспечения безопасности веб-приложений, по адресу **www.owasp.org/index.php/Category:OWASP_Top_Ten_Project**, а затем перейти к другим распространенным способам атак. Вам также нужно будет прочитать документацию модуля `os` **(docs.python.org/3/library/os.html)** для внесения некоторых исправлений. Вот дополнительный список ошибок, которые вы наверняка совершите.

1. Нежелательный обход каталога. Вероятно, вы используете базовый путь из URL (`/some/file/index.html`) и просто открываете его по запросу. Возможно, вы добавляете полный путь к файлу в ОС (`/Users/zed/web/some/file/index.html`) и считаете, что все в порядке. Попробуйте получить доступ к файлу за пределами этого каталога, используя спецификаторы пути `..`. Если вы можете запросить `/../../../../../../../../etc/passwd`, значит, все верно. Попытайтесь объяснить, почему так происходит и что вы можете сделать, чтобы исправить это.

2. Отсутствие обработки нежелательных запросов. Вы, скорее всего, ожидаете методы `GET` и `POST`, но что произойдет если кто-то запросит `HEAD` или `OPTIONS`?

3. Отправка гигантского HTTP-заголовка. Посмотрите, можете ли вы «сломать» или замедлить Python `http.server`, отправив ему чересчур большой заголовок запроса.

4. Отсутствие выдачи ошибки при запросе неизвестного домена. Кое-кто рассматривает как элемент функциональности (кхе-кхе, сервер Nginx) то, что когда сервер не распознает домен, то выдает

«случайный» сайт. На вашем сервере должен быть включен белый список, и, если сервер не распознает домен, он должен выдать ошибку 404.

Это всего лишь несколько мелких ошибок, совершаемых людьми. Изучите как можно больше других ошибок, а затем напишите автоматизированные тесты для своего сервера, чтобы наглядно их продемонстрировать, прежде чем исправлять. Если в вашем сервере нет ни одной из этих ошибок, внесите их нарочно. Узнать, что происходит при этих ошибках, тоже полезно.

## Практические задания

1. Прочтите о функции `os.chroot` в документации к библиотеке `os`.

2. Узнайте, как использовать эту функцию и другие функции модуля `os` для создания «chroot-тюрьмы» (chroot jail).

3. Используя как можно больше функций в `os` и любых доступных вам модулях, перепишите свой сервер так, чтобы он использовал механизм chroot-тюрьмы и отдавал привилегии безопасному пользователю (не суперпользователю). На `Windows` это может быть очень сложно, так что попробуйте сделать это на компьютере под Linux, либо просто пропустите это задание.

# Создание веб-сервера (moreweb)

В прошлом упражнении вы создали веб-сервер при помощи библиотеки Python `http.server`. Теперь вы можете переходить к последнему проекту. Вы создадите собственный веб-сервер с нуля, используя все, чему научились до сих пор. В упражнении 51 вы создали большую часть обработки, которая «выше» модуля `http.server`. Вы не занимались обработкой сетевых подключений или парсингом протокола HTTP. В этом финальном упражнении вы реализуете все необходимое для воссоздания того, что делает `http.server` для вашего сервера lessweb.

## Задача упражнения

Чтобы выполнить это упражнение, вам нужно будет ознакомиться с модулем Python 3 `asyncio` (**docs.python.org/3/library/asyncio.html**). Эта библиотека предоставляет инструменты для обработки запросов к сокету, создания серверов, ожидания сигналов и почти всего, что вам будет нужно. Если вы желаете повысить сложность этого упражнения, можете использовать модуль `select` (**docs.python.org/3/library/select.html**), который позволяет обрабатывать сокеты на еще более низком уровне. Вы должны использовать эту документацию для создания серии небольших серверов сокетов и клиентов.

Как только вы поймете, как создавать серверы и клиенты, которые общаются с помощью сокета TCP/IP, вам нужно перейти к обработке запросов HTTP. Эта часть проекта будет устрашающей, поскольку стандарт HTTP просто безумен и гораздо более сложен, чем следует. Я бы начал с самой простой библиотеки для парсинга HTTP, которую вы только можете создать, а затем дополнил бы ее все большим и большим количеством примеров. Первое, с чего следует начать, это документ RFC 7230 (**tools.ietf.org/html/rfc7230**), но будьте готовы столкнуться с одним из худших примеров использования английского языка.

Лучшим способ изучить RFC 7230 заключается в том, чтобы сначала сосредоточить внимание на грамматике в приложении по РФБН (Collected ABNF) (**tools.ietf.org/html/rfc7230#appendix-B**). На первый взгляд это кажется безумием: ведь это просто огромная грамматическая спецификация. На самом деле в части V этой книги вы учились читать такие спецификации, только в

меньших масштабах. Вы знаете, как работают регулярные выражения, лексические и синтаксические анализаторы и как читать подобную грамматику. Все, что вам нужно сделать, это изучить данную грамматику и реализовывать ее понемногу за раз. При реализации я бы полностью игнорировал любую «порционную» (chunk) грамматику.

Изучив эту грамматику, вы должны начать писать парсер для HTTP, используя свои наработки. Используйте свои структуры данных, инструменты синтаксического анализа и все, что можете, чтобы создать правильный парсер для небольшого подмножества HTTP. Постарайтесь охватить как можно больше этой грамматики. Чтобы вам было проще, вот набор тестовых файлов, в которых есть допустимые HTTP-запросы: **learncodethehardway.org/more-python-book/http_tests.zip**. Вы можете загрузить этот набор тестовых примеров и пропустить их через свой синтаксический анализатор, чтобы убедиться в его корректной работе. Я извлек многие из этих тестовых примеров с превосходного сервера And-HTTP (**www.and.org/andhttpd/**), а затем дополнил их более простыми примерами. Ваша цель – сделать так, чтобы как можно больше из них были успешно пройдены.

Наконец, как только у вас есть способ написать достойный сервер сокетов при помощи `asyncio` или `select`, а также способ производить парсинг HTTP, вы можете объединить их и создать свой первый функционирующий веб-сервер.

## Ломаем это

Вы определенно должны попытаться «сломать» свой веб-сервер, но вы также должны попробовать кое-что иное. Вы написали парсер для HTTP, который пытается обработать допустимый HTTP наиболее логичным способом с использованием парсинга MPC. Скорее всего, ваш синтаксический анализатор блокирует множество ошибочных запросов HTTP, так что найдите какие-нибудь атаки, произведенные в прошлом, и опробуйте их на своем веб-сервере. Вы найдете несколько инструментов автоматизации взлома веб-сайтов – выберите один из них и направьте его на свой сервер. Однако соблюдайте безопасность, и убедитесь, что вы используете только надежные инструменты тестирования и только на собственном сервере.

# Дальнейшее обучение

Если вы хотите изучить веб-серверы и данную технологию, используйте свой сервер `moreweb` для создания веб-фреймворка. Я предложил бы сначала создать сайт, а затем извлечь из него шаблоны, необходимые для веб-фреймворка. Цель такого фреймворка – инкапсулировать используемые вами шаблоны, чтобы упростить веб-приложения, которые вы будете создавать в последующем. Как и в случае с упражнениями `lessweb` и `moreweb`, ваша цель должна заключаться в исследовании, реализации и применении распространенных атак на веб-фреймворки.

Если вы хотите с головой погрузиться в TCP/IP, я рекомендую книгу Йона Снейдера «Эффективное программирование *TCP/IP*»[10]. Данная книга использует C, но это эффективный «Легкий способ выучить TCP/IP». Она охватывает 44 темы с простым кодом, чтобы вы могли понять, как работает базовый стек TCP/IP. C – это то место, где родился TCP/IP, поэтому многое из того, как другие языки обрабатывают соединения сокетов, кажется странным, пока вы не узнаете, как это делает C. Изучив это, вы получите прочное понимание основ работы серверов. Единственное, стоит помнить о том, что книга немного устарела, так что код должен работать, но это будет не самый современный код.

[10] Книга выходила на русском языке в 2001 году в издательстве Питер. – *Прим. ред.*

# Предметный указатель

## T



## U



## V

## Г



## Д



## З



## И



## К

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

**Зед Шоу**

**ЛЕГКИЙ СПОСОБ ВЫУЧИТЬ PYTHON 3 ЕЩЕ ГЛУБЖЕ**

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Ответственный редактор *Е. Горанская*
Литературный редактор *С. Ульянов*
Младший редактор *Д. Атакишиева*
Художественный редактор *А. Шуклин*
Компьютерная верстка *А. Смирнов*
Корректор *Р. Болдинова*

**ООО «Издательство «Эксмо»**
123308, Москва, ул Зорге, д 1 Тел. 8 (495) 411-68-86
Home page www eksmo ru E-mail info@eksmo ru
Өндіруші. «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй
Тел 8 (495) 411-68-86.
Home page www eksmo ru E-mail info@eksmo ru
Тауар белгісі: «Эксмо»
**Интернет-магазин** www.book24.ru

**Интернет-магазин** www book24.kz
**Интернет-дукен** www book24 kz
Импортер в Республику Казахстан ТОО «РДЦ-Алматы»
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ , Домбровский көш , 3«а», литер Б, офис 1
Тел 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген
Сертификация туралы ақпарат сайтта www.eksmo ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»
www eksmo ru/certification
Өндірген мемлекет Ресей. Сертификация қарастырылмаған

Подписано в печать 16.01.2020. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 22,04.
Тираж 2000 экз. Заказ 4866.

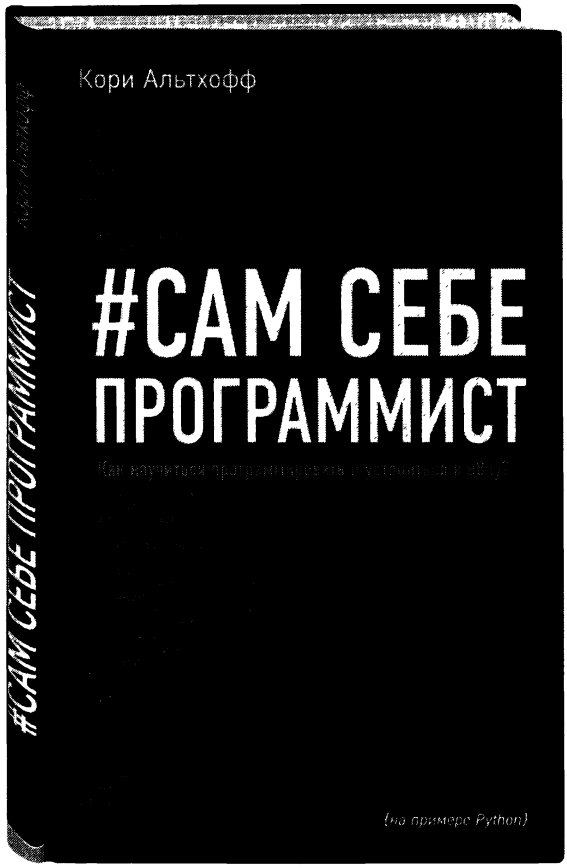
Отпечатано в ОАО «Можайский полиграфический комбинат»
143200, Россия, г. Можайск, ул. Мира, 93.
www oaompk ru, тел (495) 745-84-28, (49638) 20-685



ISBN 978-5-04-093107-1

# #Сам себе программист.

## Как научиться программировать и устроиться в Ebay?




**Как успешно пройти собеседование в любой IT компании и перестать сомневаться в собственных силах?**
**Уникальный авторский метод. С примерами на языке Python, самом востребованном и популярном языке программирования.**

2018-05

# Вы выучите Python 3 еще глубже!

## Хотите получить «черный пояс» по программированию? Зед Шоу гарантирует его вам!

Это вторая часть «Легкого способа выучить Python 3», где Зед описывал базовые принципы программирования на Python 3.

Вас ждут:

**52** идеально продуманных упражнения,

а также теория и ответы на часто задаваемые вопросы. Выполняйте задания, исправляйте ошибки, запускайте программы и наслаждайтесь результатом!

Вы узнаете:

- Как работает код
- Как выглядят хорошо написанные программы
- Как читать, писать и анализировать код
- Как находить и исправлять допущенные ошибки

Зед Шоу – программист, писатель, а еще он заядлый гитарист. Его книги прочли миллионы людей по всему миру. Написанные им программы используются в крупнейших международных компаниях. Его публикации все время цитируются многочисленными сообществами гиков в социальных сетях. Откройте для себя и вы этого интересного автора, чьи книги помогают людям исполнять свои мечты и обучаться программированию с нуля.

*«По скорости устаревания книги по программированию сравнятся разве что с кассетами для фильтров воды. Но это издание выдержало не один виток вокруг Солнца не в ущерб пользе содержания. Методика обучения, придуманная и доведенная до совершенства Зедом, по размеру фанбазы и безотказности действительно может посоперничать с наследием Аллена Карра, к которому отсылает название книги в русском варианте».*

***Кирилл Жвалов,***
***сооснователь «Moscow Coding School»***

ISBN 978-5-04-093107-1