Кори Альтхофф

//Основано на реальном опыте!

```
import sys

class CannotResolve(Exception):
    pass

class Resolver(object):
    emess = "Can't found appropriate signature of func %s() for call with" + \
            " params %r"
    def __init__(self,name):
        self.function map = {}
```

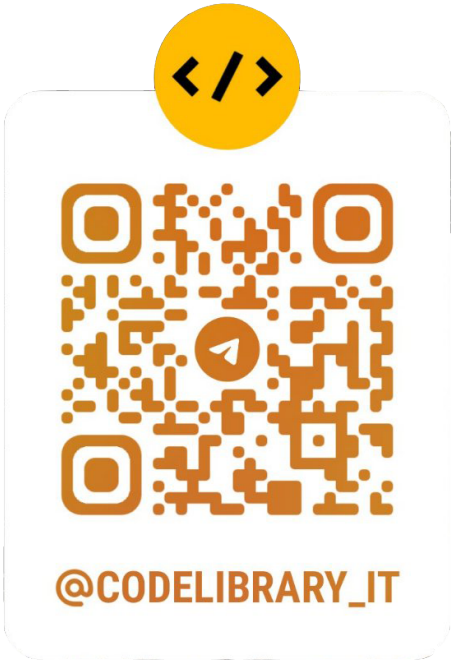
#САМ СЕБЕ ПРОГРАММИСТ

Как научиться программировать и устроиться в eBay?

```
        self.default = None
        self.name = name
    def __call__(self,*dt):

        cls = tuple(map(type,dt))

        try:
            x = self.function_map[cls]
        except KeyError:
            if self.default is not None:
                x = self.default
            else:
                raise CannotResolve(self.emess % (self.name,cls))
        return x(*dt)
def overload(*dt):

    def closure(func):
        name = func.__name__
        fr = sys._getframe(1).f_locals.get(name,Resolver(name))

        fr.function_map[dt] = func

        return fr
    return closure
def overdef(func):
    name = func.__name__
```

{на примере Python}

Мировой
компьютерный
бестселлер



Cory Althoff

#THE SELF-TAUGHT PROGRAMMER

Кори Альтхофф
//Основано на реальном опыте!

#САМ СЕБЕ ПРОГРАММИСТ

Как научиться программировать и устроиться в eBay?

{на примере Python}

БОМБОРА™
Москва 2018

УДК 004.42
ББК 32.973.26-018.1
А58

The Self-Taught Programmer: The Definitive Guide
to Programming Professionally
Cory Althoff

Альтхофф, Кори.

А58 Сам себе программист. Как научиться программировать и устроиться в Ebay? / Кори Альтхофф ; [пер. с англ. М.А. Райтмана]. — Москва : Эксмо, 2018. — 208 с. — (Мировой компьютерный бестселлер).

Автор книги всего за год научился программировать, что само по себе немало. Однако Кори Альтхофф пошел дальше, и, научившись программировать, он устроился разработчиком в одну из самых серьезных современных IT-компаний — Ebay. Как ему удалось? Читайте эту книгу, изучайте программирование на языке Python по уникальной авторской методике — вам это тоже по силам!

УДК 004.42
ББК 32.973.26-018.1

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Кори Альтхофф

САМ СЕБЕ ПРОГРАММИСТ

Как научиться программировать и устроиться в Ebay?

Директор редакции *Е. Капьёв.* Ответственный редактор *Е. Истомина*
Младший редактор *Е. Минина.* Художественный редактор *А. Шуклин*

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»
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) 2 51 59 89,90,91,92, факс: 8 (727) 251 58 12 вн. 107; 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 по адресу: http://eksmo.ru/certification/

Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 12.02.2018. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 16,85.
Тираж экз. Заказ

ISBN 978-5-04-090834-9
9 785040 908349 >
ISBN 978-5-04-090834-9

Содержание

Я посвящаю эту книгу своим родителям,
Эбби и Джеймсу Альтхоф, за то, что всегда поддерживали меня.

ЧАСТЬ I

Введение в программирование

Глава 1. Вступление

Большинство хороших программистов делают свою работу не потому, что ожидают вознаграждения или признания, а потому, что получают удовольствие от программирования.

Линус Торвальдс

Я изучал политологию в Университете Клемсона. Подумывая заняться информатикой, в первый учебный год я даже записался на курс «Введение в программирование», но быстро бросил его. Это было слишком сложно. Живя в Силиконовой долине после окончания университета, я решил, что нужно научиться программировать. Год спустя я работал в eBay на должности инженера-программиста второй категории (выше начального уровня, но ниже старшего инженера-программиста). Я не хочу, чтобы у вас создалось впечатление, будто это было легко. Это было невероятно трудно. А в промежутках между швырянием вещей в стену это было еще и очень весело.

Я начал свой путь постижения программирования с популярного языка Python. Впрочем, моя книга не направлена на обучение программированию на определенном языке. Основное внимание в ней уделяется тому, чему прочие ресурсы вас не научат. Речь идет о тех вещах, которыми мне пришлось овладеть самостоятельно. Эта книга задумывалась не для тех, кому нужно обычное введение в программирование, чтобы писать код на досуге, в качестве хобби. Она написана для тех, кто планирует заниматься программированием профессионально. Независимо от того, хотите ли вы стать инженером-программистом, предпринимателем или использовать полученные навыки в другой профессии, эта книга написана для вас.

Изучение языка программирования — лишь часть битвы; чтобы говорить на языке компьютерных ученых, вам понадобятся и другие навыки. Я научу вас всему, что постиг на пути от новичка до профессионального программиста. Я написал эту книгу, чтобы обрисовать начинающим кодерам общий контур того, что

им необходимо знать. Будучи программистом-самоучкой, я не знал, что именно мне нужно прочитать и узнать. Книги по введению в программирование все одинаковые. Они предлагают читателям лишь основы программирования на Python, либо на Ruby, а затем отправляют вас восвояси. Отклики, которые я слышал от людей, закончивших читать подобные книги, звучали так: «И что мне делать теперь? Я еще не программист, но не знаю, чему учиться дальше». Эта книга — мой ответ на подобный вопрос.

Как построена эта книга

Многим темам, охваченным всего в одной главе этой книги, посвящены целые отдельные книги. У меня не было цели детально раскрывать все темы, которые вам нужно знать. Моей целью было дать вам карту – набросок всех навыков, которыми нужно овладеть, чтобы начать программировать профессионально. Эта книга состоит из пяти частей.

Часть I. Введение в программирование. Вы напишете свою первую программу как можно скорее, надеюсь, прямо сегодня.

Часть II. Введение в объектно-ориентированное программирование. Я освещаю различные парадигмы программирования, фокусируясь на ООП. Вы создадите игру, которая продемонстрирует вам возможности программирования. После этого раздела вы «подсядете» на программирование.

Часть III. Введение в инструменты программирования. Вы научитесь использовать различные инструменты, чтобы поднять свою производительность на новый уровень. К этому моменту вы уже влюблены в программирование и хотите стать лучше. Вы узнаете больше о своей операционной системе, о том, как использовать регулярные выражения для повышения своей производительности, как устанавливать программы и пользоваться ими, а также как взаимодействовать с другими инженерами при помощи управления версиями.

Часть IV. Введение в информатику. Эта часть представляет собой краткое введение в информатику. Я касаюсь здесь двух важных тем: алгоритмов и структур данных.

Часть V. Получение работы. В заключительном разделе речь пойдет о лучших методах программирования, получении места инженера-программиста, командном взаимодействии и профессиональном совершенствовании. Я даю советы о том, как пройти собеседование, работать в команде, а также насчет того, как усовершенствовать свои навыки в будущем.

Если у вас нет опыта программирования, перед каждым разделом вы должны самостоятельно практиковаться в этом, и как можно больше. Не пытайтесь читать мою книгу слишком быстро, вместо этого используйте ее как руководство, практикуясь столько, сколько нужно в перерывах между разделами.

Сначала эндшпиль

Я учился программировать способом, полностью противоположным тому, который обычно практикуется при изучении информатики, и я структурировал книгу, следуя этому подходу. Традиционно ученики проводят много времени, изучая теорию; так много, что многие выпускники школ компьютерных наук

не знают, как программировать. В своем блоге «Почему программисты не умеют... программировать?» Джефф Этвуд пишет: «Как и меня, автора тревожит тот факт, что 199 из 200 соискателей программистских вакансий вообще не умеют писать код. Повторяю: они вообще не способны написать какой-либо код». Это подтолкнуло Этвуда к созданию задачи по кодингу **FizzBuzz** — теста на знание программирования, используемого на собеседованиях для отсеивания кандидатов. Большинство людей проваливают этот тест, вот почему в моей книге читатель тратит столько времени на получение навыков, которые пригодятся ему на практике. Не волнуйтесь, вы научитесь проходить тест FizzBuzz.

В книге «Искусство учиться» Джош Вайцкин, прославленный в фильме «В поисках Бобби Фишера», описывает, как он учился играть в шахматы в обратном порядке. Вместо того чтобы постигать первые ходы, он начал с изучения эндшпиля (когда на доске остается лишь несколько фигур). Эта стратегия позволила ему лучше понять игру, и позже он выиграл множество чемпионатов. Точно так же и я полагаю, что было бы более эффективно в первую очередь научиться программировать, а затем, когда вы уже будете до смерти хотеть знать, как там все устроено, приниматься за теорию. Вот почему я не затрагиваю теорию информатики вплоть до четвертой части книги, да и там свожу ее к минимуму. Хотя теория важна, она станет еще ценнее, когда у вас появится опыт программирования.

Вы не одиноки

Изучение программирования за пределами учебного заведения становится все более распространенным явлением. Опрос, проведенный в 2015 году Stack Overflow (онлайн-сообществом программистов), показал, что 48 % респондентов не имеют ученой степени в области компьютерных наук[1].

Преимущество самообучения

Когда меня приняли на работу в компанию eBay, я попал в одну команду с программистами из Стэнфорда, Университета Калифорнии и Дьюка, у которых были степени по информатике, а также с двумя докторами наук. В 25 лет было страшно осознавать, что мои 21-летние товарищи по команде знают о программировании и об информатике в 10 раз больше меня.

Как бы ни было страшно работать с людьми, у которых есть степень бакалавра, магистра или доктора по информатике, никогда не забывайте, что у вас есть, как я люблю это называть, «преимущество самообучения». Вы читаете эту книгу не потому, что вам так сказал преподаватель, вы читаете ее потому, что у вас есть стремление учиться, и это стремление — наибольшее преимущество из тех, что у вас есть. Кроме того, не забывайте, что некоторые наиболее успешные люди обучались программированию самостоятельно. Стив Возняк, основатель Apple, — программист-самоучка. Как и Маргарет Гамильтон, получившая Президентскую медаль Свободы за работу над программой «Аполлон» НАСА, Дэвид Карп, основатель Tumblr, Джек Дорси, основатель Твиттера, и Кевин Систром, основатель Instagram.

[1] www.infoworld.com/article/2908474/application-development/stack-overflow-survey-finds-nearly-half-have-no-degree-in-computer-science.html

Почему вы должны программировать

Программирование поможет вашей карьере независимо от выбранной профессии. Обучение программированию расширяет ваши возможности. Мне нравится придумывать новые идеи, и у меня всегда есть проекты, над которыми я хочу работать. Как только я научился программировать, я смог сесть и реализовать свои идеи, не утруждаясь поиском кого-то, кто сделал бы это за меня.

Программирование поможет вам стать лучше во всем, что вы делаете. Шутки в сторону. Существует не так много областей, в которых не пригодились бы идеально отработанные навыки решения задач. К примеру, недавно мне пришлось заниматься утомительным вопросом поиска жилья на Крейгслисте[2], и я смог написать программу, которая сделала эту работу за меня и отослала мне результаты по электронной почте. Знание программирования навсегда избавит вас от решения рутинных задач.

Если вы действительно хотите стать инженером-программистом, знайте, что спрос сейчас растет, а квалифицированных специалистов недостаточно для заполнения доступных вакансий. К 2020 году будет примерно миллион рабочих мест, связанных с программированием[3]. Даже если ваша цель не состоит в том, чтобы становиться программистом, работодатели в таких областях, как наука и финансы, тоже начинают отдавать предпочтение кандидатам с опытом в программировании.

Продолжайте этим заниматься

Если у вас нет опыта в программировании и вы чувствуете себя неуверенно, я хочу, чтобы вы знали: вам это по силам. Относительно программистов существует распространенное заблуждение, будто все они по-настоящему хороши в математике. Это не так. Чтобы научиться программировать, не нужно быть математическим гением, но все же это тяжелая работа. К счастью, большую часть материала, описанного в этой книге, легче освоить, чем вы думаете.

Чтобы улучшить свои навыки программирования, вы должны практиковаться каждый день. Единственное, что не даст вам двигаться дальше, это нежелание придерживаться учебного процесса, так что давайте рассмотрим пару способов, которые помогут вам продолжить работу.

Когда я только начинал, я составлял расписание и следил за тем, чтобы практиковаться каждый день — это помогало мне оставаться сосредоточенным.

Если вам нужна дополнительная помощь, Тим Феррисс, эксперт по производительности труда, рекомендует следующую технику поддержки мотивации: отдавайте деньги другу и члену семьи, указывая, что они должны вернуть вам их после достижения определенной цели в установленные вами сроки, а в противном случае пожертвовать организации, которая вам не нравится.

Оформление этой книги

Главы этой книги опираются друг на друга. Имейте в виду, что я стараюсь избегать повторного объяснения понятий. Когда я в первый раз ввожу новые терми-

[2] Крейгслист (Craigslist) — популярный в США сайт электронных объявлений. – *Прим. ред.*

[3] www.wsj.com/articles/computer-programming-is-a-trade-lets-act-like-it-1407109947?mod=e2fb

ны, они выделяются жирным шрифтом. Каждое такое слово определено в конце каждой главы в словаре терминов. Также в конце глав есть задания для развития навыков программирования вместе со ссылками на их решения.

Технологии, используемые в этой книге

В этой книге представлены технологии, позволяющие как можно больше погрузиться в практику программирования. Я стараюсь быть технологическим «агностиком», сосредоточиваясь на концепциях вместо технологий.

В некоторых случаях мне пришлось выбирать между множеством различных технологий. В главе 19 «Управление версиями» (для тех читателей, которые не знакомы с управлением версиями, я все объясню позже) я описываю основы использования системы Git. Я выбрал ее, поскольку считаю отраслевым стандартом для управления версиями. Я использую Python для большинства примеров в программировании, потому что это широко изучаемый язык и его просто читать, даже если вы им никогда не пользовались. Кроме того, практически в каждой отрасли на разработчиков Python наблюдается огромный спрос.

Чтобы следовать примерам в этой книге, вам понадобится компьютер. У компьютера есть **операционная система**, программа-посредник между физическими компонентами компьютера и вами. То, что вы видите, когда смотрите на экран, называется **графическим интерфейсом пользователя** или ГИП, и это часть вашей операционной системы.

Для настольных компьютеров и ноутбуков есть три распространенных операционных системы: **Windows**, **Unix** и **Linux**. Windows — операционная система Microsoft. Unix была создана в 1970-х годах. Современная операционная система корпорации Apple основана на Unix. С этого момента, когда я использую слово «Unix», я имею в виду операционную систему корпорации Apple. Linux является операционной системой с **открытым исходным кодом**, она используется большинством мировых **серверов**. Сервер — это компьютер или компьютерная программа, выполняющая задачи вроде хостинга (расположения) веб-сайта. Открытый исходный код подразумевает, что программное обеспечение не принадлежит компании или отдельному лицу и его можно изменить или заново распространить. Linux и Unix принадлежат к **Unix-подобным операционным системам**, из чего следует, что они очень похожи. Эта книга предполагает, что вы используете компьютер под управлением операционной системы Windows, Unix или Ubuntu (популярной версии Linux).

Скачивание файлов примеров

Скачать все файлы примеров из этой книги вы можете по ссылке **eksmo.ru/files/self_taught_programmer_althoff.zip**. Распакуйте скачанный файл с помощью программы-архиватора, например WinZip или WinRAR, и вы увидите структуру папок, названных согласно номерам глав в этой книге. В папке Глава 02 вы найдете файлы примеров из главы 2 и т.д.

В тексте книги рядом с листингами кода указаны имена файлов. Так, пример программы «Привет, мир!» из главы 2 вы найдете в файле *Python_ex1.py* в папке *Глава 02*.

Словарь терминов

FizzBuzz: тест на программирование, используемый на собеседованиях для отсеивания кандидатов.
Linux: операционная система с открытым исходным кодом, используемая на большинстве серверов по всему миру.
Unix: операционная система, созданная в 1970-х годах. Операционная система macOS корпорации Apple основана на Unix.
Unix-подобные операционные системы: Unix и Linux.
Windows: операционная система корпорации Microsoft.
Графический интерфейс пользователя (ГИП): часть операционной системы, которую вы видите, когда смотрите на экран компьютера.
Операционная система: программа, которая является посредником между физическими компонентами компьютера и вами.
Открытый исходный код: характеристика программного обеспечения, которое не принадлежит компании или отдельному лицу, но вместо этого поддерживается группой добровольцев.
Сервер: компьютер или компьютерная программа, которая выполняет такие задачи, как хостинг веб-сайта.

Практикум

Составьте расписание на день, которое предусматривает практику программирования.

Глава 2. Начало работы

> Хороший программист — это тот, кто смотрит в обе стороны, переходя дорогу с односторонним движением.
>
> *Даг Линдер*

Что такое программирование

Программирование — это создание инструкций, которые выполняет компьютер. Инструкции могут указывать компьютеру вывести строку `Привет, мир!`, найти данные в Интернете или считать содержимое файла и сохранить его в базе данных. Эти инструкции называются кодом. Программисты пишут код на разных языках программирования. Раньше программирование было намного сложнее, поскольку программисты были вынуждены использовать крайне сложные **низкоуровневые языки программирования**, такие как **язык ассемблера**. Когда язык программирования является низкоуровневым, это значит, что он ближе к двоичной записи (в нулях и единицах), чем **высокоуровневый язык программирования** (язык, который больше напоминает английский), и поэтому его сложнее понять. Ниже приведен пример простой программы, написанной на ассемблере:

bash_ex00.sh

```
global  _start
        section .text
```

```
_start:
        mov     rax, 1
        mov     rdi, 1
         mov     rsi, message
        mov     rdx, 13
        syscall
        ; exit(0)
        mov     eax, 60
        xor     rdi, rdi
        syscall
message:
       db      "Привет, мир!", 10
```

Ниже показана та же программа, написанная на современном языке программирования:

Python_ex001.py

```
1 ▌ print("Привет, мир!")
```

Как видите, сегодня программистам приходится гораздо проще. Вам не нужно тратить время на изучение сложных языков низкоуровневого программирования. Вместо этого вы научитесь использовать легко читаемый язык программирования Python.

Что такое Python

Python — это язык программирования с открытым исходным кодом, созданный голландским программистом Гвидо ван Россумом и названный в честь британской труппы комиков «Монти Пайтон» (Monty Python). Одним из ключевых соображений ван Россума было то, что программисты тратят больше времени на чтение кода, чем на его написание, поэтому он решил создать легко читаемый язык. Python является одним из самых популярных и простых в освоении языков программирования в мире. Он работает на всех основных операционных системах и компьютерах и применяется везде, где только можно — от создания веб-серверов до настольных приложений. Благодаря популярности этого языка, на программистов Python сегодня существует большой спрос.

Установка Python

Чтобы следовать примерам в этой книге, вам необходимо установить Python 3. Вы можете загрузить Python для Windows и Unix по адресу **python.org/downloads**. Если у вас Ubuntu, Python 3 уже установлен по умолчанию. Убедитесь, что вы загружаете версию Python 3, а не Python 2. Некоторые примеры из этой книги не будут работать в версии Python 2.

Python доступен как для 32-разрядных, так и для 64-разрядных компьютеров. Если компьютер был приобретен после 2007 года, то он, скорее всего, имеет 64-битную разрядность. Если вы не уверены, поиск в Интернете поможет вам разобраться.

Если у вас операционная система Windows или macOS, загрузите 64- или 32-разрядную версию Python, откройте файл и следуйте инструкциям. Вы также можете посетить сайт **theselftaughtprogrammer.io/installpython** и просмотреть видео, объясняющие, как установить Python в вашей операционной системе.

Исправление проблем

Начиная с этого момента, у вас должен быть установлен Python. При возникновении проблем с установкой, перейдите к главе 11 в раздел «Получение помощи».

Интерактивная оболочка

Python поставляется с программой IDLE (сокращение от interactive development environment — интерактивная среда разработки). Кроме того, это фамилия Эрика Айдла (Eric Idle), одного из членов «Летающего цирка Монти Пайтона». IDLE — это то, где вы будете вводить свой код на Python. После загрузки Python, выполните поиск IDLE в Проводнике (Windows), Finder (macOS) или Nautilus (Ubuntu). Советую создать ярлык на Рабочем столе, чтобы упростить поиск.

Щелкните мышью по ярлыку IDLE и откроется программа со следующими строками (впрочем, к моменту чтения книги все могло поменяться, так что не беспокойтесь, если сообщение отсутствует или отличается):

```
Python 3.6.3 (v3.6.3:2c5fed8, Oct  3 2017, 17:26:49) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

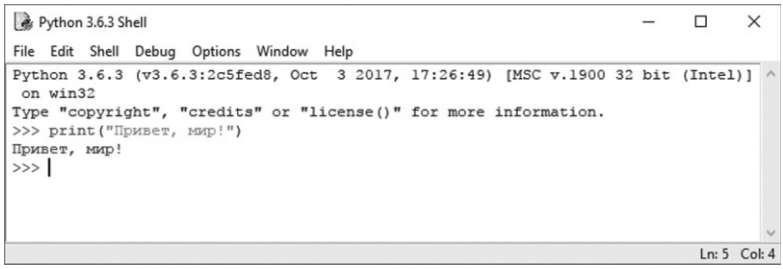
Эта программа называется интерактивной оболочкой. Вы можете вводить код Python непосредственно в интерактивную оболочку, и она выведет результаты. После приглашения >>> введите:

```
1 print("Привет, мир!")
```

Затем нажмите клавишу **Enter**.

IDLE может отвергать код, копируемый из Kindle, других электронных книг или текстовых редакторов, таких как Microsoft Word. Если вы скопировали и вставили код, но получили сообщение об ошибке, попробуйте набрать код вручную непосредственно в оболочке. Необходимо вводить код точно так, как он написан в примере, включая кавычки, круглые скобки и любые другие знаки препинания.

Интерактивная оболочка ответит выводом строки `Привет, мир!`



В мире программирования, когда вы обучаете кого-то новому языку, существует традиция в качестве самой первой программы научить его выводить строку `Привет, мир!` Так что поздравляю! Вы только что написали свою первую программу.

Сохранение программ

Интерактивная оболочка хорошо подходит для произведения быстрых вычислений, тестирования небольших фрагментов кода и написания коротких программ, которые вы больше не планируете использовать. В IDLE вы также можете сохранить программу для повторного использования. Запустите приложение IDLE, щелкните мышью по пункту **File** (Файл) в строке меню в левом верхнем углу редактора IDLE, а затем выберите команду **New File** (Новый файл). Так вы откроете текстовый редактор с пустым белым фоном. Вы можете писать свой код в этом текстовом редакторе и сохранять его, чтобы запустить позже. Когда вы запустите свой код, вывод появится в окне интерактивной оболочки. Нужно сохранить изменения при редактировании кода, прежде чем запускать его снова. Введите код программы «Привет, мир!» в текстовый редактор.

Снова щелкните мышью по пункту **File** (Файл) в строке меню и выберите команду **Save As** (Сохранить как). Присвойте файлу имя **hello_world.py** и сохраните его. Имена файлов Python должны заканчиваться расширением *.py*. Как только вы сохранили свой файл, щелкните мышью по пункту **Run** (Запустить) в строке меню редактора IDLE и выберите команду **Run Module** (Запустить модуль). Или можете просто нажать клавишу **F5**, что эквивалентно выбору команды **Run Module** (Запустить модуль). Строка `Привет, мир!` будет выведена в интерактивной оболочке, как если бы вы набрали эту строку кода в оболочке. Но теперь, поскольку вы сохранили свою программу, вы можете запускать ее столько раз, сколько хотите.

Созданная вами программа — это просто файл с расширением *.py*, расположенный на вашем компьютере там, где вы его сохранили. Имя, которое я выбрал для файла, *hello_world*, совершенно произвольно, вы можете называть файлы как угодно. Как и в этом примере, написание программ на Python заключается во вводе текста в файлах и запуске их с помощью интерактивной оболочки. Легко, правда?

Запуск программ-примеров

По ходу книги я буду приводить примеры кода и результатов, выводимых при его запуске. Всякий раз, когда я это делаю, вы должны ввести код и запустить его самостоятельно.

Короткие примеры удобнее всего реализовывать с помощью оболочки, а текстовый редактор лучше подходит для более длинных программ, которые нужно

сохранять и редактировать. Если вы допустили ошибку в своем коде в интерактивной оболочке, — например, опечатались — и он не работает, вам нужно вводить все заново. Использование текстового редактора позволяет сохранять вашу работу, поэтому, если вы допустили ошибку, просто исправляйте ее и запускайте программу повторно.

Еще один важный момент — вывод в программе, выполняемой из файла и из оболочки, может отличаться. Если вы введете `100` в интерактивную оболочку и нажмете клавишу **Enter**, интерактивная оболочка выведет `100`. Если вы введете `100` в файл с расширением .py и запустите его, вывода не будет вообще. Это различие может вызвать путаницу, поэтому, если вдруг вы не получите тот же результат, что в примере, проверьте, откуда вы запускаете программу.

Словарь терминов

Python: простой в чтении язык программирования с открытым исходным кодом, который вы научитесь использовать в этой книге. Создан Гвидо ван Россумом и назван в честь британской комедийной труппы «Монти Пайтон».
Высокоуровневый язык программирования: язык программирования, который больше похож на английский, чем язык программирования низкого уровня.
Код: инструкции компьютеру, которые пишут программисты.
Низкоуровневый язык программирования: язык программирования, запись которого ближе к двоичному формату (0 и 1), чем записи языка программирования высокого уровня.
Программирование: написание инструкций, которые выполняет компьютер.
Язык ассемблера: тип трудного для чтения языка программирования.

Практикум

Попробуйте вывести что-то отличное от `Привет, мир!`.

Решение: *chap2_challenge1.py*.

Глава 3. Введение в программирование

> Это единственная работа, где я могу быть одновременно инженером и художником. В ней есть что-то невероятное, технически строгое — что я очень люблю, поскольку оно требует крайне точного мышления. С другой стороны, здесь также присутствует простор для творчества, где единственным настоящим ограничением являются границы воображения.
>
> *Энди Херцфельд*

Наша первая программа выводила строку `Привет, мир!`. Давайте осуществим вывод сто раз. Введите следующий код в интерактивную оболочку (отступ перед командой `print` должен составлять ровно четыре пробела):

Python_ex2.py

```
1 for i in range(100):
2     print("Привет, мир!")
```

Ваша оболочка должна вывести строку `Привет, мир!` сто раз. Даже несмотря на то, что вам, вероятно, никогда не понадобится сто раз выводить `Привет, мир!`, этот пример наглядно демонстрирует, насколько мощно программирование. Можете придумать еще что-то, что вы так же легко сделаете сотню раз? Я не могу. В этом и есть сила программирования.

Примеры

Отныне примеры с кодом будут выглядеть так:

Python_ex2.py

```
1 for i in range(100):
2     print("Привет, мир!")
```

```
>> Привет, мир!
>> Привет, мир!
>> Привет, мир!
...
```

Текст *Python_ex2.py* обозначает имя файла в папке с примерами для данной главы (3), в котором сохранен соответствующий код. Поэтому, если у вас возникнут проблемы с выполнением кода, вы можете просто скопировать его из файла и вставить в текстовый редактор IDLE. Текст, следующий за символами >> — это вывод интерактивной оболочки. Далее в книге вы будете встречать символы >> после некоторых примеров – так будет указываться вывод программы в интерактивной оболочке. Многоточие (`...`) обозначает «и так далее».

Если после примера отсутствуют символы >>, тогда либо программа не предусматривает вывода, либо я объясняю общую концепцию и вывод не важен.

Весь текст, набранный шрифтом `Courier New`, относится к коду, выводу или программистскому жаргону. Например, если я упоминаю ключевое слово `for` из предыдущего примера, оно будет написано шрифтом `Courier New`.

Courier New — это шрифт с фиксированной шириной (моноширинный шрифт), который часто используется для отображения программного кода. Каждый знак имеет одинаковую ширину, так что отступы и другие элементы выравнивания кода легче отслеживать.

Вы можете запускать код в примерах или из оболочки, или из файлов с расширением *.py*. Помните, что, как я упоминал ранее, что в вашем случае вывод в оболочке может быть несколько иным? Из-за этого результат работы программ может немного отличаться. Если в примере содержится вывод, но отсутствует слово `print`, нужно вводить код в оболочку. Если в примере есть слово `print`, нужно запускать код из файла *.py*.

Комментарии

Комментарий — это строка (или часть строки) кода, написанная на русском (или любом другом языке), которой предшествует специальный символ, указывающий языку программирования игнорировать эту строку (или часть строки) кода. В Python для создания комментариев используется символ `#`.

Комментарий объясняет, что делает строка кода. Программисты используют комментарии, чтобы упростить понимание строки кода для тех, кто ее читает. Вы можете писать в комментарии все, что захотите, главное, в одну строку.

Python_ex003.py

```
1 # Комментарий
2 print("Привет, мир!")
```

```
>> Привет, мир!
```

Пишите комментарий только в том случае, если в своем коде вы делаете что-то необычное или объясняете то, что не является очевидным исходя из самого кода. Используйте комментарии экономно — не комментируйте каждую строку кода — храните их для особых ситуаций. Ниже приведен пример необязательного комментария.

Python_ex004.py

```
1 # вывести строку Привет, мир!
2 print("Привет, мир!")
```

Этот комментарий необязателен, поскольку и так понятно, что делает строка кода. Ниже приведен пример хорошего комментария.

Python_ex005.py

```
1 import math

2 # Длина диагонали
3 l = 4
4 w = 10
5 d = math.sqrt(l**2 + w**2)
```

Даже если вы поняли, как работает этот код, вы все еще можете не знать, как рассчитать длину диагонали прямоугольника, так что комментарий уместен.

Вывод

В своих программах вы можете выводить все, что пожелаете, не только `Привет, мир!`, но не забывайте заключать текст в кавычки.

Python_ex006.py

```
1 print("Python")
```

```
>> Python
```

Python_ex007.py

```
1 print("Хай!")
```

```
>> Хай!
```

Строки кода

Программы Python состоят из кода. Взгляните на следующую программу:

Python_ex008.py

```
1 # строка1
2 # строка2
3 # строка3
```

Перед вами три строки кода. Удобно ссылаться на каждый фрагмент кода по номеру строки, в которой он находится. В IDLE вы можете выбрать команду меню **Edit ⇨ Go to Line** (Правка ⇨ Перейти к строке), чтобы перейти к определенной строке в вашей программе. За один раз в оболочку можно вводить лишь одну строку кода, нельзя копировать и вставлять несколько строк.

Иногда фрагмент кода слишком длинный и не вмещается в одну строку. Код, помещенный в тройные кавычки, круглые, квадратные и фигурные скобки, может быть продолжен на следующей строке.

Python_ex009.py

```
1 print("""Это очень очень очень
2       очень очень очень длинная
3       строка кода.""")
```

Также, чтобы продолжить код на следующей строке, можно использовать символ обратного слеша (\).

Python_ex010.py

```
1 print\
2 ("""Это очень очень очень очень очень
3  очень длинная строка кода.""")
```

Оба эти примера выводят одно и то же. Обратный слеш позволил мне поместить код `("""Это очень очень очень очень очень очень длинная строка кода.""")` и `print` на отдельных строках, что без него сделать было бы нельзя.

Ключевые слова

В языках программирования типа Python существуют слова со специальным значением, называемые **ключевыми словами**. Ключевое слово `for` уже встречалось нам — оно используется для выполнения кода несколько раз. В этой главе вы узнаете о других ключевых словах.

Отступы

Давайте еще раз взглянем на вашу программу, выводящую сто раз строку `Привет, мир!`.

Python_ex011.py

```
for i in range(100):
    print("Привет, мир!")
```

Как я отмечал ранее, перед ключевым словом `print` указывается отступ в четыре пробела. Если коротко, то это делается за тем, чтобы Python мог определить, когда начинаются и заканчиваются блоки кода. Также обратите внимание, что если в коде используется отступ, этот отступ содержит четыре пробела. Ваша программа не будет работать, если вы используете неправильный отступ, например, три или пять пробелов, или неправильных строк.

Другие языки программирования не используют отступы таким образом; вместо этого, в них применяются ключевые слова или квадратные скобки. Ниже показана та же самая программа, написанная на языке программирования JavaScript.

Python_ex012.py

```
# Это программа JavaScript.
# Она не будет работать.

for (i = 0; i < 100; i++) {
    console.log("Привет, мир!");
}
```

Сторонники Python считают, что использование правильного отступа делает Python менее утомительным для чтения и при написании кода в сравнении с другими языками. Как в примере выше, даже если пустые промежутки не являются частью кода, программисты все равно используют их, чтобы код было легче читать.

Типы данных

Данные в Python сгруппированы в различные категории, называемые **типами данных**. В Python каждое значение, например 2 или `"Привет, мир!"`, называется **объектом**. Вы узнаете больше об объектах в части II, но пока рассмотрим объект как значение данных в Python, которое имеет три свойства: идентификатор, тип данных и, собственно, значение. Идентификатор объекта — это место, где он хранится в памяти вашего компьютера, и оно никогда не меняется. Тип данных объекта — это категория данных, к которой принадлежит объект, она определяет свойства объекта и также никогда не меняется. Значение объекта — это данные, которые он представляет, например, число 2 имеет значение 2.

`"Привет, мир!"` — объект с типом данных **`str`**, сокращение от английского слова string — **строка**, и значением `"Привет, мир!"`. Когда вы ссылаетесь на объект с типом данных `str`, то называете его строкой. Строка представляет собой последовательность из одного или нескольких символов, помещенных в кавычки. **Символ** — это один знак, вроде `a` или `1`. Вы можете использовать одинарные или двойные кавычки, но они должны быть одинаковыми в начале и в конце строки.

Python_ex013.py

```
1 ▌ "Привет, мир!"
```

```
>> "Привет, мир!"
```

Python_ex014.py

```
1 ▌ 'Привет, мир!'
```

```
>> "Привет, мир!"
```

Строки используются для представления текста и обладают уникальными свойствами.

Числа, которые вы использовали для математических операций в предыдущем разделе, также являются объектами, но не являются строками. Целые числа (`1`, `2`, `3`, `4` и т.д.) имеют тип данных **`int`**, сокращенно от английского слова integer — **целый**. Подобно строкам, у целых чисел есть особые свойства. Например, вы можете выполнить умножение двух целых чисел, но не можете умножить две строки.

Вещественные числа (числа с десятичной точкой) имеют тип данных **`float`**. `2.1`, `8.2` и `9.9999` все являются объектами с типом данных `float`. Они называются **числами с плавающей точкой**. Как и у всех остальных типов данных, у чисел с плавающей точкой есть особые свойства, и они ведут себя определенным образом, подобно целым числам.

Python_ex015.py

```
1 ▌ 2.2 + 2.2
```

```
>> 4.4
```

Объекты, имеющие тип данных **`bool`**, называются булевыми или **логическими**, они принимают значение `True` или `False`, то есть могут быть истинными или ложными.

Python_ex016.py

```
1 ▌ True
```

```
>> True
```

Python_ex017.py

```
1 ▌ False
```

```
>> False
```

Объекты с типом данных **`NoneType`** всегда имеют значение **`None`**. Они используются для представления отсутствия значения.

Python_ex018.py

```
1 ▌ None
```

На протяжении этой главы я объясняю, как использовать различные типы данных.

Константы и переменные

Вы можете использовать Python для выполнения математических операций, точно так же, как если бы использовали калькулятор. Вы можете складывать, вычитать, делить, умножать, возводить в степень и делать многое другое. Не забудьте ввести в оболочку все примеры этого раздела.

Python_ex019.py

```
1 2 + 2

>> 4
```

Python_ex020.py

```
1 2 - 2

>> 0
```

Python_ex021.py

```
1 4 / 2

>> 2.0
```

Python_ex022.py

```
1 2 * 2

>> 4
```

Константа — это значение, которое никогда не меняется. Каждое из чисел в предыдущем примере является константой: число два всегда будет представлять значение 2. **Переменная** же напротив относится к значению, которое может измениться. Переменная состоит из имени, в котором есть один или несколько символов. Это имя присваивается значению с помощью **оператора присваивания** (символа =).

Некоторые языки программирования требуют, чтобы программист указывал «объявления» переменных, которые сообщают языку, какой тип данных будет иметь переменная. Например, в языке программирования C переменная создается следующим образом:

```
1 # Не выполнять.

2 int a;
3 a = 144;
```

В Python с этим проще: вы создаете переменную, просто присваивая ей значение с помощью оператора присваивания.

Python_ex023.py

```
1 b = 100
2 b
```

```
>> 100
```

Ниже показано, как изменять значение переменной:

Python_ex024.py

```
1 x = 100
2 x

3 x = 200
4 x
```

```
>> 100
>> 200
```

Вы также можете использовать две переменные, чтобы выполнять арифметические операции.

Python_ex025.py

```
1 x = 10
2 y = 10
3 z = x + y
4 z
5 a = x - y
6 a
```

```
>> 20
>> 0
```

Часто при программировании необходимо **увеличить** (инкрементировать) или **уменьшить** (декрементировать) значение переменной. Поскольку это стандартная операция, у Python есть специальный синтаксис, позволяющий сокращать код. Чтобы увеличить значение переменной, назначьте переменную самой себе, а по другую сторону от знака равенства прибавьте к этой переменной число, на которое вы хотите увеличить ее значение.

Python_ex026.py

```
1 x = 10
2 x = x + 1
3 x
```

```
>> 11
```

Чтобы уменьшить значение переменной, сделайте то же самое, только теперь вычтите число, на которое вы хотите уменьшить значение переменной.

Python_ex027.py

```
1 x = 10
2 x = x - 1
3 x
```

```
>> 9
```

Эти примеры совершенно верны, но существует более лаконичный способ увеличивать и уменьшать значения переменных, и лучше использовать его.

Python_ex028.py

```
1 x = 10
2 x += 1
3 x
```

```
>> 11
```

Python_ex029.py

```
1 x = 10
2 x -= 1
3 x
```

```
>> 9
```

Тип данных, которые можно хранить в переменных, не ограничивается целыми числами. Переменные могут ссылаться на любой тип.

Python_ex030.py

```
1 hi = "Привет, мир!"
```

Python_ex031.py

```
1 my_float = 2.2
```

Python_ex032.py

```
1 my_boolean = True
```

Вы можете присваивать переменным любые имена, но должны следовать четырем правилам:

1. Имена переменных не могут содержать пробелы. Если вы хотите использовать в имени два слова, укажите между ними знак нижнего подчеркивания: например, `my_variable = "Строка!"`.
2. Имена переменных могут содержать только латинские буквы, цифры и символ подчеркивания.

3. Нельзя начинать имя переменной с цифры. И хотя вы можете начинать переменную с подчеркивания, пока что так не делайте — это отдельная тема, которую мы рассмотрим позже.
4. Вы не можете использовать ключевые слова Python в качестве имен переменных. Список ключевых слов можно найти по адресу **zetcode.com/lang/python/keywords**.

Синтаксис

Синтаксис — это набор правил, принципов и процессов, которые определяют структуру предложений на определенном языке, в частности, порядок слов[4]. У русского языка есть синтаксис, и у Python есть синтаксис.

В Python строки всегда берутся в кавычки — это пример синтаксиса Python. Ниже приведена синтаксически правильная программа Python.

Python_ex033.py

```
1 print("Привет, мир!")
```

Она верна, потому что вы следовали синтаксису Python, используя кавычки при определении строки. Если бы вы использовали кавычки только с одной стороны текста, то нарушили бы синтаксис Python, и ваш код бы не работал.

Ошибки и исключения

Если вы пишите программу на Python и игнорируете синтаксис этого языка, то при запуске своей программы получите одну или несколько ошибок. Оболочка Python сообщит вам, что ваш код не работает, и выдаст информацию об ошибке. Взгляните, что произойдет, если в Python вы попытаетесь определить строку с кавычками только с одной стороны.

Python_ex034.py

```
1 # Код содержит ошибку.

2 my_string = "Привет, мир.
```

```
>> File "/Users/coryalthoff/PycharmProjects/se.py", line 1 my_
string = 'd ^ SyntaxError: EOL while scanning string literal
```

Это сообщение указывает, что ваша программа содержит **синтаксическую ошибку**. Синтаксические ошибки фатальны; программа, содержащая их, не может работать. Когда вы пытаетесь запустить программу с синтаксической ошибкой, Python сообщает об этом в оболочке. Сообщение предоставит вам информацию о том, в каком файле была ошибка, в какой она произошла строке, и что это была за ошибка. Хотя ошибки могут показаться пугающими, они случаются постоянно.

Если в вашем коде была допущена ошибка, вы должны перейти к номеру строки кода, в которой возникла проблема, и попытаться выяснить, что было

[4] pythonworld.ru/osnovy/sintaksis-yazyka-python.html

сделано неправильно. В данном примере нужно перейти к первой строке. Если вы внимательно к ней присмотритесь, то заметите, что она содержит лишь открывающие кавычки. Чтобы исправить это, закройте кавычки в конце строки и повторно запустите программу. С этого момента я буду оформлять вывод ошибки следующим образом:

```
>> SyntaxError: EOL while scanning string literal
```

Для удобства чтения я буду указывать последнюю строку ошибки.

У Python есть два типа ошибок: синтаксические ошибки и исключения. **Исключением** называется любая ошибка, которая не является синтаксической. `ZeroDivisionError` — пример исключения, которое возникает, если вы попытаетесь делить на ноль.

В отличие от синтаксических ошибок, исключения не обязательно фатальны (в следующей главе вы узнаете о способе заставить программу работать даже при наличии исключения). Когда появляется исключение, программисты Python говорят, что «Python (или вашей программой) было сгенерировано исключение». Ниже приведен пример исключения.

Python_ex035.py

```
1 # Код содержит ошибку.

2 10 / 0
```

```
>> ZeroDivisionError: division by zero
```

Если в своем коде вы неправильно использовали отступы, то получите ошибку `IndentationError`.

Python_ex036.py

```
1 # Код содержит ошибку.
2 y = 2
3           x = 1
```

```
>> IndentationError: unexpected indent
```

В процессе обучения программированию вы будете часто получать синтаксические ошибки и исключения (включая те, которые здесь не рассматриваются), но со временем их количество снизится. Помните, когда вы сталкиваетесь с синтаксической ошибкой или исключением, переходите к строке, где возникла проблема, и всматривайтесь в нее, пока не найдете решение (а если вы оказались в тупике, выполните поиск по тексту ошибки или исключению в Интернете).

Арифметические операторы

Ранее вы использовали Python для выполнения простых арифметических вычислений, например, `4 / 2`. Специальные символы, которые мы использовали в этих примерах, называются **операторами**. Операторы в Python делятся на не-

сколько категорий, и те, что вы встречали до сих пор, называются **арифметическими операторами**. Ниже приведен список некоторых наиболее распространенных арифметических операторов в Python.

Оператор	Значение	Пример	Результат
**	Возведение в степень	2 ** 2	4
%	Деление по модулю	14 % 4	2
//	Целочисленное деление	13 // 8	1
/	Деление	13 / 8	1.625
*	Умножение	8 * 2	16
-	Вычитание	7 - 1	6
+	Сложение	2 + 2	4

При выполнении операции деления получают частное и остаток. Частное — это результат деления, а остаток — то, что осталось после операции. Оператор деления по модулю возвращает значение остатка. Например, деление 13 на 5 даст 2 и 3 в остатке.

Python_ex037.py

```
1 13 // 5
```

```
>> 2
```

Python_ex038.py

```
1 13 % 5
```

```
>> 3
```

Когда вы используете деление по модулю для двух чисел, и остаток отсутствует (соответствующий оператор возвращает 0), число четное. Если есть остаток, число нечетное.

Python_ex039.py

```
1 # четное
2 12 % 2
```

```
>> 0
```

Python_ex040.py

```
1 # нечетное
2 11 % 2
```

```
>> 1
```

Для деления используются два оператора. Первый, //, возвращает неполное частное.

Python_ex041.py

```
1 ▌ 14 // 3
```

```
>> 4
```

Второй, /, возвращает результат деления первого числа на второе в виде числа с плавающей точкой.

Python_ex042.py

```
1 ▌ 14 / 3
```

```
>> 4.666666666666667
```

Вы можете возвести число в степень с помощью оператора **.

Python_ex043.py

```
1 ▌ 2 ** 2
```

```
>> 4
```

Значения (в этом случае числа) по обе стороны от оператора называются **операндами**. Вместе два операнда и оператор образуют **выражение**. Когда ваша программа запускается, Python вычисляет выражение и возвращает единственное значение. Когда вы вводите в оболочку выражение 2 + 2, Python вычисляет его в 4.

Порядок операций — это набор правил, используемых для вычисления выражения. Порядок операций в математических равенствах следующий: круглые скобки, возведение в степень, умножение, деление, сложение и вычитание. Скобки являются первоочередными по отношению к возведению в степень, степень является первоочередной по отношению к умножению и делению, а те выполняются перед сложением и вычитанием. Если у операторов одинаковая очередность, как в случае 15 / 3 × 2, операция выполняется слева направо. В данном случае ответом является результат деления 15 на 3, умноженный на 2. Python следует тому же порядку операций, когда вычисляет значения математических выражений.

Python_ex044.py

```
1 ▌ 2 + 2 * 2
```

```
>> 6
```

Python_ex045.py

```
1 ▌ (2 + 2) * 2
```

```
>> 8
```

В первом примере сначала вычисляется 2 * 2, поскольку умножение имеет приоритет перед сложением.

Во втором примере сначала вычисляется (2 + 2), поскольку выражения в скобках Python всегда вычисляет в первую очередь.

Операторы сравнения

Операторы сравнения — иная категория операторов в Python. Подобно арифметическим операторам, они используются в выражениях с операндами с обеих сторон. В отличие от выражений с арифметическими операторами, выражения с операторами сравнения принимают значение `True` или `False`.

Оператор	Значение	Пример	Результат
>	Больше	`100 > 10`	`True`
<	Меньше	`100 < 10`	`False`
>=	Больше или равно	`2 >= 2`	`True`
<=	Меньше или равно	`1 <= 4`	`True`
==	Равно	`6 == 9`	`False`
!=	Не равно	`3 != 2`	`True`

Выражение с оператором > возвращает значение `True`, если число слева больше числа справа, и `False` в противном случае.

Python_ex046.py

```
1 ▌ 100 > 10
```

```
>> True
```

Выражение с оператором < возвращает значение `True`, если число слева меньше числа справа, и `False` в противном случае.

Python_ex047.py

```
1 ▌ 100 < 10
```

```
>> False
```

Выражение с оператором >= возвращает значение `True`, если число слева больше или равно числу справа. В противном случае выражение возвращает `False`.

Python_ex048.py

```
1 ▌ 2 >= 2
```

```
>> True
```

Выражение с оператором <= возвращает значение `True`, если число слева меньше или равно числу справа. В противном случае выражение возвращает `False`.

Python_ex049.py

```
1 ▌ 2 <= 2
>> True
```

Выражение с оператором == возвращает значение `True`, если два операнда равны. В противном случае выражение возвращает `False`.

Python_ex050.py

```
1 ▌ 2 == 2
```

```
>> True
```

Python_ex051.py

```
1 ▌ 1 == 2
```

```
>> False
```

Выражение с оператором != возвращает значение `True`, если два операнда не равны. В противном случае выражение возвращает `False`.

Python_ex052.py

```
1 ▌ 1 != 2
```

```
>> True
```

Python_ex053.py

```
1 ▌ 2 != 2
```

```
>> False
```

Ранее вы назначали переменные числам, используя оператор =, например `x = 100`. Возможно, у вас был соблазн прочитать это как «x равно 100», но это неверно. Как вы уже заметили, символ = используется для присвоения значения переменной, а не для проверки на равенство. Когда вы видите `x = 100`, читайте выражение как «x получает сто». Оператор сравнения == используется для проверки равенства, поэтому, если вы видите `x == 100`, читайте выражение как «x равно 100».

Логические операторы

Логические операторы — еще одна категория операторов в Python. Как и в случае с операторами сравнения, выражения с логическими операторами также принимают значение `True` или `False`.

Оператор	Значение	Пример	Результат
`and`	И	`True and True`	`True`
`or`	ИЛИ	`True or False`	`True`
`not`	НЕ	`not True`	`False`

Ключевое слово `and` принимает два выражения и возвращает `True`, если выражения принимают значение `True`. Если какое-либо из выражений `False`, оператор `and` возвращает `False`.

Python_ex054.py

```
1 ▌ 1 == 1 and 2 == 2
```

```
>> True
```

Python_ex055.py

```
1 ▌ 1 == 2 and 2 == 2
```

```
>> False
```

Python_ex056.py

```
1 ▌ 1 == 2 and 2 == 1
```

```
>> False
```

Python_ex057.py

```
1 ▌ 2 == 1 and 1 == 1
```

```
>> False
```

В одном выражении ключевое слово `and` можно использовать несколько раз.

Python_ex058.py

```
1 ▌ 1 == 1 and 10 != 2 and 2 < 10
```

```
>> True
```

Ключевое слово `or` принимает два и более выражения и возвращает `True`, если хотя бы одно из них принимает значение `True`.

Python_ex059.py

```
1 ▌ 1==1 or 1==2
```

```
>> True
```

Python_ex060.py

```
1 ▌ 1==1 or 2==2
```

```
>> True
```

Python_ex061.py

```
1 ▌ 1==2 or 2==1
```

```
>> False
```

Python_ex062.py

```
1 ▌ 2==1 or 1==2
```

```
>> False
```

Как и and, ключевое слово or в одном выражении можно использовать несколько раз.

Python_ex063.py

```
1 1==1 or 1==2 or 1==3
```

```
>> True
```

Это выражение истинно (равно True), поскольку 1 == 1 является True, хотя все остальные выражения принимают значение False.

Вы можете поместить ключевое слово not перед выражением, и оно изменит значение выражения на противоположное. Если выражение принимало значение True, то как только перед ним будет помещено not, оно станет False.

Python_ex064.py

```
1 not 1 == 1
```

```
>> False
```

Python_ex065.py

```
1 not 1 == 2
```

```
>> True
```

Условные инструкции

Ключевые слова if, elif и else используются в **условных инструкциях**. Условные инструкции — это тип **структуры управления потоком**: блок кода, который может принимать решения, анализируя значения переменных. Условная инструкция представляет собой код, который может выполнять дополнительный код в зависимости от определенного условия. Ниже приведен пример в **псевдокоде** (ненастоящий код, используемый для иллюстрации примера), чтобы прояснить принцип работы.

```
1 # Не выполнять

2 If (expression) Then
3             (code_area1)
4 Else
5             (code_area2)
```

Этот псевдокод объясняет, что вы можете определить две условных инструкции, которые работают вместе. Если выражение, определенное в первой условной инструкции, равно True, выполняется весь код, определенный в code_area1. Если выражение, определенное в первой условной инструкции, равно False, выполняется весь код, определенный в code_area2. Первая часть примера называется **инструкцией if**, а последняя называется **инструкцией else**.

Вместе они образуют **конструкцию if-else**: способ, которым программисты говорят «если это произойдет, сделайте это, в противном случае сделайте то». Ниже приведен пример инструкции `if-else` в Python.

Python_ex066.py

```
1 home = "Россия"
2 if home == "Россия":
3     print("Привет, Россия!")
4 else:
5     print("Привет, мир!")
```

```
>> Привет, Россия!
```

Строки кода 5 и 6 образуют инструкцию `if`. Инструкция `if` состоит из строки кода, начинающейся с ключевого слова `if`, за которым следует выражение, двоеточие, отступ и одна или несколько строк кода, которые должны выполняться, если выражение в первой строке принимает значение `True`. Строки 7 и 8 образуют инструкцию `else`. Инструкция `else` начинается с ключевого слова `else`, за которым следует двоеточие, отступ и одна или несколько строк кода, которые должны выполняться, если выражение в инструкции `if` равно `False`.

Вместе они образуют инструкцию `if-else`. В этом примере выводится строка `Привет, Россия!`, потому что выражение в инструкции `if` имеет значение `True`. Если вы измените значение переменной `home` на `Казахстан`, выражение в инструкции `if` примет значение `False` и запустится инструкция `else`, что приведет к выводу другой строки, `Привет, мир!`.

Python_ex067.py

```
1 home = "Казахстан"
2 if home == "Россия":
3     print("Привет, Россия!")
4 else:
5     print("Привет, мир!")
```

```
>> Привет, мир!
```

Вы можете использовать инструкцию `if` отдельно.

Python_ex068.py

```
1 home = "Россия"
2 if home == "Россия":
3     print("Привет, Россия!")
```

```
>> Привет, Россия!
```

Можно использовать несколько инструкций `if-else` подряд.

Python_ex069.py

```
x = 2
if x == 2:
    print("Число: 2.")
if x % 2 == 0:
    print("Число: четное.")
if x % 2 != 0:
    print("Число: нечетное.")
```

```
>> Число: 2.
>> Число: четное.
```

Каждая инструкция `if` выполнит свой код только в том случае, если ее выражение принимает значение `True`. В этом случае первые два выражения принимают значение `True`, так что их код выполняется, но третье выражение принимает значение `False`, и его код не выполняется.

Если вы действительно хотите сойти с ума, то можете даже поместить одну инструкцию `if` внутрь другой инструкции `if` (это называется вложением).

Python_ex070.py

```
x = 10
y = 11

if x == 10:
    if y == 11:
        print(x + y)
```

```
>> 21
```

В этом случае `x + y` будет выводиться только в том случае, если выражения в обеих инструкциях `if` принимают значение `True`.

Инструкцию `else` нельзя использовать саму по себе, а только в конце инструкции `if-else`.

Мы используем ключевое слово `elif` для создания **инструкций `elif`**. `elif` означает сокращение от «else if». Инструкции `elif` можно бесконечно добавлять в инструкцию `if-else`, чтобы позволить ей принимать дополнительные решения.

Если внутри инструкции `if-else` есть инструкции `elif`, значение выражения инструкции `if` оценивается первым. Если это выражение равно `True`, выполняется только его код. Однако если он принимает значение `False`, тогда оценивается каждая последующая инструкция `elif`. Как только выражение в инструкции `elif` принимает значение `True`, выполняется ее код и никакой другой. Если ни одна из инструкций `elif` не принимает значение `True`, выполняется код в инструкции `else`. Ниже приведен пример инструкции `if-else` с инструкциями `elif` внутри.

Python_ex071.py

```
home = "Тайланд"
if home == "Япония":
    print("Привет, Япония!")
elif home == "Тайланд":
    print("Привет, Тайланд!")
elif home == "Индия":
    print("Привет, Индия!")
elif home == "Китай":
    print("Привет, Китай!")
else:
    print("Привет, мир!")
```

```
>> Привет, Тайланд!
```

Ниже приведен пример, где ни одна из инструкций `elif` не принимает значение `True`, поэтому выполняется код инструкции `else`.

Python_ex072.py

```
home = "Марс"
if home == "Россия":
    print("Привет, Россия!")
elif home == "Казахстан":
    print("Привет, Казахстан!")
elif home == "Тайланд":
    print("Привет, Тайланд!")
elif home == "Мексика":
    print("Привет, Мексика!")
else:
    print("Привет, мир!")
```

```
>> Привет, мир!
```

Наконец, ваш код может содержать несколько инструкций `if` и `elif`.

Python_ex073.py

```
x = 100
if x == 10:
    print("10!")
elif x == 20:
    print("20!")
else:
```

```
    print("Не знаю!")

if x == 100:
    print("x равно 100!")

if x % 2 == 0:
   print("x четное!")
else:
    print("x нечетное!")
```

```
>> Не знаю!
>> x равно 100!
>> x четное!
```

Инструкции

Инструкция — это технический термин, описывающий различные части языка Python. Можете рассматривать инструкцию Python как команду или вычисление. В данном разделе вы подробно рассмотрите синтаксис инструкций. Не беспокойтесь, если сначала кое-что из этого покажется запутанным — чем больше времени вы будете тратить на практику, тем более понятным все будет становиться. Этот раздел поможет вам понять несколько концепций программирования.

В Python есть два вида инструкций: **простые** и **составные**. Простые инструкции могут быть выражены в одной строке кода, тогда как составные обычно занимают несколько строк. Ниже приведены примеры простых инструкций.

Python_ex074.py

```
print("Привет, мир!")
```

```
>> Привет, мир!
```

Python_ex075.py

```
2 + 2
```

```
>> 4
```

Инструкции `if` и `else`, а также первая написанная вами в этой главе программа, выводящая строку `Привет, мир!` сто раз, — это все примеры составных инструкций.

Составные инструкции состоят из одной или нескольких **ветвей**. Ветвь состоит из двух или более строк кода: **заголовка**, за которым следует **тело**. Заголовок представляет собой строку кода в ветви, содержащую ключевое слово, за которым следует двоеточие, и последовательность из одной или нескольких строк с отступом. После отступа содержится тело, состоящее из одной или двух простых инструкций. Тело — это всего лишь строка кода внутри ветви. Заголовок управляет телом. Наша программа, выводящая `Привет, мир!` сто раз, состоит из одной составной инструкции.

Python_ex076.py

```
for i in range(100):
    print("Привет, мир!")
```

```
>> Привет, мир!
>> Привет, мир!
>> Привет, мир!
...
```

Первая строка этой программы — заголовок. Он состоит из ключевого слова `for`, за которым следует двоеточие. После отступа находится тело — код `print("Привет, мир!")`. В этом случае заголовок использует тело для вывода `Привет, мир!` сто раз. Такой код называется циклом; о нем вы узнаете больше в главе 7. Этот код содержит только одну ветвь.

В составной инструкции также может быть несколько ветвей. Вы уже сталкивались с этим в случаях с инструкциями `if-else`. Всегда, когда за инструкцией `if` следует инструкция `else`, результатом является составная инструкция с несколькими ветвями. Когда в составной инструкции есть несколько ветвей, эти ветви работают вместе. В случае с составной инструкцией `if-else`, когда инструкция `if` принимает значение `True`, выполняется тело инструкции `if`, а тело инструкции `else` не выполняется. Когда инструкция `if` принимает значение `False`, тело инструкции `if` не выполняется, и вместо этого выполняется `else`. Последний пример из предыдущего раздела содержит три составных инструкции.

Python_ex077.py

```
x = 100
if x == 10:
    print("10!")
elif x == 20:
    print("20!")
else:
    print("Не знаю!")

if x == 100:
    print("x равно 100!")

if x % 2 == 0:
    print("x четное!")
else:
    print("x нечетное!")
```

```
>> Не знаю!
>> x равно 100!
>> x четное!
```

Первая составная инструкция содержит три ветви, вторая — одну, а последняя — две.

И последнее, что касается инструкций, — между ними могут быть пробелы. Пробелы между инструкциями не влияют на код. Иногда пустое пространство добавляют, чтобы сделать код более читабельным.

Python_ex078.py

```
1 print("Майкл")



2 print("Джордан")
```

```
>> Майкл
>> Джордан
```

Словарь терминов

`Bool`: тип данных логических объектов.
`Float`: тип данных вещественных чисел (чисел с десятичной точкой).
`Int`: тип данных целых чисел.
`None`: объект с типом данных `NoneType`. Его значением всегда является `None`.
`NoneType`: тип данных объектов `None`.
`Str`: тип данных строки.
Арифметический оператор: категория операторов, используемых в арифметических выражениях.
Ветвь: строительные блоки составных инструкций. Ветвь состоит из двух или более строк кода: заголовка, за которым следует тело.
Выражение: код с оператором между двумя операндами.
Декрементирование: уменьшение значение переменной.
Заголовок: строка кода в ветви, содержащая ключевое слово, за которым следует двоеточие, и последовательность из одной или нескольких строк с отступом.
Инкрементирование: увеличение значения переменной.
Инструкция `elif`: инструкции, которые можно бесконечно добавлять в инструкцию `if-else`, позволяя последней принимать дополнительные решения.
Инструкция `else`: вторая часть инструкции `if-else`.
Инструкция `if`: первая часть инструкции `if-else`.
Инструкция `if-else`: способ, которым программисты говорят «если это произойдет, сделайте это, в противном случае сделайте то».
Инструкция: команда или вычисление.
Исключение: ошибка, не являющаяся фатальной.
Ключевое слово: слово в языке программирования, имеющее особый смысл. Все ключевые слова Python можно найти на сайте **theselftaughtprogrammer.io/keywords**.
Комментарий: строка (или часть строки) кода, написанная на русском (или другом) языке, которой предшествует специальный символ, сообщающий исполь-

зуемому вами языку программирования, что эту строку (или часть строки) кода следует проигнорировать.
Константа: значение, которое никогда не изменяется.
Логический оператор: категория операторов, оценивающих значения двух выражений и возвращающих либо `True`, либо `False`.
Логический: объект с типом данных `bool`. Его значением является либо `True` (истина), либо `False` (ложь).
Объект: значение с тремя свойствами: идентификатором, типом данных и значением.
Операнд: значение с любой стороны от оператора.
Оператор присваивания: символ = в Python.
Оператор сравнения: категория операторов, которые используются в выражении, принимающем значение либо `True` (истина), либо `False` (ложь).
Оператор: символы, использующиеся в выражении с операндами.
Переменная: имя, которому с помощью оператора присваивания присвоено значение.
Порядок операций: набор правил, используемых в математике при подсчете выражений.
Простая инструкция: инструкция, которую можно выразить с помощью одной строки кода.
Псевдокод: ненастоящий код, используемый для иллюстрации примера.
Символ: один знак, например `a` или `1`.
Синтаксис: комплекс правил, принципов и процессов, определяющих структуру предложений в данном языке, в частности, порядок слов[5].
Синтаксическая ошибка: фатальная ошибка в программировании, вызванная нарушением синтаксиса языка программирования.
Составная инструкция: инструкция, обычно занимающая больше одной строки кода.
Строка: объект с типом данных `str`. Значением строки является последовательность одного или больше символов, помещенных в кавычки.
Структура управления потоком: блок кода, который принимает решения, анализируя значения переменных.
Тело: строка кода внутри ветви, которой управляет заголовок.
Тип данных: категория данных.
Условная инструкция: код, выполняющий дополнительный код при наступлении определенных условий.
Целое: объект с типом данных `int`. Его значением является целое число.
Число с плавающей точкой: объект с типом данных `float`. Его значением является вещественное число (число с десятичной точкой).

Практикум

1. Выведите три различные строки.
2. Напишите программу, которая будет выводить одно сообщение, если значение переменной меньше 10, и другое, если переменная больше или равна 10.
3. Напишите программу, которая будет выводить одно сообщение, если значе-

[5] stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-forand-how-is-it-used

ние переменной меньше или равно 10, другое, если переменная больше 10, но меньше или равна 25, и третье сообщение, если переменная больше 25.

4. Напишите программу, которая выполняет деление двух чисел и выводит остаток.
5. Напишите программу, которая принимает две переменные, делит одну на другую и выводит частное.
6. Напишите программу, которая будет выводить разные строки в зависимости от того, какое целое число было вами присвоено переменной `age`, содержащейся в этой программе.

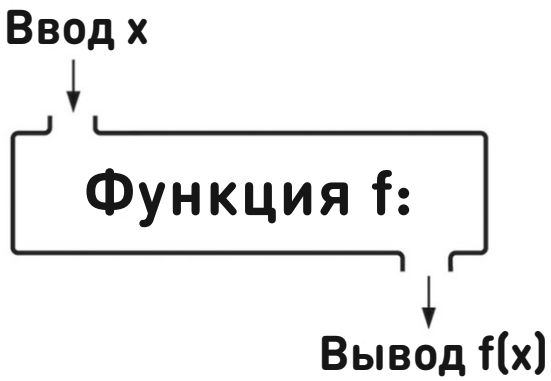
Решения: *ch3_challenge1.py – ch3_challenge6.py*.

Глава 4. Функции

Функции должны делать одну вещь и делать ее хорошо
и делать только ее.

Роберт С. Мартин

В этой главе вы изучите **функции** — составные инструкции, которые могут принимать данные ввода, выполнять указания и возвращать данные вывода. Функции позволяют определять и повторно использовать определенную функциональность.



Синтаксис

С этого момента для объяснения концепций программирования я буду использовать новое **соглашение** (согласованный способ делать что-то). Например, для иллюстрации использования функции вывода данных я буду записывать следующее: `print("вывод_на_экран")`.

Здесь я объединил код Python и курсивный текст — это суть данной концепции. В подобных примерах все, кроме курсивного текста внутри, является допустимым кодом Python. При выполнении примера курсивный текст нужно заменить описанным кодом. Этот текст подсказывает, какой именно код нужно использовать. В данном случае, значение *вывод_на_экран* надо заменить тем текстом, который нужно вывести на экран.

Функции

Вызвать функцию — значит передать ей данные ввода, необходимые для выполнения указаний и возвращения вывода. Каждый ввод в функцию является **параметром**. Когда вы передаете функции параметр, это называется **передача** параметра функции.

Функции в Python похожи на математические функции. Ниже показан пример, если вы не помните функции из алгебры:

```
# Не выполнять

f(x) = x * 2
```

Левая часть инструкции сверху определяет функцию, `f`, принимающую один параметр, `x`. Правая часть — определение функции, которое использует переданный параметр (`x`), чтобы произвести вычисление и вернуть результат (вывод). В этом случае значением функции является ее параметр, умноженный на два.

Как в Python, так и в алгебре, функция записывается следующим образом: *`имя_функции`*`(`*`параметры_через_запятую`*`)`. Чтобы вызвать функцию, после ее имени нужно указать круглые скобки и поместить внутрь параметры, отделив каждый из них запятой. Для математической функции `f`, определенной как `f(x) = 2 * x`, значение `f(2)` равно `4`, значение `f(10)` равно `20`.

Определение функций

Для создания функций в Python выберите ее имя, определите параметры, укажите, что функция должна делать и какое значение возвращать. Ниже показан синтаксис определения функции.

```
# Не выполнять.

def имя_функции(параметры):
    определение_функции
```

Математическая функция `f(x) = x * 2` в Python будет выглядеть вот так:

Python_ex079.py

```
def f(x):
    return x * 2
```

Ключевое слово `def` сообщает Python, что вы определяете функцию. После `def` вы указываете имя функции; оно должно отвечать тем же правилам, что и имена переменных. Согласно конвенции, в имени функции нельзя использовать заглавные буквы, а слова должны быть разделены подчеркиванием, `вот_так`.

Как только вы присвоили своей функции имя, укажите после него круглые скобки. Внутри скобок должен содержаться один или несколько параметров.

После скобок ставится двоеточие, а новая строка начинается с отступа в четыре пробела (как с любой другой составной инструкцией). Любой код с от-

ступом в четыре пробела после двоеточия является определением функции. В этом случае определение нашей функции состоит только из одной строки: `return x * 2`. Ключевое слово `return` используется для определения значения, которое функция выводит при вызове, то есть значения, которое она возвращает.

Чтобы вызвать функцию в Python, мы используем синтаксис *`имя_функции`*`(`*`параметры_через_запятую`*`)`.

Ниже описан вызов функции `f` из предыдущего примера с параметром `2`.

Python_ex080.py

```
# Продолжение
# предыдущего примера

f(2)
```

Консоль ничего не вывела. Можно сохранить вывод вашей функции в переменной и передать ее функции `print`.

Python_ex081.py

```
# Продолжение
# предыдущего примера

result = f(2)
print(result)
```

```
>> 4
```

Вы можете сохранить результат, возвращаемый вашей функцией, в переменной и использовать это значение в программе позднее.

Python_ex082.py

```
def f(x):
    return x + 1

z = f(4)

if z == 5:
    print("z равно 5")
else:
    print("z не равно 5")
```

```
>> z равно 5
```

У функции может быть один параметр, несколько параметров или вообще их не быть. Чтобы опередить функцию, не требующую параметров, оставьте круглые скобки пустыми.

Python_ex083.py

```
def f():
    return 1 + 1

result = f()
print(result)
```

```
>> 2
```

Если хотите, чтобы функция принимала больше одного параметра, отделите каждый параметр в скобках запятой.

Python_ex084.py

```
def f(x, y, z):
    return x + y + z

result = f(1, 2, 3)
print(result)
```

```
>> 6
```

Наконец, функция не обязана содержать инструкцию `return`. Если функции нечего возвращать, она возвращает значение `None`.

Python_ex085.py

```
def f():
    z = 1 + 1

result = f()
print(result)
```

```
>> None
```

Встроенные функции

Python содержит библиотеку функций, встроенных в язык программирования. Они называются **встроенными функциями**. Они выполняют всевозможные расчеты и задания и готовы к использованию без каких-либо дополнительных действий с вашей стороны. Вы уже встречали один пример встроенной функции: первая программа, написанная вами, использовала функцию `print`, чтобы вывести строку `"Привет, мир!"`.

`len` — еще одна встроенная функция. Она возвращает длину объекта – например, строки (т.е. количество символов в ней).

Python_ex086.py

```
len("Монти")
```

```
>> 5
```

Python_ex087.py

```
1 len("Пайтон")
```

```
>> 6
```

Встроенная функция `str` принимает объект и возвращает новый объект с типом данных `str`. Например, функцию `str` можно использовать для преобразования целого числа в строку.

Python_ex088.py

```
1 str(100)
```

```
>> '100'
```

Функция `int` принимает объект и возвращает его в виде целого числа.

Python_ex089.py

```
1 int("1")
```

```
>> 1
```

Функция `float` принимает объект и возвращает число с плавающей точкой.

Python_ex090.py

```
1 float(100)
```

```
>> 100.0
```

Необходимо, чтобы параметр, который вы передаете функциям `str`, `int` или `float`, мог стать строкой, целым числом или числом с плавающей точкой. Функция `str` может принимать в качестве параметра большинство объектов, но функция `int` принимает только строку, содержащую число, или объект в виде числа с плавающей точкой. Функция `float` принимает только строку, содержащую число, или объект в виде целого числа.

Python_ex091.py

```
1 int("110")
2 int(20.54)

3 float("16.4")
4 float(99)
```

```
>> 110
>> 20
>> 16.4
>> 99.0
```

Если вы попытаетесь передать функции `int` или `float` параметр, который нельзя преобразовать в целое число или число с плавающей точкой, Python сгенерирует исключение.

Python_ex092.py

```
1 int("Принц")
```

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

`input` — это встроенная функция, собирающая информацию от человека, который использует программу.

Python_ex093.py

```
1 age = input("Укажите возраст:")
2 int_age = int(age)
3 if int_age < 21:
4     print("Вы - молоды!")
5 else:
6     print("Ну вы и старик!")
```

```
>> Укажите возраст:
```

Функция `input` принимает в качестве параметра строку и отображает ее человеку, использующему программу. Затем пользователь может набрать ответ в оболочке, и его ответ можно сохранить в переменной — в нашем случае, в переменной `age`.

Затем используйте функцию `int`, чтобы преобразовать переменную `age` из строки в целое число. Функция `input` собирает данные от пользователя в виде `str`, но чтобы сравнивать переменную с целыми числами, нужно, чтобы она была `int`. Как только вы получили целое число, ваша инструкция `if-else` определяет, какое сообщение нужно выводить пользователю в зависимости от того, что тот ввел в оболочку. Если пользователь ввел число, меньшее `21`, выводится текст `Вы - молоды!`. Если было введено число, большее `21`, выводится строка `Ну вы и старик!`.

Многократное использование функций

Функции используются не только для вычисления и возвращения значений. Они могут инкапсулировать функциональность, которую вы пожелаете использовать повторно.

Python_ex094.py

```
1 def even_odd(x):
2     if x % 2 == 0:
3         print("even")
4     else:
5         print("odd")
```

```
even_odd(2)
even_odd(3)
```

```
>> четное
>> нечетное
```

Вы не определили значение, которое функция должна возвращать, но она все еще приносит пользу. Функция проверяет условие `x % 2 == 0` и в выводе сообщает, четным или нечетным является параметр `x`.

Функции также позволяют писать меньше кода благодаря многократному использованию. Ниже приведен пример программы, написанной без функций.

Python_ex095.py

```
n = input("введите число: ")
n = int(n)

if n % 2 == 0:
    print("n - четное.")
else:
  print("n — нечетное.")

n = input("введите число: ")
n = int(n)
if n % 2 == 0:
    print("n - четное.")
else:
    print("n — нечетное.")
n = input("введите число: ")
n = int(n)
if n % 2 == 0:
    print("n - четное.")
else:
    print("n — нечетное.")
```

```
>> введите число:
```

Программа трижды просит пользователя ввести число. Затем инструкция `if-else` проверяет, четное ли это число. Если оно четное, выводится сообщение `n - четное.`, в противном случае выводится `n - нечетное`.

Недостаток этой программы заключается в том, что она повторяет один и тот же код три раза. Можно сократить программу и сделать ее более удобной для чтения, поместив функциональность внутрь функции и вызвав ее трижды.

Python_ex096.py

```
def even_odd():
n = input("введите число: ")
n = int(n)
if n % 2 == 0:
    print("n - четное.")
else:
    print("n — нечетное.")

even_odd()
even_odd()
even_odd()
```

```
>> введите число:
```

Функциональность этой программы аналогична предыдущей, но поскольку вы поместили ее в функцию, которую можно вызывать по мере необходимости, код вашей программы стал короче и удобнее для чтения.

Обязательные и необязательные параметры

Функция может принимать параметры двух типов. Те, что встречались вам до этого, называются **обязательными параметрами**. Когда пользователь вызывает функцию, он должен передать в нее все обязательные параметры, иначе Python сгенерирует исключение.

В Python есть и другой вид параметров — **опциональные**. Они позволяют тому, кто вызвал функцию, при необходимости передать в нее параметр, но он не является обязательным. Опциональные параметры определяются с помощью следующего синтаксиса: *имя_функции*(*имя_параметра = значение_параметра*). Как и обязательные, опциональные параметры нужно отделять запятыми. Ниже приведен пример функции, в коде которой используется опциональный параметр.

Python_ex097.py

```
def f(x=2):
     return x**x

print(f())
print(f(4))
```

```
>> 4
>> 256
```

Сначала функция вызывается без передачи параметра. Так как параметр необязательный, `x` автоматически становится `2`, и функция возвращает `4`.

Затем та же функция вызывается с параметром `4`. Функция игнорирует значение по умолчанию, `x` становится `4`, и функция возвращает `256`. Вы можете

определить функцию, которая принимает как обязательные, так и опциональные параметры, но обязательные нужно определять в первую очередь.

Python_ex098.py

```
def add_it(x, y=10):
    return x + y

result = add_it(2)
print(result)
```

```
>> 12
```

Область видимости

У переменных есть важное свойство, называемое **областью видимости**. Когда вы определяете переменную, ее область видимости указывает на то, какая часть вашей программы может находить и изменять значение переменной. Область видимости переменной зависит от того, где в программе была определена переменная. Если вы определили ее за пределами функции (или класса, о которых вы узнаете в части II), область видимости переменной — **глобальная**. Тогда значение переменной можно найти и изменить из любой позиции программы. Переменная с глобальной областью видимости называется **глобальной переменной**. Если вы определите переменную в пределах функции (или класса), у нее будет **локальная область видимости**: ваша программа сможет найти и изменить значение переменной только в функции, в пределах которой она была определена. Ниже показаны переменные с глобальной областью видимости.

Python_ex099.py

```
x = 1
y = 2
z = 3
```

Эти переменные не были определены внутри функции (или класса), следовательно, у них глобальная область видимости. Это значит, что вы можете находить и изменять их откуда угодно, в том числе изнутри функции.

Python_ex100.py

```
x = 1
y = 2
z = 3

def f():
    print(x)
    print(y)
    print(z)
```

```
8 f()
```

```
>> 1
>> 2
>> 3
```

Если вы определите эти же переменные внутри функции, то сможете лишь находить и изменять их внутри этой функции. Если вы попытаетесь получить к ним доступ вне функции, в которой они были определены, Python сгенерирует исключение.

Python_ex101.py

```
1 def f():
2     x = 1
3     y = 2
4     z = 3

5 print(x)
6 print(y)
7 print(z)
```

```
>> NameError: name 'x' is not defined
```

Если вы определите переменные внутри функции, ваш код будет работать.

Python_ex102.py

```
1 def f():
2     x = 1
3     y = 2
4     z = 3
5     print(x)
6     print(y)
7     print(z)

8 f()
```

```
>> 1
>> 2
>> 3
```

Попытка использовать за пределами функции переменную, определенную внутри этой функции, аналогична использованию переменной, которая еще не была определена, что приводит к вызову того же исключения.

Python_ex103.py

```
if x > 100:
    print("x > 100")
```

```
>> NameError: name 'x' is not defined
```

Вы можете изменять глобальную переменную откуда угодно, но для ее изменения внутри локальной области видимости требуется дополнительный шаг. Перед переменной, которую вы желаете изменить, необходимо использовать ключевое слово `global`. Этот шаг нужен для того, чтобы удостовериться, что если вы определили переменную `x` внутри функции, вы случайно не измените значение любой переменной, ранее определенной за пределами вашей функции. Ниже приведен пример изменения глобальной переменной изнутри функции.

Python_ex104.py

```
x = 100

def f():
    global x
    x += 1
    print(x)

f()
```

```
>> 101
```

Без областей видимости вы могли бы получить доступ к каждой переменной в любой части программы, что создавало бы проблемы. Если у вас огромная программа, и вы создали функцию, использующую переменную `x`, вы можете случайно изменить значение переменной с таким же именем, ранее определенной где-то в программе. Подобные ошибки могут повлиять на поведение вашей программы и привести к ошибкам или непредвиденным результатам. Чем большей становится ваша программа и чем больше она содержит переменных, тем вероятнее это произойдет.

Обработка исключений

Полагание на пользовательский ввод из функции `input` ограничивает вас в контроле этого ввода — он полностью определяется пользователем и может приводить к ошибкам. Скажем, вы написали программу, которая принимает от пользователя два числа и выводит результат деления первого числа на второе.

Python_ex105.py

```
a = input("введите число:")
b = input("введите еще одно:")
a = int(a)
b = int(b)
```

```
5 print(a / b)
```

```
>> введите число:
>> 10
>> введите еще одно:
>> 5
>> 2
```

Программа как будто работает. Однако если пользователь введет `0` в качестве второго числа, вы столкнетесь с проблемой.

Python_ex106.py

```
1 a = input("введите число:")
2 b = input("введите еще одно:")
3 a = int(a)
4 b = int(b)
5 print(a / b)
```

```
>> введите число:
>> 10
>> введите еще одно:
>> 0
>> ZeroDivisionError: integer division or modulo by zero
```

Нельзя просто надеяться, что пользователь не введет `0`. Способом выйти из положения может считаться **обработка исключений** — она позволит вам протестировать программу на возможные ошибки, «перехватить» исключения, если таковые возникают, и решить, что делать дальше.

Для обработки исключений используются ключевые слова `try` и `except`. При использовании обработки исключений, когда пользователь вводит `0` в качестве второго числа, программа может вывести им сообщение о недопустимости ввода `0` вместо генерирования исключения.

Все исключения в Python являются объектами, так что вы можете использовать их в своих программах. Список встроенных исключений можете найти на странице в Интернете по адресу **goo.gl/A2Utav**. Если вам кажется, что ваш код может сгенерировать исключение, используйте составную инструкцию с ключевыми словами `try` и `except`, чтобы перехватить его.

Ветвь `try` содержит возможную ошибку. Ветвь `except` содержит код, который будет выполняться лишь в том случае, если в ветви `try` появится исключение. Ниже приведен пример использования обработки исключений, когда программа не прерывается при вводе 0 в качестве второго числа.

Python_ex107.py

```
1 a = input("введите число:")
2 b = input("введите еще одно:")
```

```
a = int(a)
b = int(b)
try:
    print(a / b)
except ZeroDivisionError:
    print("b не может быть нулем.")
```

```
>> введите число:
>> 10
>> введите еще одно:
>> 0
>> b не может быть нулем.
```

Если пользователь вводит в качестве `b` что-то отличное от `0`, запускается код в блоке `try`, а блок `except` ничего не делает. Если же пользователь вводит `0`, вместо того, чтобы сгенерировать исключение, выполняется код в блоке `except`, и программа выводит текст `b не может быть нулем`.

Ваша программа также прервет работу, если пользователь введет строку, которую Python не может преобразовать в целое число.

Python_ex108.py

```
a = input("введите число:")
b = input("введите еще одно:")
a = int(a)
b = int(b)
try:
    print(a / b)
except ZeroDivisionError:
    print("b не может быть нулем.")
```

```
>> введите число:
>> один
>> введите еще одно:
>> миллион
>> ValueError: invalid literal for int() with base 10: 'один'
```

Это можно исправить, переместив ту часть вашей программы, что принимает ввод, внутрь инструкции `try` и указав инструкции `except` искать два исключения: `ZeroDivisionError` и `ValueError`. Исключение `ValueError` возникает, если вы передаете встроенным функциям `int`, `string`, или `float` неверный ввод. С помощью круглых скобок и запятой между исключениями мы заставляем инструкцию `except` перехватывать два исключения.

Python_ex109.py

```
try:
    a = input("введите число:")
    b = input("введите еще одно:")
    a = int(a)
    b = int(b)
    print(a / b)
except(ZeroDivisionError, ValueError):
    print("Ошибка ввода.")
```

```
>> введите число:
>> один
>> введите еще одно:
>> миллион
>> Ошибка ввода.
```

Не используйте в инструкции `except` переменные, определенные в инструкции `try`, поскольку исключение может возникнуть прежде, чем будет определена переменная, и как только вы попытаетесь использовать инструкцию `except`, внутри нее сгенерируется исключение.

Python_ex110.py

```
try:
    10 / 0
    c = "Я никогда не определюсь."
except ZeroDivisionError:
    print(c)
```

```
>> NameError: name 'c' is not defined
```

Строки документации

Когда вы определяете функцию с параметрами, иногда для ее работы необходимо, чтобы эти параметры были определенного типа данных. Как сообщить об этом всем, кто вызывает вашу функцию? Когда пишете функцию, вверху не помешает оставить комментарий, называемый **строкой документации**, чтобы объяснить, какой тип данных должен иметь каждый параметр. Строки документации объясняют, что делает функция, и какие она требует параметры.

Python_ex111.py

```
def add(x, y):
    """
    Возвращает x + y.
    :параметр x: целое число.
```

```
    :параметр y: целое число.
    :return: целочисленная сумма x и y.
    """
return x + y
```

Первая строка кода строки документации четко объясняет, что делает функция, чтобы когда другие разработчики будут использовать вашу функцию или метод, им не приходилось читать весь ваш код в поисках ее предназначения. Остальные строки определяют параметры функции, типы параметров, и что данная функция возвращает. Это поможет вам программировать быстрее, ведь теперь, чтобы выяснить, что делает функция, достаточно прочесть лишь строки документации, а не читать весь код.

Чтобы примеры в этой книге оставались краткими, я опустил строки документации, которые обычно использую. Когда я пишу код, то включаю строки документации, делая код удобным для всех будущих читателей.

Используйте переменные, только когда это необходимо

Сохраняйте данные в переменной, только если собираетесь затем ее использовать. К примеру, не сохраняйте в переменной целое число, чтобы просто вывести его.

Python_ex112.py

```
x = 100
print(x)
```

```
>> 100
```

Вместо этого передавайте число сразу в функцию вывода.

Python_ex113.py

```
print(100)
```

```
>> 100
```

Во многих примерах в этой книге я нарушаю данное правило, чтобы вам было проще понимать, что я делаю. Вам необязательно делать то же самое со своим кодом.

Словарь терминов

Встроенная функция: функция, поставляющаяся с Python.
Вызов: передача функции вводных данных, необходимые для выполнения указаний и возвращения вывода.
Глобальная область видимости: область видимости переменной, которую можно найти и изменить откуда угодно в программе.
Глобальная переменная: переменная с глобальной областью видимости.

Локальная область видимости: область видимости переменной, которую можно найти и изменить только из функции (или класса), внутри которой переменная была определена.
Область видимости: то, где переменную можно найти и изменить.
Обработка исключений: концепция программирования, позволяющая вам тестировать программу на возможные ошибки, «перехватывать» исключения, если таковые возникают, и решать, что делать дальше.
Обязательный параметр: необходимый параметр.
Опциональный параметр: необязательный параметр.
Параметр: данные, переданные в функцию.
Соглашение: согласованный способ делать что-то.
Строка документации: объясняет, что делает функция, и какие она требует параметры.
Функции: составные инструкции, которые могут принимать данные ввода, выполнять указания и возвращать данные вывода.

Практикум

1. Напишите функцию, которая принимает число в качестве ввода, возводит его в квадрат и возвращает.
2. Создайте функцию, которая принимает строку в качестве параметра и возвращает ее.
3. Напишите функцию, которая принимает три обязательных и два необязательных параметра.
4. Напишите программу с двумя функциями. Первая функция должна принимать в качестве параметра целое число и возвращать результат деления этого числа на 2. Вторая функция должна принимать в качестве параметра целое число и возвращать результат умножения этого числа на 4. Вызовите первую функцию, сохраните результат в переменной и передайте ее в качестве параметра во вторую функцию.
5. Напишите функцию, которая преобразовывает строку в тип данных `float` и возвращает результат. Используйте обработку исключений, чтобы перехватить возможные исключения.
6. Добавьте строку документации ко всем функциям, которые вы написали в заданиях 1–5.

Решения: *chap4_challenge1.py – chap4_challenge6.py.*

Глава 5. Контейнеры

Где дурак удивленно размышляет, мудрец спрашивает.

Бенджамин Дизраэли

В главе 3 вы узнали, как сохранять объекты в переменных, в этой главе вы узнаете, как сохранять их в контейнерах. Контейнеры подобны картотекам — они организовывают хранение ваших данных определенным образом. Вы изучите три наиболее часто используемых контейнера: списки, кортежи и словари.

Методы

В главе 4 вы изучили функции. В Python есть схожее понятие — **методы**. Методы — это функции, тесно связанные с определенным типом данных. Они могут выполнять код и возвращать результат как функции, но, в отличие от последних, методы вызываются с объектами. Вы также можете передавать им параметры. Ниже приведен пример вызова методов `upper` и `replace` у строки.

Python_ex114.py

```
1 "Привет".upper()
```

```
>> 'Привет'
```

Python_ex115.py

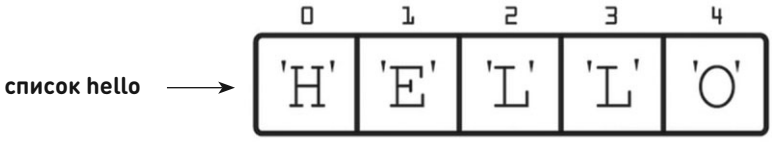
```
1 "Привет".replace("e", "@")
```

```
>> 'Прив@т'
```

Больше информации о методах вы найдете в части II.

Списки

Список — это контейнер, хранящий объекты в определенном порядке.



Для представления списков используются квадратные скобки. Создать список можно двумя способами. Во-первых, пустой список создается при помощи функции `list`.

Python_ex116.py

```
1 fruit = list()
2 fruit
```

```
>> []
```

Или можно просто использовать квадратные скобки.

Python_ex117.py

```
1 fruit = []
2 fruit
```

```
>> []
```

Вы можете создавать списки сразу с элементами в них, используя синтаксис `[]` и помещая внутрь скобок каждый желаемый элемент через запятую.

Python_ex118.py

```
1 fruit = ["Яблоко", "Апельсин", "Персик"]
2 fruit
```

```
>> ['Яблоко', 'Апельсин', 'Персик']
```

В вашем списке три элемента: `"Яблоко"`, `"Апельсин"` и `"Персик"`. Списки хранят элементы в определенном порядке. Если только вы не измените порядок списка, `"Яблоко"` всегда будет первым элементом, `"Апельсин"` — вторым и `"Персик"` — третьим. `"Яблоко"` представляет собой начало списка, а `"Персик"` — конец. Добавление в список нового элемента осуществляется с помощью метода `append`.

Python_ex119.py

```
1 fruit = ["Яблоко", "Апельсин", "Персик"]
2 fruit.append("Банан")
3 fruit.append("Дыня")
4 fruit
```

```
>> ['Яблоко', 'Апельсин', 'Персик', 'Банан', 'Дыня']
```

Каждый объект, который вы передали методу `append`, теперь является элементом в вашем списке. Метод `append` всегда добавляет новый элемент в конец списка.

В списках можно хранить любой тип данных, не только строки.

Python_ex120.py

```
1 random = []
2 random.append(True)
3 random.append(100)
4 random.append(1.1)
5 random.append("Привет")
6 random
```

```
>> [True, 100, 1.1, 'Привет']
```

Строки, списки и кортежи поддерживают **итерирование** (программа может их перебирать, значение за значением), то есть к каждому их элементу можно получить доступ через цикл — такие объекты называются **итерируемыми**. Каждый элемент в итерируемом объекте имеет **индекс** — число, представляющее позицию элемента в этом объекте. Индекс первого элемента в списке — `0`, а не `1`.

В следующем примере элемент `"Яблоко"` записан в списке с индексом `0`, `"Апельсин"` — с индексом `1`, а `"Персик"` — с индексом `2`.

Python_ex121.py

```
1 fruit = ["Яблоко", "Апельсин", "Персик"]
```

Вы можете обратиться к элементу при помощи его индекса, используя синтаксис *имя_списка*[*индекс*].

Python_ex122.py

```
1 fruit = ["Яблоко", "Апельсин", "Персик"]
2 fruit[0]
3 fruit[1]
4 fruit[2]
```

```
>> 'Яблоко'
>> 'Апельсин'
>> 'Персик'
```

Если вы попытаетесь получить доступ к несуществующему индексу, Python сгенерирует исключение.

Python_ex123.py

```
1 colors = ["синий", "зеленый", "желтый"]
2 colors[4]
```

```
>> IndexError: list index out of range
```

Списки **изменяемы**. Когда контейнер является изменяемым, это значит, что вы можете добавлять в него объекты или удалять их. Изменить объект в списке можно, присвоив его индекс новому объекту.

Python_ex124.py

```
1 colors = ["синий", "зеленый", "желтый"]
2 colors
3 colors[2] = "красный"
4 colors
```

```
>> ['синий', 'зеленый', 'желтый']
>> ['синий', 'зеленый', 'красный']
```

С помощью метода pop можно удалить последний элемент в списке.

Python_ex125.py

```
1 colors = ["синий", "зеленый", "желтый"]
2 colors
3 item = colors.pop()
4 item
5 colors
```

```
>> ['синий', 'зеленый', 'желтый']
>> 'желтый'
>> ['синий', 'зеленый']
```

Нельзя использовать `pop` с пустым списком. В этом случае Python сгенерирует исключение.

При помощи оператора сложения можно соединять два списка.

Python_ex126.py

```
1 colors1 = ["синий", "зеленый", "желтый"]
2 colors2 = ["оранжевый", "розовый", "черный"]
3 colors1 + colors2
```

```
>> ['синий', 'зеленый', 'желтый', 'оранжевый', 'розовый',
   'черный']
```

Проверить, есть ли элемент в списке, можно с помощью ключевого слова `in`.

Python_ex127.py

```
1 colors = ["синий", "зеленый", "желтый"]
2 "зеленый" in colors
```

```
>> True
```

Для проверки отсутствия элемента в списке используйте ключевое слово `not`.

Python_ex128.py

```
1 colors = ["синий", "зеленый", "желтый"]
2 "черный" not in colors
```

```
>> True
```

С помощью функции `len` можно узнать длину списка, то есть количество его элементов.

Python_ex129.py

```
1 len(colors)
```

```
>> 3
```

Ниже приведен пример использования списка на практике.

Python_ex130.py

```
1 colors = ["фиолетовый",
2           "оранжевый",
3           "зеленый"]
```

```
guess = input("Угадайте цвет:")

if guess in colors:
    print("Вы угадали!")
else:
    print("Неправильно! Попробуйте еще.")
```

```
>> Угадайте цвет:
```

Список `colors` содержит различные строки, представляющие цвета. При помощи встроенной функции `input` программа предлагает пользователю угадать цвет и сохраняет его ответ в переменной. Если этот ответ содержится в списке `colors`, программа сообщает пользователю, что его догадка была верной. В противном случае, программа предлагает попытаться угадать еще раз.

Кортежи

Кортеж — это контейнер, хранящий объекты в определенном порядке. В отличие от списков, кортежи **неизменяемы**, то есть их содержимое нельзя изменить. Как только вы создали кортеж, значение какого-либо его элемента уже нельзя изменить, как нельзя добавлять и удалять элементы. Кортежи объявляются с помощью круглых скобок. Элементы в кортеже должны быть разделены запятыми. Для создания кортежей используют один из двух вариантов синтаксиса. Первый:

Python_ex131.py

```
my_tuple = tuple()
my_tuple
```

```
>> ()
```

Второй:

Python_ex132.py

```
my_tuple = ()
my_tuple
```

```
>> ()
```

Чтобы добавить в кортеж новые объекты, создайте его вторым способом, указав через запятую каждый желаемый элемент.

Python_ex133.py

```
rndm = ("М. Джексон", 1958, True)
rndm
```

```
>> ('М. Джексон', 1958, True)
```

Даже если кортеж содержит только один элемент, после этого элемента все равно нужно поставить запятую. Таким образом Python отличает кортеж от числа в скобках, определяющих порядок выполнения операций.

Python_ex134.py

```
# это кортеж
("self_taught",)

# это не кортеж
(9) + 1
```

```
>> ('self_taught',)
>> 10
```

После создания кортежа в него нельзя добавлять новые элементы или изменять существующие. При попытке изменить элемент в кортеже после его создания Python сгенерирует исключение.

Python_ex135.py

```
dys = ("1984",
       "О дивный новый мир",
       "451 градус по Фаренгейту")

dys[1] = "Рассказ служанки"
```

```
>> TypeError: 'tuple' object does not support item assignment
```

Получить элементы кортежа можно тем же способом, что и элементы списка указывая индекс элемента.

Python_ex136.py

```
dys = ("1984",
       "О дивный новый мир",
       "451 градус по Фаренгейту")

dys[2]
```

```
>> '451 градус по Фаренгейту'
```

Проверить, содержится ли элемент в кортеже, можно с помощью ключевого слова `in`.

Python_ex137.py

```
dys = ("1984",
```

```
        "О дивный новый мир",
        "451 градус по Фаренгейту")

"1984" in dys
```

```
>> True
```

Поместите перед `in` ключевое слово `not` для проверки отсутствия элемента в кортеже.

Python_ex138.py

```
dys = ("1984",
       "О дивный новый мир",
       "451 градус по Фаренгейту")

"Рассказ служанки" not in dys
```

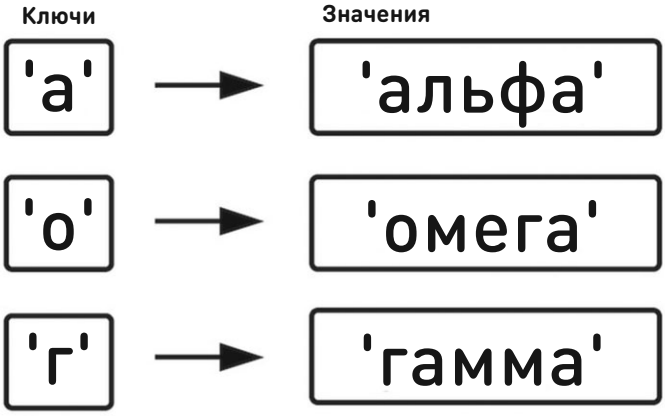
```
>> True
```

Вам, наверное, интересно, зачем использовать структуру данных, которая кажется менее гибкой, чем список. Кортежи удобно использовать, когда вы имеете дело со значениями, которые никогда не изменятся, и вы хотите быть уверенными, что их не изменят другие части вашей программы. Примером данных, которые удобно хранить в кортеже, могут быть географические координаты. Долготу и широту города следует сохранить в кортеже, поскольку эти значения никогда не изменятся, и сохранение в кортеже будет гарантировать, что другие части программы случайно их не изменят. Кортежи, в отличие от списков, могут использоваться в качестве ключей в словарях, о которых пойдет речь в следующем разделе этой главы.

Словари

Словари — еще один встроенный контейнер для хранения объектов. Они используются для связывания одного объекта, называемого **ключом**, с другим, называемым **значением**. Такое связывание называется **отображением**. Результатом будет **пара ключ-значение**. Пары ключ-значение добавляются в словарь. Затем вы можете найти в словаре ключ и получить соответствующее ему значение. Однако нельзя использовать значение для нахождения ключа.

Словари являются изменяемыми, так что в них можно добавлять новые пары ключ-значение. В отличие от списков и кортежей, словари не хранят объекты в определенном порядке. Их полезность заключается в связях между ключами и значениями — существует множество ситуаций, в которых вам нужно будет сохранять данные попарно. Например, в словаре можно сохранить информацию о ком-либо, связав ключ, представляющий рост, со значением роста человека, ключ, представляющий цвет глаз, со значением цвета глаз человека, и ключ, представляющий национальность, с соответствующим значением.



Словари объявляются с помощью фигурных скобок. Для создания словарей существуют два варианта синтаксиса. Первый:

Python_ex139.py

```
my_dict = dict()
my_dict
```

```
>> {}
```

Второй:

Python_ex140.py

```
my_dict = {}
my_dict
```

```
>> {}
```

При создании словарей в них можно добавлять пары ключ-значение. Оба варианта синтаксиса предполагают отделение ключа от значения двоеточием. Пары ключ-значение отделяются запятыми. В отличие от кортежей, если у вас есть только одна пара ключ-значение, запятая после нее не нужна. Ниже показано, как при создании словаря в него добавляются пары ключ-значение.

Python_ex141.py

```
fruits = {"Яблоко":
          "красное",
          "Банан":
          "желтый"}
fruits
```

```
>> {'Яблоко': 'красное', 'Банан': 'желтый'}
```

Ваш вывод может содержать элементы словаря в порядке, отличном от моего, поскольку словари не хранят объекты в определенном порядке – и Python выводит их в произвольном порядке (это касается всех примеров в данном разделе).

Словари изменяемы. Как только вы создали словарь, можете добавлять в него пары ключ-значение, используя синтаксис *имя_словаря[ключ] = значение*, а также искать значение при помощи ключа, используя синтаксис *имя_словаря[ключ]*.

Python_ex142.py

```
facts = dict()

# add a value
facts["код"] = "смешной"
# look up a key
facts["код"]

# add a value
facts["Билл"] = "Гейтс"
# look up a key
facts["Билл"]

# add a value
facts["основание"] = 1776
# look up a key
facts["основание"]
```

```
>> 'смешной'
>> Гейтс
>> 1776
```

Значением в словаре может быть любой объект. В предыдущем примере первые два значения были строками, а последнее значение, `1776`, – целым числом.

В отличие от значения словаря, ключ словаря должен быть неизменяемым. Ключом словаря может быть строка или кортеж, но не список или словарь.

Для определения наличия ключа в словаре используйте ключевое слово `in`. Слово `in` нельзя использовать для проверки наличия в словаре значения.

Python_ex143.py

```
bill = dict({"Билл Гейтс":
             "щедрый"})

"Билл Гейтс" in bill
```

```
>> True
```

Если вы попытаетесь получить доступ к ключу, отсутствующему в словаре, Python сгенерирует исключение.

Чтобы определить отсутствие ключа в словаре, перед `in` добавьте ключевое слово `not`.

Python_ex144.py

```
bill = dict({"Билл Гейтс":
             "щедрый"})

"Билл Дорз" not in bill
```

```
>> True
```

Из словаря можно удалить пару ключ-значение с помощью ключевого слова `del`.

Python_ex145.py

```
books = {"Дракула": "Стокер",
         "1984": "Оруэлл",
         "Процесс": "Кафка"}
del books["Процесс"]

books
```

```
>> {'Дракула': 'Стокер', '1984': 'Оруэлл'}
```

Ниже приведен пример программы, использующей словарь.

Python_ex146.py

```
rhymes = {"1": "смех",
          "2": "синий",
          "3": "я",
          "4": "этаж",
          "5": "жизнь"
}

n = input("введите число:")
if n in rhymes:
    rhyme = rhymes[n]
    print(rhyme)
else:
    print("Не найдено.")
```

```
>> введите число:
```

Ваш словарь (`rhymes`) содержит шесть названий песен (ключей), связанных с шестью музыкантами (значениями). Вы просите пользователя ввести название песни и сохраняете его ответ в переменной. Прежде чем искать этот ответ в словаре, проверьте, существует ли ключ с помощью ключевого слова `in`. Если ключ существует, тогда вы ищете название песни в словаре и выводите имя ее исполнителя. В противном случае пользователю выводится сообщение об отсутствии названия песни.

Контейнеры внутри контейнеров

Вы можете сохранять контейнеры в других контейнерах. Например, можно хранить списки внутри списка.

Python_ex147.py

```
lists = []
rap = ["Баста",
       "Кравц",
       "Злой дух",
       "25-17"]

rock = ["Наутилус Помпилиус",
       "Кино",
       "Ария"]

djs = ["Paul Oakenfold",
       "Tiesto"]

lists.append(rap)
lists.append(rock)
lists.append(djs)

print(lists)
```

```
>> [['Баста', 'Кравц', 'Злой дух', '25-17'], ['Наутилус
   Помпилиус', 'Кино', 'Ария'], ['Paul Oakenfold', 'Tiesto']]
```

В этом примере список `lists` имеет три индекса, каждый из которых является списком: первый индекс — список рэперов, второй — рокеров, третий — диджеев. К этим спискам можно получить доступ с помощью их соответствующих индексов.

Python_ex148.py

```
# Продолжение
# предыдущего примера

rap = lists[0]
print(rap
```

```
>> ['Баста', 'Кравц', 'Злой дух', '25-17']
```

Если вы добавите новый элемент в список rap, то увидите разницу при выводе.

Python_ex149.py

```
# Продолжение
# предыдущего примера

rap = lists[0]
rap.append("Наше дело")
print(rap)
print(lists)
```

```
>> ['Баста', 'Кравц', 'Злой дух', '25-17', 'Наше дело']
>> [['Баста', 'Кравц', 'Злой дух', '25-17', 'Наше дело'],
   ['Наутилус Помпилиус', 'Кино', 'Ария'], ['Paul Oakenfold›',
   'Tiesto']]
```

Можно хранить кортеж внутри списка, список внутри кортежа и словарь внутри списка или кортежа.

Python_ex150.py

```
locations = []

tula = (54.1960, 37.6182)
moscow = (55.7522, 37.6155)

locations.append(tula)
locations.append(moscow)

print(locations)
```

```
>> [(54.196, 37.6182), (55.7522, 37.6155)]
```

Python_ex151.py

```
eights = ["Эдгар Алан По",
          "Чарльз Диккенс"]

nines = ["Хемингуэй",
         "Фицджеральд",
         "Оруэлл"]

authors = (eights, nines)
print(authors)
```

```
>> (['Эдгар Алан По', 'Чарльз Диккенс'], ['Хемингуэй',
   'Фицджеральд', 'Оруэлл'])
```

Python_ex152.py

```
bday = {"Хемингуэй":
        "21.07.1899",
        "Фицджеральд":
        "24.09.1896"}

my_list = [bday]
print(my_list)
my_tuple = (bday,)
print(my_tuple)
```

```
>> [{'Хемингуэй': '21.07.1899', 'Фицджеральд': '24.09.1896'}]
>> ({'Хемингуэй': '21.07.1899', 'Фицджеральд': '24.09.1896'},)
```

Список, кортеж или словарь могут быть значениями в словаре.

Python_ex153.py

```
ru = {"расположение":
       (55.7522,
       37.6155),

      "знаменитости":
       ["Андрей Звягинцев",
        "Юрий Быков",
        "Петр Буслов"],

      "факты":
      {"город":
       "Москва",
       "страна":
       "Россия"}
}
```

В данном примере ваш словарь, `ru`, имеет три ключа: «`расположение`», «`знаменитости`» и «`факты`». Значением первого ключа выступает кортеж, поскольку географические координаты никогда не меняются. Значением второго ключа является список знаменитостей, живущих в Москве, — это список, поскольку данные могут измениться. Значением третьего ключа является словарь, поскольку пары ключ-значение — лучший способ представить факты о Москве.

Словарь терминов

Значение: значение, связанное в словаре с ключом.
Изменяемый: когда контейнер является изменяемым, это значит, что его содержимое можно изменить.
Индекс: число, представляющее позицию элемента в итерируемом объекте.
Итерирование: объект поддерживает итерирование, если к каждому его элементу можно получить доступ через цикл.
Итерируемые объекты: объекты, поддерживающие итерирование, например строки, списки и кортежи.
Ключ: значение, при помощи которого находят значение в словаре.
Метод: функция, тесно связанная с определенным типом данных.
Неизменяемый: если контейнер является изменяемым, это значит, что его содержимое нельзя изменить.
Отображение: связывание одного объекта с другим.
Пара ключ-значение: ключ, связанный в словаре со значением.
Словарь: встроенный контейнер для хранения объектов. Связывает один объект, называемый ключом, с другим объектом — значением.
Список: контейнер, хранящий объекты в определенном порядке.

Практикум

1. Создайте список ваших любимых музыкантов.
2. Создайте список кортежей, где каждый кортеж содержит долготу и широту любого места, в котором вы жили или которое посещали.
3. Создайте словарь, содержащий различные данные о вас: рост, любимый цвет, любимый актер и т.д.
4. Напишите программу, которая запрашивает у пользователя его вес, любимый цвет или актер, и возвращает результат из словаря, созданного в предыдущем задании.
5. Создайте словарь, связывающий ваших любимых музыкантов со списком ваших любимых песен, написанных ими.
6. Списки, кортежи и словари — лишь некоторые из встроенных в Python контейнеров. Самостоятельно изучите множества (тип контейнеров). В каком случае бы вы использовали множество?

Решения: *chap5_challenge1.py – chap5_challenge5.py.*

Глава 6. Операции со строками

В теории между теорией и практикой нет никакой разницы.
Но на практике разница есть.

Ян ван де Снепшойт

Python содержит инструменты для выполнения операций со строками, позволяющий, к примеру, разделять строку на две части по заданному символу или изменять регистр строки. Скажем, если у вас есть строка, написанная ПРОПИС-

НЫМИ БУКВАМИ, и вы хотите изменить и на строчные, это можно сделать в Python. В этой главе вы узнаете больше о строках, а также ознакомитесь с некоторыми наиболее полезными инструментами Python для управления строками.

Тройные строки

Если строка занимает более одной строки кода, нужно поместить эту строку в тройные кавычки.

Python_ex154.py

```
""" строка один
    строка два
    строка три
"""
```

Если вы попытаетесь определить строку, занимающую больше одной строки кода, с помощью одинарных или двойных кавычек, то получите синтаксическую ошибку.

Индексы

Строки, так же как списки и кортежи, итерируемы. Доступ к каждому символу в строке производится при помощи индекса. Как и у остальных итерируемых объектов, первый символ в строке имеет индекс `0`, каждый последующий индекс увеличивается на `1`.

Python_ex155.py

```
author = "Кафка"
author[0]
author[1]
author[2]
author[3]
author[4]
```

```
>> 'К'
>> 'а'
>> 'ф'
>> 'к'
>> 'а'
```

В данном примере для получения доступа к каждому символу в строке «Кафка» использовались индексы `0`, `1`, `2`, `3` и `4`. Если вы попытаетесь извлечь символ, индекс которого больше, чем последний индекс вашей строки, Python сгенерирует исключение.

Python_ex156.py

```
author = "Кафка"
author[5]
```

```
>> IndexError: string index out of range
```

Python также позволяет извлекать элементы из списка с помощью **отрицательного индекса** (должен быть отрицательным числом), то есть индекса, который находит элементы в итерируемом объекте справа налево, а не слева направо. Чтобы получить доступ к последнему элементу в итерируемом объекте, используйте отрицательный индекс `-1`.

Python_ex157.py

```
author = "Кафка"
author[-1]
```

```
>> 'а'
```

Отрицательный индекс `-2` находит предпоследний элемент, отрицательный индекс `-3` — элемент, третий с конца, и так далее.

Python_ex158.py

```
author = "Кафка"
author[-2]
author[-3]
```

```
>> 'к'
>> 'ф'
```

Строки неизменяемы

Как и кортежи, строки неизменяемы. Нельзя изменять символы в строке – если вы хотите это сделать, нужно создавать новую строку.

Python_ex159.py

```
ff = "Ф. Фицджеральд"
ff = "Ф. Скотт Фицджеральд"
ff
```

```
>> 'Ф. Скотт Фицджеральд'
```

В Python для создания новых строк из существующих есть несколько методов, о которых вы узнаете в этой главе.

Конкатенация

С помощью оператора сложения можно соединять две или больше строк. Результатом этой операции будет строка, состоящая из символов первой строки, за ко-

торыми следуют символы из следующей строки (строк). Соединение строк называют конкатенацией.

Python_ex160.py

```
1 ▌ "кот" + "в" + "шляпе"
```

```
>> 'котвшляпе'
```

Python_ex161.py

```
1 ▌ "кот" + " в" + " красной" + " шляпе"
```

```
>> 'кот в красной шляпе'
```

Умножение строк

С помощью оператора умножения строку можно умножать на число.

Python_ex162.py

```
1 ▌ "Сойер" * 3
```

```
>> 'СойерСойерСойер'
```

Изменение регистра

При помощи вызова метода `upper` можно превратить каждую букву в строке в прописную.

Python_ex163.py

```
1 ▌ "Истина где-то рядом...".upper()
```

```
>> 'ИСТИНА ГДЕ-ТО РЯДОМ...'
```

Аналогично каждую букву в строке можно сделать строчной, вызвав в этой строке метод `lower`.

Python_ex164.py

```
1 ▌ "ТАК БУДЕТ ПРОДОЛЖАТЬСЯ.".lower()
```

```
>> 'так будет продолжаться.'
```

Первую букву предложения можно сделать прописной, вызвав метод `capitalize`.

Python_ex165.py

```
1 ▌ "троглодиты...".capitalize()
```

```
>> 'Троглодиты...'
```

Метод `format`

Новую строку можно создать при помощи метода `format`, проверяющего вхождения в строке фигурных скобок `{}` и заменяющего их переданными ему параметрами.

Python_ex166.py

```
1 "Уильям {}".format("Фолкнер")
```

```
>> 'Уильям Фолкнер'
```

В качестве параметра можно также передавать переменную.

Python_ex167.py

```
1 last = "Фолкнер"
2 "Уильям {}".format(last)
```

```
>> 'Уильям Фолкнер'
```

Вы можете использовать в строке фигурные скобки столько раз, сколько пожелаете.

Python_ex168.py

```
1 author = "Уильям Фолкнер"
2 year_born = "1897"

3 "{} родился в {}.".format(author, year_born)
```

```
>> 'Уильям Фолкнер родился в 1897.'
```

Метод `format` может пригодиться, если вы создаете строку из пользовательского ввода.

Python_ex169.py

```
1  n1 = input("Введите существительное:")
2  v = input("Введите глагол:")
3  adj = input("Введите прилагательное:")
4  n2 = input("Введите существительное:")

5  r = """Как обычно, {} {} {} {}
6      """.format(n1,
7                 v,
8                 adj,
9                 n2)
10 print(r)
```

```
>> Введите существительное:
```

Программа предлагает пользователю вести два существительных, глагол и прилагательное, а затем при помощи метода `format` создает из ввода новую строку и выводит ее.

Метод `split`

Для строк существует метод `split`, который используется для разделения одной строки на две или больше строк. В качестве параметра методу `split` передается строка, и он использует эту строку для разделения исходной строки на несколько строк. Например, строку `"Я прыгнул через голову. Это целых 2 метра!"` можно разделить на две отдельные строки, передав методу `split` в качестве параметра точку.

Python_ex170.py

```
1 "Я прыгнул через голову.Это целых 2 метра!".split(".")
```

```
>> ['Я прыгнул через голову', 'Это целых 2 метра!']
```

Результатом операции будет список с двумя элементами: строкой, состоящей из всех символов до точки, и строкой, состоящей из всех символов после точки.

Метод `join`

Метод `join` позволяет добавлять новые символы между всеми символами в строке.

Python_ex171.py

```
1 first_three = "абв"
2 result = "+".join(first_three)
3 result
```

```
>> 'а+б+в'
```

Превратить список строк в единую строку можно, вызвав метод `join` в пустой строке и передав этот список в качестве параметра метода.

Python_ex172.py

```
1 words = ["Рыжая",
2          "лисица",
3          "сделала",
4          "кувырок",
5          "через",
6          "голову",
7          "."]
8 one = "".join(words)
9 one
```

```
>> 'Рыжаялисицасделалакувырокчерезголову.'
```

Если вызвать метод `join` в строке, содержащей лишь пробел, можно создать новую строку, в которой каждое слово отделено пробелом.

Python_ex173.py

```
words = ["Рыжая",
         "лисица",
         "сделала",
         "кувырок",
         "через",
         "голову",
         "."]
one = " ".join(words)
one
```

```
>> 'Рыжая лисица сделала кувырок через голову .'
```

Метод `strip`

Метод `strip` используется для удаления пробельных символов в начале и конце строки.

Python_ex174.py

```
s = "    Москва        "
s = s.strip()
s
```

```
>> 'Москва'
```

Метод `replace`

Метод `replace` заменяет каждое вхождение строки другой строкой. Первый параметр — строка, которую нужно заменить, второй — строка, которой нужно заменить вхождения.

Python_ex175.py

```
equ = "Все животные одинаковы."
equ = equ.replace("о", "@")
print(equ)
```

```
>> Все жив@тные @динак@вы.
```

Поиск индекса

Индекс первого вхождения символа в строке можно найти с помощью метода `index`. Передайте в качестве параметра метода символ, который вы ищете, и метод `index` вернет индекс первого вхождения этого символа в строке.

Python_ex176.py

```
1 "животные".index("н")
```

```
>> 5
```

Если метод `index` не найдет соответствия, Python сгенерирует исключение.

Python_ex177.py

```
1 "животные".index("з")
```

```
>> ValueError: substring not found
```

Если вы не уверены, есть ли в строке искомое соответствие, можете воспользоваться обработкой исключений.

Python_ex178.py

```
1 try:
2     "животные".index("з")
3 except:
4     print("Не обнаружено.")
```

```
>> Не обнаружено.
```

Ключевое слово `in`

Ключевое слово `in` проверяет, содержится ли строка в другой строке, и возвращает значение `True` или `False`.

Python_ex179.py

```
1 "Кот" in "Кот в шляпе."
```

```
>> True
```

Python_ex180.py

```
1 "Мышь" in "Кот в шляпе."
```

```
>> False
```

Поместите ключевое слово `not` перед `in`, чтобы проверить отсутствие строки в другой строке.

Python_ex181.py

```
1 "Поттер" not in "Гарри"
```

```
>> True
```

Управляющие символы

Если вы используете кавычки внутри строки, то получите синтаксическую ошибку.

Python_ex182.py

```
1 # этот код не работает.
2 "Она сказала "Непременно.""
```

```
>> SyntaxError: invalid syntax
```

Эту ошибку можно исправить, поместив перед кавычками символ обратного слеша.

Python_ex183.py

```
1 "Она сказала \"Непременно.\""
```

```
>> 'Она сказала "Непременно."'
```

Python_ex184.py

```
1 'Она сказала \"Непременно.\"'
```

```
>> 'Она сказала "Непременно."'
```

Управляющие символы сообщают Python, что знак, перед которым они помещены (в нашем случае, кавычки), не имеет специального значения, а предназначен для представления обычного символа. Для этого Python использует обратный слеш.

Не нужно указывать управляющие символы перед одинарными кавычками в строке с двойными кавычками.

Python_ex185.py

```
1 "Она сказала 'Непременно.'"
```

```
>> "Она сказала 'Непременно.'"
```

Также можно поместить двойные кавычки внутри одинарных — это проще, чем указывать управляющие символы перед двойными кавычками.

Python_ex186.py

```
1 'Она сказала "Непременно."'
```

```
>> 'Она сказала "Непременно."'
```

Новая строка

Помещение символов `\n` внутрь строки выполняет перенос строки.

Python_ex187.py

```
1 print("строка1\nстрока2\nстрока3")
```

```
>> строка1
>> строка2
>> строка3
```

Извлечение среза

Извлечение среза — это способ вернуть новый итерируемый объект, состоящий из подмножества элементов другого итерируемого объекта. Синтаксис для извлечения среза следующий: *итерируемый_объект[начальный_индекс:конечный_индекс]*. Начальный индекс указывает на начало среза, конечный — на конец среза.

Ниже показано, как извлечь срез списка.

Python_ex188.py

```
1 fict = ["Толстой",
2         "Дик",
3         "Оруэлл",
4         "Пелевин",
5         "Остин"]
6 fict[0:3]
```

```
>> ['Толстой', 'Дик', 'Оруэлл']
```

При извлечении среза начальный индекс указывает на элемент под этим индексом, но конечный индекс указывает на элемент перед соответствующим индексом. Вследствие этого, если вы хотите получить срез от "Толстой" (индекс 0) до "Оруэлл" (индекс 2), нужно извлекать срез от индекса 0 до индекса 3.

Ниже приведен пример извлечения среза строки.

Python_ex189.py

```
1 ivan = """Петр Иванович успокоился и с интересом стал
  расспрашивать подробности о кончине Ивана Ильича."""

2 ivan[0:24]
3 ivan[24:93]
```

```
>> 'Петр Иванович успокоился'
>> ' и с интересом стал расспрашивать подробности о кончине
   Ивана Ильича.'
```

Если ваш начальный индекс — 0, тогда можете оставить его пустым.

Python_ex190.py

```
1 ivan = """Петр Иванович успокоился и с интересом стал рас-
  спрашивать подробности о кончине Ивана Ильича."""

2 ivan[:24]
```

```
>> 'Петр Иванович успокоился'
```

Если ваш конечный индекс является последним индексом в итерируемом объекте, можете также оставить его пустым.

Python_ex191.py

```
1 ivan = """Петр Иванович успокоился и с интересом стал
  расспрашивать подробности о кончине Ивана Ильича."""

2 ivan[24:]
```

```
>> ' и с интересом стал расспрашивать подробности о кончине
   Ивана Ильича.'
```

Если вы оставите пустыми и начальный, и конечный индексы, то после извлечения среза получите исходный объект.

Python_ex192.py

```
1 ivan = """Петр Иванович успокоился и с интересом стал рас-
  спрашивать подробности о кончине Ивана Ильича."""

2 ivan[:]
```

```
>> 'Петр Иванович успокоился и с интересом стал расспрашивать
   подробности о кончине Ивана Ильича.'
```

Словарь терминов

Извлечение среза: способ вернуть новый итерируемый объект, состоящий из подмножества элементов другого итерируемого объекта.
Конечный индекс: индекс, на котором заканчивается извлечение среза.
Начальный индекс: индекс, с которого начинается извлечение среза.
Отрицательный индекс: индекс (должен быть отрицательным числом), который используется для нахождения элементов в итерируемом объекте справа налево, а не слева направо.
Управляющие символы: сообщают Python, что знак, перед которым они помещены, в данном случае не имеет специального значения, а предназначен для представления обычного символа.

Практикум

1. Выведите каждый символ в строке «Чехов».
2. Напишите программу, которая принимает от пользователя две строки, вставляет их в строку `"Вчера я написал первый_ответ. Вчера я ходил в второй_ответ!"` и выводит новую строку.
3. Используйте метод, чтобы исправить грамматическую ошибку в строке `"олдос Хаксли родился в 1894 году."`, сделав первую букву в первом слове предложения прописной.
4. Вызовите метод, который превращает строку `"Где это? Кто это? Когда это?"` в список `["Где это?", "Кто это?", "Когда это?"]`.

5. Превратите список `["Рыжая", "лиса", "перепрыгнула", "через", "низкий", "забор", "."]` в грамматически правильное предложение. Каждое слово должно отделяться пробелом, но между словом «забор» и следующей за ним точкой пробела быть не должно. (Не забывайте, вы выучили метод, превращающий список строк в единую строку.)
6. Замените каждое вхождение буквы `"о"` в строке `"Ребенок — зеркало поступков родителей."` цифрой 0.
7. Используйте метод, чтобы определить индекс символа `"м"` в строке `"Хемингуэй"`.
8. Найдите в своей любимой книге диалог (с кавычками) и превратите его в строку.
9. Создайте строку «тритритри», используя конкатенацию, а затем сделайте то же самое, только с помощью умножения.
10. Извлеките срез строки «И незачем так орать! Я и в первый раз прекрасно слышал.» так, чтобы она содержала только символы до восклицательного знака.

Решения: *chap6_challenge1.py – chap6_challenge10.py*.

Глава 7. Циклы

Восемьдесят процентов успеха — это просто прийти.

Вуди Аллен

Одна из программ, представленных в этой книге, сто раз выводила пользователю строку `Привет, мир!`. Это осуществлялось при помощи цикла — фрагмента кода, непрерывно выполняющего инструкции, пока удовлетворено определенное в коде условие. В этой главе вы узнаете о циклах и о том, как их использовать.

Циклы `for`

В этом разделе вы узнаете, как использовать **`for`** — цикл, перебирающий итерируемый объект. Процесс перебора называется **итерированием**. Цикл `for` можно использовать, чтобы определять инструкции, которые будут выполняться один раз для каждого элемента в итерируемом объекте, и с помощью таких инструкций вы можете получать доступ ко всем этим элементам и осуществлять операции с ними. Например, с помощью цикла `for`, выполняющего перебор списка строк, и метода `upper` можно сделать символы каждой строки прописными.

Цикл `for` определяется синтаксисом `for имя_переменной in имя_итерируемого_объекта: инструкции`, где `имя_переменной` – выбранное вами имя переменной, которая назначается значению каждого элемента в итерируемом объекте, а `инструкции` — код, который выполняется при каждом прохождении цикла. Ниже приведен пример использования цикла `for` для перебора (итерирования) символов строки.

Python_ex193.py

```
1 name = "Тед"
2 for character in name:
3     print(character)
```

```
>> T
>> e
>> д
```

При каждом прохождении цикла переменная `character` назначается элементу итерируемого объекта `name`. При первом прохождении выводится буква `T`, поскольку переменная `character` назначена первому элементу объекта `name`. При втором прохождении выводится буква `e`, ведь `character` назначена второму элементу `name`. Процесс продолжается до тех пор, пока каждый элемент в итерируемом объекте не будет назначен переменной `character`.

Ниже приведен пример использования цикла `for` для перебора элементов списка.

Python_ex194.py

```
1 shows = ["Во все тяжкие",
2          "Секретные материалы",
3          "Фарго"]
4 for show in shows:
5      print(show)
```

```
>> Во все тяжкие
>> Секретные материалы
>> Фарго
```

Пример использования цикла `for` для итерирования элементов кортежа.

Python_ex195.py

```
1 coms = ("Теория большого взрыва",
2         "Друзья",
3         "Папины дочки")
4 for show in coms:
5 print(show)
```

```
>> Теория большого взрыва
>> Друзья
>> Папины дочки
```

Пример использования цикла `for` для перебора ключей в словаре.

Python_ex196.py

```
people = {"Джим Парсонс":
          "Теория большого взрыва",
          "Брайан Крэнстон":
          "Во все тяжкие",
          "Екатерина Старшова":
          "Папины дочки"
}

for character in people:
    print(character)
```

```
>> Джим Парсонс
>> Брайан Крэнстон
>> Екатерина Старшова
```

При помощи цикла `for` можно изменять элементы в изменяемом итерируемом объекте, например списке.

Python_ex197.py

```
tv = ["Во все тяжкие",
      "Секретные материалы",
      "Фарго "]
i = 0
for show in tv:
    new = tv[i]
    new = new.upper()
    tv[i] = new
    i += 1

print(tv)
```

```
>> ['ВО ВСЕ ТЯЖКИЕ', 'СЕКРЕТНЫЕ МАТЕРИАЛЫ', 'ФАРГО']
```

В данном примере цикл `for` использовался для перебора списка `tv`. Вы отслеживаете текущий элемент в списке с помощью **переменной индекса** — переменной, хранящей целое число, которое представляет индекс в итерируемом объекте. Значение переменной индекса `i` начинается с `0` и увеличивается при каждом прохождении цикла. Вы используете переменную индекса, чтобы получить текущий элемент списка, который затем сохраняете в переменной `new`. После этого вы вызываете метод `upper` в переменной `new`, сохраняете результат и используете свою переменную индекса, чтобы заменить этим результатом текущий элемент в списке. Наконец, вы увеличиваете `i`, чтобы при следующем прохождении цикла взять следующий элемент в списке.

Поскольку получение доступа к каждому элементу в итерируемом объекте и его индексу — распространенная задача, у Python для этого есть специальный синтаксис.

Python_ex198.py

```
tv = ["Во все тяжкие", "Секретные материалы",
      "Фарго"]
for i, show in enumerate(tv):
    new = tv[i]
    new = new.upper()
    tv[i] = new

print(tv)
```

```
>> ['ВО ВСЕ ТЯЖКИЕ', 'СЕКРЕТНЫЕ МАТЕРИАЛЫ', 'ФАРГО']
```

Вместо перебора списка `tv` вы передали список `tv` в `enumerate` и выполнили перебор результата, что позволило ввести новую переменную `i`, отслеживающую текущий индекс.

Циклы `for` можно использовать для перемещения данных между изменяемыми итерируемыми объектами. Например, вы можете использовать два цикла `for`, чтобы взять все строки из двух разных списков, сделать прописными все символы в этих строках и поместить измененные строки в новый список.

Python_ex199.py

```
tv = ["Во все тяжкие", "Секретные материалы",
      "Фарго"]
coms = ["Теория большого взрыва",
        "Друзья",
        "Папины дочки"]
all_shows = []

for show in tv:
    show = show.upper()
    all_shows.append(show)

for show in coms:
    show = show.upper()
    all_shows.append(show)

print(all_shows)
```

```
>> ['ВО ВСЕ ТЯЖКИЕ', 'СЕКРЕТНЫЕ МАТЕРИАЛЫ', 'ФАРГО', 'ТЕОРИЯ
   БОЛЬШОГО ВЗРЫВА', 'ДРУЗЬЯ', 'ПАПИНЫ ДОЧКИ']
```

В этом примере присутствуют три списка: `tv`, `coms` и `all_shows`. Первый цикл перебирает все элементы списка `tv`, при этом вы используете метод `upper`, чтобы сделать буквы прописными, и метод `append`, чтобы добавить их все в список `all_shows`. Во втором цикле вы делаете то же самое со списком `coms`. Когда выводится список `all_shows`, он содержит все элементы обоих списков, и каждый из этих элементов представляет собой слова прописными буквами.

Функция `range`

Можно использовать встроенную функцию `range`, чтобы создать последовательность целых чисел и цикл `for`, чтобы выполнить ее перебор. Функция `range` принимает два параметра: число, с которого последовательность начинается, и число, на котором она заканчивается. Последовательность целых чисел, возвращаемая функцией `range`, включает в себя первый параметр (число, с которого нужно начать), но не включает второй (число, на котором нужно закончить). Ниже приведен пример использования функции `range` для создания последовательности чисел и их перебора.

Python_ex200.py

```
1 for i in range(1, 11):
2     print(i)
```

```
>> 1
...
>> 9
>> 10
```

В этом примере вы использовали цикл `for`, чтобы вывести все числа в итерируемом объекте, которые возвращаются функцией `range`. Программисты часто присваивают переменной, используемой для перебора списка целых чисел, имя `i`.

Циклы `while`

В данном разделе вы узнаете, как использовать **цикл `while`**, выполняющий код до тех пор, пока выражение принимает истинно (`True`). Синтаксис цикла `while` следующий: `while выражение:` *`код_для_выполнения`*. *`Выражение`* определяет, будет цикл продолжаться или нет, а *`код_для_выполнения`* должен выполняться циклом.

Python_ex201.py

```
1 x = 10
2 while x > 0:
3     print('{}'.format(x))
4     x -= 1
5 print("С Новым годом!")
```

```
>> 10
>> 9
>> 8
>> 7
>> 6
>> 5
>> 4
>> 3
>> 2
>> 1
>> С Новым годом!
```

Ваш цикл `while` выполняет свой код до тех пор, пока выражение, определенное в его заголовке, `x > 0`, истинно — принимает значение `True`. При первом прохождении цикла `x` равен `10`, и выражение `x > 0` принимает значение `True`. Цикл `while` выводит значение `x`, затем уменьшает `x` на `1`. Теперь `x` равен `9`. В следующий раз снова выводится `x`, и он уменьшается до `8`. Этот процесс продолжается, пока `x` не уменьшится до `0`, здесь выражение `x > 0` примет значение `False`, и тогда ваш цикл завершится. Затем Python выполнит следующую за циклом строку кода и выведет строку `С Новым годом!`.

Если вы определите цикл `while` с помощью выражения, всегда принимающего значение `True`, ваш цикл будет выполняться вечно. Цикл, который никогда не завершается, называется **бесконечным циклом**. Ниже приведен пример бесконечного цикла (будьте готовы нажать на клавиатуре сочетание клавиш **Ctrl+C**, чтобы прекратить выполнение цикла).

Python_ex202.py

```
1 while True:
2     print("Привет, мир!")
```

```
>> Привет, мир!
...
```

Поскольку цикл `while` выполняется, пока выражение, определенное в его заголовке, истинно, — а выражение `True` истинно всегда, — этот цикл будет выполняться вечно.

Инструкция `break`

Вы можете использовать **инструкцию `break`** — инструкцию с ключевым словом `break`, — чтобы прекратить цикл. Следующий цикл выполнится сто раз.

Python_ex203.py

```
1 for i in range(0, 100):
2     print(i)
```

```
>> 0
>> 1
...
```

Если вы добавите инструкцию `break`, цикл выполнится лишь один раз.

Python_ex204.py

```
for i in range(0, 100):
    print(i)
    break
```

```
>> 0
```

Как только Python сталкивается с инструкцией `break`, цикл завершается. Вы можете использовать цикл `while` и ключевое слово `break` для написания программы, которая будет просить пользователя ввести данные, пока он не введет букву `X`, чтобы выйти.

Python_ex205.py

```
qs = ["Как тебя зовут?",
      "Твой любимый цвет?",
      "Что ты делаешь?"]
n = 0
while True:
    print("Введи X для выхода")
    a = input(qs[n])
    if a == "X":
        break
    n = (n + 1) % 3
```

```
>> Введи X для выхода
>> Как тебя зовут?
```

При каждом прохождении цикла ваша программа задает пользователю один из вопросов из списка `qs`.

`n` — переменная индекса. При каждом прохождении цикла вы присваиваете `n` значение выражения `(n + 1) % 3`, что позволяет бесконечно проходить по каждому вопросу в списке `qs`. При первом прохождении цикла `n` начинает с `0`. Затем `n` присваивается значение выражения `(0 + 1) % 3`, которое равно `1`. После чего `n` присваивается значение `(1 + 1) % 3`, которое равно `2`, ведь всегда, когда первое число в выражении с оператором деления по модулю меньше второго, ответом является это первое число. Наконец, `n` присваивается значение `(2 + 1) % 3`, равное `0`, как и в начале.

Инструкция `continue`

Вы можете использовать **инструкцию `continue`**, чтобы прервать текущую итерацию цикла и продолжить со следующей итерации. Скажем, вам нужно вывести все числа от `1` до `5`, кроме числа `3`. Вы можете это осуществить, используя цикл `for` и инструкцию `continue`.

Python_ex206.py

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

```
>> 1
>> 2
>> 4
>> 5
```

В этом цикле, когда переменная `i` принимает значение `3`, выполняется инструкция `continue` — тогда вместо того, чтобы полностью завершиться, как в случае с ключевым словом `break`, цикл продолжает работать. Он переходит к следующей итерации, пропуская код, который должен был выполниться. Когда переменная `i` принимает значение `3`, Python выполняет инструкцию `continue`, а не выводит число `3`.

Аналогичного результата можно достичь при помощи цикла `while` и инструкции `continue`.

Python_ex207.py

```
i = 1
while i <= 5:
    if i == 3:
        i += 1
        continue
    print(i)
    i += 1
```

```
>> 1
>> 2
>> 4
>> 5
```

Вложенные циклы

Вы можете различными способами комбинировать циклы. Например, можно поместить один цикл в другой, или создать цикл внутри цикла внутри цикла. Нет никаких ограничений по количеству циклов, которые можно помещать внутрь

других циклов, хотя эти ограничения важны. Когда цикл находится внутри другого цикла, второй цикл является вложенным в первый. В этом случае цикл, содержащий внутри другой цикл, называется **внешним**, а вложенный цикл — **внутренним**. Когда у вас есть вложенный цикл, внутренний цикл выполняет перебор своего итерируемого объекта один раз за итерацию внешнего цикла.

Python_ex208.py

```
1 for i in range(1, 3):
2     print(i)
3     for letter in ["а", "б", "в"]:
4         print(letter)
```

```
>> 1
>> а
>> б
>> в
>> 2
>> а
>> б
>> в
```

Вложенный цикл `for` будет перебирать список `["а", "б", "в"]` столько раз, сколько раз выполняется внешний цикл — в нашем случае дважды. Если бы вы сделали так, чтобы внешний цикл выполнялся три раза, то и внутренний цикл также перебирал бы свой список трижды.

Вы можете использовать циклы `for` для прибавления каждого числа из одного списка к каждому числу из другого списка.

Python_ex209.py

```
1 list1 = [1, 2, 3, 4]
2 list2 = [5, 6, 7, 8]
3 added = []
4 for i in list1:
5     for j in list2:
6         added.append(i + j)
7 print(added)
```

```
>> [6, 7, 8, 9, 7, 8, 9, 10, 8, 9, 10, 11, 9, 10, 11, 12]
```

Первый цикл выполняет итерирование каждого целого числа в списке `list1`. Для каждого элемента в этом списке второй цикл перебирает каждое целое число в собственном итерируемом объекте, затем прибавляет его к числу из `list1` и добавляет результат в список `added`. Во втором цикле `for` я назвал переменную `j`, поскольку имя `i` уже было занято в первом цикле.

Вы можете вкладывать цикл `for` внутрь цикла `while` и наоборот.

Python_ex210.py

```
while input('д или н?') != 'н':
    for i in range(1, 6):
        print(i)
```

```
>> д или н?д
>> 1
>> 2
>> 3
>> 4
>> 5
>> д или н?д
>> 1
>> 2
>> 3
>> 4
>> 5
>> д или н?н
```

Программа будет выводить числа от `1` до `5`, пока пользователь не введет `н`.

Словарь терминов

Бесконечный цикл: цикл, который никогда не завершается.
Внешний цикл: цикл, содержащий вложенный цикл.
Внутренний цикл: цикл, вложенный в другой цикл.
Инструкция `break`: инструкция с ключевым словом `break`, использующаяся для прекращения цикла.
Инструкция `continue`: инструкция с ключевым словом `continue`, использующаяся, чтобы прервать текущую итерацию цикла и продолжить со следующей итерации.
Итерирование (перебор): использование цикла для получения доступа к каждому элементу итерируемого объекта.
Переменная индекса: переменная, хранящая целое число, которое представляет индекс в итерируемом объекте.
Цикл `for`: цикл, перебирающий итерируемый объект — например, строку, список, кортеж или словарь.
Цикл `while`: цикл, выполняющий код до тех пор, пока выражение принимает значение `True`.
Цикл: фрагмент кода, непрерывно выполняющий инструкции, пока удовлетворено определенное в коде условие.

Практикум

1. Выведите каждый элемент в следующем списке: `["Ходячие мертвецы", "Красавцы", "Клан Сопрано", "Дневники вампира"]`.
2. Выведите все числа от `25` до `50`.
3. Выведите каждый элемент в списке из первого задания вместе с индексами.
4. Напишите программу с бесконечным циклом (с возможностью ввести букву `X`, чтобы выйти) и списком чисел. При каждом переборе цикла предлагайте пользователю отгадать число из списка и сообщайте, правильно ли он отгадал.
5. Умножьте все числа в списке `[8, 19, 148, 4]` на все числа в списке `[9, 1, 33, 83]` и поместите результаты в третий список.

Решения: chap7_challenge1.py — chap7_challenge5.py.

Глава 8. Модули

Упорство и сила духа творили чудеса во все времена.

Джордж Вашингтон

Представьте, что вы написали программу размером в 10 000 строк кода. Если бы вы поместили весь код в один файл, в нем было бы сложно разобраться. Каждый раз при возникновении ошибки или исключения вам пришлось бы пролистывать 10 000 строк кода в поисках одной-единственной проблемной строки. Программисты выходят из ситуации путем разделения огромных программ на **модули**, — другое название файлов с кодом на языке Python — которые содержат отдельные фрагменты кода. Python позволяет использовать код из одного модуля в другом модуле. В Python также есть **встроенные модули**, содержащие важную функциональность. В этой главе вы узнаете о модулях и о том, как их использовать.

Импорт встроенных модулей

Чтобы использовать модуль, его сначала нужно **импортировать**, то есть написать код, который сообщит Python, где искать модуль. Импортировать модуль можно командой с синтаксисом `import` *`имя_модуля`*. Замените значение *`имя_модуля`* именем модуля, который вы импортируете. Как только вы выполнили импорт модуля, вы можете использовать его переменные и функции.

У Python есть много различных модулей, включая модуль `math`, предоставляющий математический функционал. Полный перечень встроенных модулей Python можно найти на странице **docs.python.org/3/py-modindex.html**. Ниже показано, как импортируется модуль `math`.

Python_ex211.py

```
1 import math
```

Как только вы импортировали модуль, можно использовать его код при помощи синтаксиса *`имя_модуля.код`*, указав *`имя_модуля`*, который вы импор-

тировали, и код — имя желаемой функции или переменной из модуля. Ниже приведен пример импорта и использования функции pow из модуля math, принимающей два параметра, x и y, и возводящей x в степень y.

Python_ex212.py

```
1 import math
2 math.pow(2, 3)
```

```
>> 8.0
```

Для начала, первой строкой импортируйте модуль math. Все модули следует импортировать в верхней части файла, чтобы их было проще отслеживать. Затем вызовите функцию pow с помощью инструкции math.pow(2, 3). Функция вернет значение 8.0.

random — еще один встроенный модуль. Вы можете использовать его функцию randint для создания случайного числа: передайте функции два целых числа, и она вернет выбранное случайным образом целое число в промежутке между ними.

Python_ex213.py

```
1 # В вашем случае может быть не 52,
2 # при запуске число случайно!

3 import random

4 random.randint(0,100)
```

```
>> 52
```

Также можно использовать встроенный модуль statistics, чтобы подсчитать среднее значение, медиану и моду в итерируемом объекте, состоящем из чисел.

Python_ex214.py

```
1 import statistics

2 # среднее
3 nums = [1, 5, 33, 12, 46, 33, 2]
4 statistics.mean(nums)

5 # медиана
6 statistics.median(nums)

7 # мода
8 statistics.mode(nums)
```

```
>> 18.857142857142858
>> 12
>> 33
```

Встроенный модуль `keyword` позволяет проверить, является ли строка ключевым словом в Python.

Python_ex215.py

```
1 import keyword

2 keyword.iskeyword("for")
3 keyword.iskeyword("football")
```

```
>> True
>> False
```

Импорт других модулей

В этом разделе вы создадите модуль, импортируете его в другой модуль и используете его код. Сначала создайте папку *tstp*. В этой папке создайте файл *hello.py*. Введите в созданный файл указанный ниже код и сохраните файл.

Python_ex216.py

```
1 def print_hello():
2     print("Привет")
```

В папке *tstp* создайте еще один файл, *project.py*. Введите в файл *project.py* указанный ниже код и сохраните файл.

Python_ex217.py

```
1 import hello

2 hello.print_hello()
```

```
>> Привет
```

В данном примере вы указали ключевое слово `import`, чтобы использовать код из вашего первого модуля во втором модуле.

Когда вы импортируете модуль, выполняется весь код в нем. При помощи следующего кода создайте модуль *module1.py*.

Python_ex218.py

```
1 # код в module1
2 print("Привет!")
```

```
>> Привет!
```

Код из модуля *module1.py* будет выполнен, когда вы импортируете этот модуль в другой, *module2.py*.

Python_ex219.py

```
1 # код в module2
2 import hello
```

```
>> Привет!
```

Подобное поведение может доставлять неудобства. К примеру, в вашем модуле может быть тестовый код, который вы бы не хотели выполнять при импорте модуля. Эта проблема решается путем добавления инструкции `if __name__ == "__main__"`. Так, код в модуле *module1.py* из предыдущего примера можно изменить следующим образом.

Python_ex220.py

```
1 # код в module1
2 if __name__ == "__main__":
3     print("Привет!")
```

```
>> Привет!
```

Когда вы запускаете эту программу, вывод остается прежним. Но когда вы осуществляете импорт из модуля *module2.py*, код из модуля *module1.py* больше не выполняется, и слово `Привет!` не выводится.

Python_ex221.py

```
1 # код в module2
2 import hello
```

Словарь терминов

Встроенные модули: модули, которые поставляются в составе Python и содержат важную функциональность.
Импортирование модуля: строка кода, которая сообщает Python, где искать нужный модуль.
Модуль: другое название файла с кодом на языке Python.

Практикум

1. Вызовите какую-нибудь другую функцию из модуля `statistics`.
2. Создайте модуль `cubed`, содержащий функцию, которая принимает в качестве параметра число, возводит это число в куб и возвращает его. Импортируйте и вызовите функцию из другого модуля.

Решения: *chap8_challenge1.py и chap8_challenge2.py*.

Глава 9. Файлы

Я твердо убежден, что самообразование —
это единственно возможное образование.

Айзек Азимов

Вы можете работать с файлами, используя Python. Например, с помощью Python можно считывать данные из файла и записывать данные в файл. **Чтение** данных из файла означает получение доступа к этим данным. **Запись** данных в файл означает добавление или изменение данных файла. В этой главе вы познакомитесь с основами работы с файлами.

Запись в файлы

Первый шаг в работе с файлом — открыть его с помощью встроенной в Python функции `open`. Эта функция принимает два параметра: строку, представляющую путь к нужному файлу, и строку, определяющую режим, в котором нужно открыть этот файл.

Путь к файлу представляет собой место на компьютере, в котором находится файл. К примеру, строка `/Users/bob/st.txt` — это путь к файлу *st.txt*. Каждое слово перед именем файла, отделенное слешем, указывает на имя папки, а все это вместе представляет путь к файлу. Если путь к файлу состоит лишь из имени файла (и нет никаких папок, отделенных слешами), Python будет искать этот файл в той папке, откуда вы запустили свою программу. Нельзя прописывать путь к файлу самостоятельно. Unix-подобные операционные системы и Windows используют в путях к файлам разное количество слешей. Чтобы избежать проблем с работой вашей программы в разных операционных системах, пути к файлам всегда нужно создавать с помощью встроенного модуля Python `os module`. Функция `path` этого модуля принимает в качестве параметра каждую папку из пути к файлу и выстраивает вам правильный путь к файлу.

Python_ex222.py

```
1 import os
2 os.path.join("Users",
3               "bob",
4               "st.txt")
```

```
>> 'Users/bob/st.txt'
```

Создание путей к файлу при помощи функции `path` гарантирует, что файлы будут работать в любой операционной системе. Но работа с путями к файлам все еще может вызывать трудности. Если у вас возникли проблемы, посетите сайт **theselftaughtprogrammer.io/filepaths** для получения дополнительной информации.

Режим, который вы передаете функции `open`, определяет действия, которые можно будет совершать с открываемым файлом. Ниже представлено несколько возможных режимов открытия файла:

- `"r"` — открывает файл только для чтения.
- `"w"` — открывает файл только для записи. Удаляет содержимое файла, если файл существует; если файл не существует, создает новый файл для записи.
- `"w+"` — открывает файл для чтения и записи. Удаляет содержимое файла, если файл существует; если файл не существует, создает новый файл для чтения и записи[6].

Функция `open` возвращает так называемый **файловый объект**, который используется для чтения и/или записи в ваш файл. Когда вы используете режим `"w"`, функция `open` создает новый файл (если он еще не существует) в каталоге, где работает ваша программа.

Затем вы можете использовать метод `write` на файловом объекте, чтобы осуществить запись в файл, и метод `close`, чтобы закрыть его. Если вы откроете файл при помощи метода `open`, закрыть его нужно при помощи метода `close`. Если вы используете метод `open` на нескольких файлах и забудете закрыть их, это может вызвать проблемы с программой. Ниже приведен пример открытия файла, записи в него и закрытия.

Python_ex223.py

```
1 st = open("st.txt", "w")
2 st.write("привет от Python!")
3 st.close()
```

В данном примере вы используете функцию `open`, чтобы открыть файл, и сохраняете файловый объект, возвращаемый ей, в переменной `st`. Затем вы вызываете метод `write` на переменной `st`, который принимает строку как параметр и записывает ее в новый файл, созданный Python. Наконец, вы закрываете файл, вызывая метод `close` на файловом объекте.

Автоматическое закрытие файлов

Есть также второй вариант синтаксиса для открытия файлов, с ним вам не нужно держать в памяти необходимость закрыть файлы. Чтобы использовать этот синтаксис, поместите весь код, которому требуется доступ к файловому объекту, внутрь **`with`** — составной инструкции, автоматически выполняющей действие после того, как Python проходит ее.

Синтаксис для открытия файла с помощью инструкции `with` следующий: `with open(`*`путь_к_файлу, режим`*`) as `*`имя_переменной: ваш_код`*. Значение *`путь_к_файлу`* представляет путь к вашему файлу, затем указывается `режим`, в котором нужно открыть файл, *`имя_переменной`*, которой назначен файловый объект, а значение *`ваш_код`* обозначает код, у которого есть доступ к этой переменной.

Когда вы используете этот синтаксис для открытия файла, файл автоматически закрывается после того, как выполняется последняя строка тела `ваш_код`. Ниже приведен пример из предыдущего раздела, использующий для открытия, записи и закрытия файла этот синтаксис.

[6] www.tutorialspoint.com/python/python_files_io.htm

Python_ex224.py

```
1 with open("st.txt", "w") as f:
2     f.write("привет от Python!")
```

Пока интерпретатор находится внутри инструкции `with`, вы можете производить доступ к файловому объекту (в этом случае, `f`). Как только Python завершает выполнение всего кода в инструкции `with`, Python закрывает файл автоматически.

Чтение из файлов

Если вы хотите прочесть данные из файла, то передаете `"r"` в качестве второго параметра в `open`. Затем вы вызываете метод `read` в своем файловом объекте, что возвращает итерируемый объект со всеми строками файла.

Python_ex225.py

```
1 # убедитесь, что этот файл
2 # создается из предыдущего
3 # примера

4 with open("st.txt", "r") as f:
5     print(f.read())
```

```
>> привет от Python!
```

Вызвать `read`, не закрывая и не открывая файл заново, можно лишь один раз, так что если вам понадобится содержимое файла позже, вы должны сохранить его в переменной или контейнере. Ниже показано, как сохранять содержимое файла из предыдущего примера в списке.

Python_ex226.py

```
1 my_list = list()

2 with open("st.txt", "r") as f:
3     my_list.append(f.read())

4 print(my_list)
```

```
>> ['привет от Python!']
```

Теперь позже в программе вы сможете получить доступ к этим данным.

CSV-файлы

Python содержит встроенный модуль, позволяющий работать с **CSV-файлами**. CSV-файл имеет расширение *.csv* и содержит данные, разделенные с помощью запятых (CSV расшифровывается как Comma Separated Values — значения, раз-

деленные запятыми). Программы типа Excel, обрабатывающие электронные таблицы, часто используют CSV-файлы. Каждый фрагмент данных, отделенный запятой в CSV-файле, представляет собой ячейку в электронной таблице, а каждая строка файла — строку в таблице. **Разделителем** выступает символ (например, запятая или вертикальная черта |), используемый для разделения данных в CSV-файле. Ниже показано содержимое CSV-файла *self_taught.csv*.

```
один,два,три,четыре,пять,шесть
```

Если бы вы загрузили этот файл в Excel, тогда `один`, `два` и `три` заняли бы по ячейке в первой строке электронной таблицы, а `четыре`, `пять` и `шесть` — по ячейке во второй.

CSV-файл можно открыть с помощью инструкции `with`, но внутри нее нужно использовать модуль `csv`, чтобы конвертировать файловый объект в объект `csv`. У модуля `csv` есть метод `writer`, который принимает файловый объект и разделитель. Метод `writer` возвращает объект `csv` с помощью метода `writerow`. Метод `writerow` принимает в качестве параметра список, и вы можете его использовать для записи в CSV-файл. Каждый элемент в списке записывается — отделенный разделителем, который вы передали методу `writer` — в строку в CSV-файле. Метод `writerow` создает только одну строку, так что для создания двух строк его нужно вызвать дважды.

Python_ex227.py

```
import csv

with open("st.csv", "w") as f:
    w = csv.writer(f,
                   delimiter=",")
    w.writerow(["один",
                "два",
                "три"])
    w.writerow(["четыре",
                "пять",
                "шесть"])
```

Эта программа создает новый файл с именем *st.csv*. Если вы откроете его в текстовом редакторе, его содержимое будет выглядеть вот так:

```
один,два,три
четыре,пять,шесть
```

Если вы откроете этот файл в программе Excel (или в Google Таблицы, бесплатной альтернативе), запятые исчезнут, и слова `один`, `два` и `три` будут помещены в ячейки в первой строке, а слова `четыре`, `пять` и `шесть` — в ячейки во второй строке.

Модуль `csv` также можно использовать для чтения содержимого файла. Чтобы выполнить чтение из CSV-файла, сначала передайте значение `"r"` в качестве второго параметра функции `open`, чтобы открыть файл для чтения. После этого внутри инструкции `with` вызовите метод `reader`, передав в него файловый объ-

ект и запятую в качестве разделителя — это вернет итерируемый объект, с помощью которого можно получить доступ к каждой строке файла.

Python_ex228.py

```
# убедитесь, что этот файл
# создается из предыдущего примера

import csv

with open("st.csv", "r") as f:
    r = csv.reader(f, delimiter=",")
    for row in r:
        print(",".join(row))
```

```
>> один,два,три
>> четыре,пять,шесть
```

В этом примере вы открываете файл *st.csv* для чтения и конвертируете его в объект `csv`, используя метод `reader`. Затем с помощью цикла вы выполняете перебор объекта `csv`. При каждом прохождении цикла вы вызываете метод `join` в запятой, чтобы добавить запятую между каждым фрагментом данных в файле, и выводите содержимое так, как оно выглядит в исходном файле (с разделяющими запятыми).

Словарь терминов

CSV-файл: файл с расширением *.csv*, внутри которого данные разделяются с помощью запятых (CSV расшифровывается как Comma Separated Values — значения, разделенные запятыми). Часто используется в программах наподобие Excel, обрабатывающих электронные таблицы.
Запись: добавление или изменение данных в файле.
Инструкция `with`: составная инструкция, автоматически выполняющая действие после того, как интерпретатор Python проходит ее.
Путь к файлу: расположение в системе хранения данных (например, на жестком диске) компьютера, в котором сохранен файл.
Разделитель: символ (например, запятая), используемый для разделения данных в CSV-файле.
Файловый объект: объект, который может использоваться для чтения или записи в файл.
Чтение: получение доступа к содержимому файла.

Практикум

1. Найдите у себя на компьютере файл и выведите его содержимое с помощью Python.
2. Напишите программу, которая задает пользователю вопрос и сохраняет ответ в файл.

3. Примите элементы в списке списков `[["Звездные войны", "Терминатор", "Искусственный интеллект"], ["Дурак", "Матильда", "Левиафан"], ["Люди в черном", "Я - робот", "Эволюция"]]` и запишите их в CSV-файл. Данные каждого списка должны быть строкой в файле, при этом каждый элемент списка должен быть отделен запятой.

Решения: *chap9_challenge2.py, chap9_challenge3.py и movies.csv.*

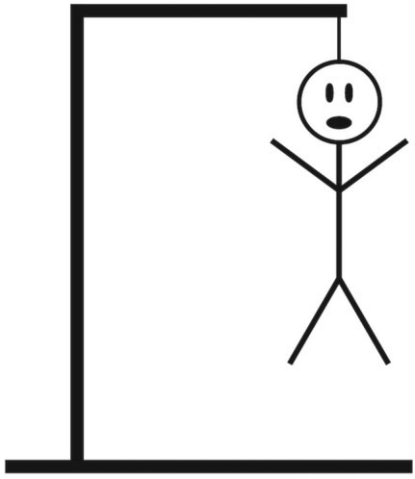
Глава 10. Практикум. Часть I

Всему, что я знаю, я научился из книг.

Авраам Линкольн

В этой главе вы объедините все ранее полученные знания и создадите текстовую игру, классическую «Виселицу». Если вы никогда не играли в «Виселицу», то прочитайте правила ниже:

1. Первый игрок загадывает слово и рисует черту для каждой буквы в этом слове (вы будете использовать нижнее подчеркивание).
2. Второй игрок пытается отгадать слово по одной букве за раз.
3. Если второй игрок правильно угадывает букву, первый игрок заменяет соответствующую черту этой правильной буквой. В данной версии игры, если буква встречается в слове дважды, ее нужно отгадать дважды.
4. Если второй игрок угадал неправильно, первый игрок рисует часть повешенной фигурки (начиная с головы).
5. Если второй игрок отгадывает все слово прежде, чем будет полностью нарисован висельник, он побеждает. Если нет, проигрывает.



В вашей программе компьютер будет первым игроком, а отгадывающий человек — вторым. Вы готовы создать игру «Виселица»?

Игра «Виселица»

Ниже вы видите начало кода игры «Виселица»:

Python_ex229.py

```
def hangman(word):
    wrong = 0
    stages = [""",
              "_________          ",
              "|                  ",
              "|         |        ",
              "|         0        ",
              "|        /|\       ",
              "|        / \       ",
              "|                  "
               ]
    rletters = list(word)
    board = ["__"] * len(word)
    win = False
    print("Добро пожаловать на казнь!")
```

Для начала создайте функцию `hangman`, где будет храниться игра. Эта функция принимает в качестве параметра переменную `word` — это слово, которое должен отгадать второй игрок. Также присутствует переменная `wrong`, которая будет отслеживать, сколько неправильных предположений сделал второй игрок.

Переменная `stages` представляет собой список со строками, которые вы будете использовать, чтобы рисовать висельника. Когда Python выводит каждую строку в списке `stages` на новой строке, появляется рисунок висельника. Переменная `rletters` является списком, содержащим каждый символ в переменной `word`, и она отслеживает, какие буквы осталось отгадать.

Переменная `board` является списком строк, она используется для отслеживания подсказок, которые отображаются для второго игрока. Например, `к__т`, если правильное слово `кот` (и второй игрок уже отгадал буквы `к` и `т`). Вы используете инструкцию `["__"] * len(word)`, чтобы заполнить список `board` (одно нижнее подчеркивание для каждого символа в переменной `word`). Так, если слово — `кот`, тогда `board` вначале принимает вид `["__", "__", "__"]`.

Также есть переменная `win`, вначале принимающая значение `False`, для отслеживания, победил ли уже второй игрок. Затем код выводит строку `Добро пожаловать на казнь!`.

Следующей частью программы является цикл, обеспечивающий работу игры.

Python_ex230.py

```
while wrong < len(stages) - 1:
    print("\n")
```

```
    msg = "Введите букву: "
    char = input(msg)
    if char in rletters:
        cind = rletters.index(char)
        board[cind] = char
        rletters[cind] = '$'
    else:
        wrong += 1
    print((" ".join(board)))
    e = wrong + 1
    print("\n".join(stages[0: e]))
    if "__" not in board:
        print("Вы выиграли! Было загадано слово: ")
        print(" ".join(board))
        win = True
        break
```

Ваш цикл (и игра) продолжается до тех пор, пока значение переменной `wrong` меньше, чем `len(stages) − 1`. Переменная `wrong` отслеживает количество неправильных букв, указанных вторым игроком, так что когда второй игрок предпримет больше неудачных догадок, чем есть строк для изображения фигуры висельника (количество строк в списке `stages`), игра окончится. Нужно вычесть `1` из длины списка `stages`, чтобы уравновесить то, что счет в списке `stages` начинается с `0`, а в списке `wrong` — с `1`.

Внутри цикла сразу выведите пустую область, чтобы украсить игру при выводе в оболочке. Затем примите догадку второго игрока с помощью встроенной функции `input` и сохраните значение в переменной `guess`.

Если значение переменной `guess` содержится в `rletters` (списке, который отслеживает буквы, которые второй игрок еще не угадал), значит, догадка игрока была правильной. В таком случае нужно обновить список `board`, который вы позже используете для отображения оставшихся букв. Если бы второй игрок отгадал `к`, вы бы изменили список `board` на `["к", "__", "__"]`.

Используйте метод `index` в списке `rletters`, чтобы получить первый индекс буквы, которую отгадал второй игрок, и используйте его, чтобы в списке `board` заменить нижнее подчеркивание, имеющее индекс правильно угаданной буквы.

Здесь есть одна проблема. Поскольку `index` возвращает только первый индекс искомого символа, ваш код не будет работать, если переменная `word` содержит повторяющиеся символы. Чтобы обойти это, измените список `rletters`, заменив правильно угаданный символ знаком доллара. Таким образом, при следующем прохождении цикла функция `index` найдет следующее вхождение буквы (если оно есть), а не остановится на первом вхождении.

Если же игрок делает неправильную догадку, вы увеличиваете значение переменной `wrong` на `1`.

Далее вы выводите строчку с результатом угадывания и виселицу при помощи списков `board` и `stages`. Код для первого следующий: `' '.join(board)`.

Вывод висельника немного сложнее. Когда каждая из строк в списке `stages` выводится на новой строке, отображается весь рисунок висельника. Вы можете создать весь рисунок с помощью кода `'\n'.join(stages)`, что добавит строкам в списке `stages` новую строку, чтобы каждая строка выводилась по отдельности.

Чтобы выводить висельника на каждом этапе игры, выполните срез списка `stages`. Начните с `0` и закончите на индексе, соответствующем значению переменной `wrong` плюс один. Нужно прибавить один, поскольку при выполнении среза его конец не включается в результат. Этот срез дает вам лишь те строки, которые нужны для отображения текущей версии висельника.

Наконец, вы проверяете, победил ли второй игрок. Если в списке `board` больше нет нижних подчеркиваний, значит, игрок отгадал все буквы и победил. Если второй игрок победил, выводится строка `Вы выиграли! Было загадано слово:` и правильно отгаданное слово. Также присвойте переменной `win` значение `True`, чтобы прервать цикл.

Как только вы прервали цикл, в случае, если победил второй игрок, игра окончена, и вы ничего не предпринимаете. Если же игрок проиграл, значение переменной `win` `False` — в этом случае, выводится рисунок висельника полностью и строка `Вы проиграли! Было загадано слово:` с указанием слова, которое игрок не смог отгадать.

Python_ex231.py

```
if not win:
print("\n".join(stages[0: wrong]))
print("Вы проиграли! Было загадано слово:
{}.".format(word))
```

Код игры целиком выглядит так:

Python_ex232.py

```
def hangman(word):
    wrong = 0
    stages = ["",
              "________          ",
              "|                 ",
              "|         |       ",
              "|         0       ",
              "|        /|\      ",
              "|        / \      ",
              "|                 "
               ]
    rletters = list(word)
    board = ["__"] * len(word)
```

```
    win = False
    print("Добро пожаловать на казнь!")
    while wrong < len(stages) - 1:
        print("\n")
        msg = "Введите букву: "
        char = input(msg)
        if char in rletters:
            cind = rletters.index(char)
            board[cind] = char
            rletters[cind] = '$'
        else:
            wrong += 1
        print((" ".join(board)))
        e = wrong + 1
        print("\n".join(stages[0: e]))
        if "__" not in board:
            print("Вы выиграли! Было загадано слово: ")
            print(" ".join(board))
            win = True
            break
    if not win:
        print("\n".join(stages[0: wrong]))
        print("Вы проиграли! Было загадано слово:
              {}.".format(word))

hangman("кот")
```

Практикум

Измените игру таким образом, чтобы слово случайно выбиралось из списка слов.

Решение: *chap10_challenge1.py*.

Глава 11. Дополнительная информация

> Практика не приводит к совершенству. Практика приводит к образованию миелина, а миелин приводит к совершенству.
>
> *Дэниел Койл*

Если это ваша первая книга по программированию, рекомендую потратить время на поиск дополнительной информации, прежде чем переходить к следующему разделу. Ниже приведены некоторые ресурсы для исследования — на них также можно получить советы, если у вас возникли трудности.

Для прочтения

1. **softwareengineering.stackexchange.com/questions/44177/what-is-the-single-most-effective-thing-you-did-to-improve-your-programming-skil**

Другие ресурсы

Документация к Python — **docs.python.org/3/**.

Краткий справочник Python — c**loud.github.com/downloads/kroger/python-quick-ref/python-quick-ref.pdf**.

Получение помощи

Если вы у вас возникли трудности, у меня есть для вас несколько предложений. В первую очередь, опубликуйте свой вопрос в группе Self-Taught Programmers в социальной сети Facebook по адресу **www.facebook.com/groups/selftaughtprogrammers**. Это сообщество дружелюбных и целеустремленных программистов, которые помогут найти ответ на любой ваш вопрос.

Я также советую посетить ресурс **www.stackoverflow.com**, на котором можно публиковать вопросы по программированию и получать ответы участников сообщества.

Для меня важным уроком было научиться рассчитывать на помощь других людей. Попытки в чем-то разобраться — это основная часть учебного процесса, но на определенном этапе они могут стать контрпродуктивными. В прошлом, когда я работал над проектами, мои усилия были ниже точки продуктивности. Если подобное случается сегодня, и я не нахожу ответ быстро, то публикую вопрос онлайн. Каждый раз, когда я задавал вопрос в Интернете, на него обязательно отвечали. В этой связи я не могу в полной мере описать то, насколько дружелюбным и стремящимся помочь является программистское сообщество.

ЧАСТЬ II

Введение в объектно-ориентированное программирование

Глава 12. Парадигмы программирования

Существуют лишь два вида языков программирования:
те, которые постоянно ругают, и те, которыми никто не пользуется.

Бьерн Страуструп

Парадигма программирования — это стиль программирования. Существует множество различных парадигм программирования. Для того чтобы программировать профессионально, вам нужно изучить парадигмы либо объектно-ориентированного, либо функционального программирования. В этой главе вы узнаете о процедурном, функциональном и объектно-ориентированном программировании, а больше всего внимания будет уделено объектно-ориентированному программированию.

Состояние

Одним из фундаментальных различий между разными парадигмами программирования является управление **состоянием**. Состояние — это значение переменных в программе при ее работе. **Глобальное состояние** — значение глобальных переменных в программе при ее работе.

Процедурное программирование

В части I вы программировали, используя парадигму **процедурного программирования** — стиль программирования, в котором пишется последовательность шагов по направлению к решению, и каждый шаг изменяет состояние программы. В процедурном программировании вы пишете код, чтобы «сделать это, затем то».

Python_ex233.py

```
x = 2
y = 4
z = 8
xyz = x + y + z
xyz
```

```
>> 14
```

Каждая строка кода в этом примере изменяет состояние программы. Сначала вы определяете x, затем y, затем z. В конце вы определяете значение xyz.

Когда вы используете процедурное программирование, то сохраняете данные в глобальных переменных и управляете ими при помощи функций.

Python_ex234.py

```
rock = []
country = []

def collect_songs():
  song = "Укажите песню."
 ask = "Введите p (рок) или к (кантри). Введите X для выхода"

    while True:
        genre = input(ask)
        if genre == "X":
            break

        if genre == "p":
            rk = input(song)
            rock.append(rk)

        elif genre ==("к"):
            cy = input(song)
            country.append(cy)

        else:
            print("Неверно.")
    print(rock)
    print(country)

collect_songs()
```

```
>> Введите p (рок) или к (кантри). Введите X для выхода
```

Процедурное программирование подходит для написания небольших программ вроде этой, однако из-за того, что все состояния программы сохраняются в глобальных переменных, когда код становится больше, появляются проблемы. Проблема с использованием глобальных переменных заключается в том, что они вызывают непредвиденные ошибки. Когда код вашей программы увеличивается в размере, вы используете глобальные переменные в большом количестве функций, и невозможно отследить все места, в которых глобальная переменная изменяется. Например, функция может изменить значение глобальной переменной, а позже в программе другая функция может изменить ту же глобальную переменную, потому что программист, написавший вторую функцию, забыл, что глобальная переменная уже была изменена первой функцией. Подобные ситуации возникают довольно часто, искажая код программы.

По мере того, как ваша программа усложняется, количество глобальных переменных в ней возрастает. Когда это возрастание совмещается с увеличением числа функций, необходимых программе для обработки новой функциональности, каждая из которых изменяет глобальные переменные, код вашей программы быстро становится непригодным для обслуживания. Более того, этот подход к программированию опирается на **побочные эффекты**. Побочный эффект — это изменение состояния глобальной переменной. При процедурном программировании вы будете часто сталкиваться с непреднамеренными побочными эффектами, такими как случайное двукратное увеличение переменной.

Эта проблема привела к развитию парадигм объектно-ориентированного и функционального программирования, и эти парадигмы используют разные подходы к ее решению.

Функциональное программирование

Функциональное программирование происходит от лямбда-исчисления — наименьшего в мире универсального языка программирования (созданного математиком Алонзо Черчем). Функциональное программирование решает проблемы процедурного программирования с помощью устранения глобального состояния. Функциональный программист полагается на функции, которые не используют и не изменяют глобальное состояние; единственное используемое ими состояние — параметры, которые вы передаете в функцию. Результат, возвращаемый функцией, обычно передается в другую функцию. Таким образом, выполняя передачу из функции в функцию, функциональный программист избегает глобального состояния. Отказ от глобального состояния избавляет от побочных эффектов и сопутствующих им проблем.

Программист из Великобритании, Мэри Роуз Кук, дает функциональному программированию такое определение: «Функциональный код отличается одним свойством: отсутствием побочных эффектов. Он не полагается на данные вне текущей функции, и не меняет данные, находящиеся вне функции»[7]. Свое определение она продолжает примером функции, имеющей побочные эффекты.

[7] maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming

Python_ex235.py

```
a = 0

def increment():
    global a
    a += 1
```

И примером функции без побочных эффектов.

Python_ex236.py

```
def increment(a):
    return a + 1
```

У первой функции есть побочные эффекты, поскольку она полагается на данные за ее пределами и изменяет данные вне текущей функции, увеличивая значение глобальной переменной. У второй функции нет побочных эффектов, так как она не полагается на данные за ее пределами и не изменяет эти данные.

Преимущество функционального программирования заключается в том, что оно устраняет целую категорию ошибок, вызванных глобальным состоянием (в функциональном программировании нет глобального состояния). Его недостаток заключается в том, что некоторые проблемы легче осмыслить при помощи состояния. К примеру, проектирование пользовательского интерфейса с глобальным состоянием легче концептуализировать, чем интерфейс без него. Если вы хотите написать программу, где будет кнопка, нажатие которой переключает режим изображения между видимым и скрытым, такую кнопку проще создать в программе с глобальным состоянием. Можно создать глобальную переменную, принимающую значение `True` или `False` и в зависимости от своего текущего состояния скрывающую или показывающую это изображение. Без глобального состояния создать такую кнопку сложнее.

Объектно-ориентированное программирование

Парадигма **объектно-ориентированного программирования** также решает проблемы, возникающие в процедурном программировании путем устранения глобального состояния, но здесь состояние сохраняется не в функциях а в объектах. В объектно-ориентированном программировании **классы** определяют набор объектов, которые могут взаимодействовать между собой. Классы являются механизмом, позволяющим программисту классифицировать и сгруппировывать похожие объекты. Представьте пакет апельсинов. Каждый апельсин — это объект. Все апельсины обладают схожими свойствами, такими как цвет и вес, но значения этих свойств разные у каждого апельсина. Вы можете использовать класс, чтобы смоделировать апельсины и создать объекты апельсинов с разными значениями. Например, можно определить класс, позволяющий создать объект апельсина темного цвета, весящего 300 грамм, и объект светлого апельсина весом 350 грамм.

Каждый объект — это **экземпляр** класса. Если вы определите класс `Orange` и создадите два объекта `Orange`, каждый из них будет экземпляром класса `Orange`,

и у них будет одинаковый тип данных — `Orange`. Термины «объект» и «экземпляр» взаимозаменяемы. При определении класса экземпляры класса будут похожими — они все будут иметь свойства, определенные в их классе, как цвет или вес для класса, представляющего апельсин, — но свойства каждого экземпляра будут иметь разные значения.

В Python класс является составной инструкцией с заголовком и телом. Класс определяется при помощи синтаксиса `class` *`имя:`* *`тело`*, где *`имя`* — это имя класса, *`тело`* — определяемое вами тело класса. По соглашению, имена классов в Python всегда начинаются с прописной буквы и записываются в горбатом регистре, то есть при наличии в имени класса больше одного слова первые буквы всех слов нужно сделать прописными (`LikeThis`), а не добавлять между словами нижние подчеркивания, как с именами функций. Тело в классе может быть простой или составной инструкцией, называемой **методом**. Методы напоминают функции, только их определяют внутри класса и вызывают в объекте, созданном классом (как в части I, когда вы вызывали методы вроде `"hello".upper()` в строках). Имена методов, как и функций, должны указываться строчными буквами, а слова должны быть отделены нижними подчеркиваниями.

Методы определяются с помощью такого же синтаксиса, что и функции, с двумя отличиями: нужно определить метод как тело в классе, и он должен принимать, по меньшей мере, один параметр (за исключением особых случаев). По соглашению, первый параметр метода всегда называется `self`. При создании метода вы должны всегда определять хотя бы один параметр, поскольку, когда метод вызывается в объекте, Python автоматически передает вызвавший метод объект в этот метод в качестве параметра.

Python_ex237.py

```
1 class Orange:
2     def __init__(self):
3         print("Создано!")
```

Вы можете использовать параметр `self`, чтобы определить *переменную экземпляра* — переменную, принадлежащую объекту. Если вы создадите несколько объектов, у них всех будут разные значения переменных экземпляра. Переменные экземпляра объявляются с помощью синтаксиса `self.`*`имя_переменной = значение_переменной`*. Обычно переменные экземпляра определяются внутри специального метода `__init__` (от англ. слова initialize — инициализировать), который вызывается Python при создании объекта.

Ниже приведен пример класса, представляющего апельсин.

Python_ex238.py

```
1 class Orange:
2     def __init__(self, w, c):
3         self.weight = w
4         self.color = c
5         print("Создано!")
```

Код в __init__ выполняется при создании объекта Orange (чего в этом примере не происходит) и создает две переменные экземпляра: weight и color. Их можно использовать как обычные переменные, в любом методе вашего класса. Когда вы создаете объект Orange, код в __init__ выводит строку Создано!. Любой метод, окруженный двойными нижними подчеркиваниями (как __init__), является **магическим методом**, который Python использует для особых целей, таких как создание объекта.

Вы можете создать новый объект Orange с помощью того же синтаксиса, что вы использовали для вызова функции — *имя_функции(параметры)* — только замените *имя_функции* именем класса, который вы хотите использовать для создания объекта, а слово *параметры* — параметрами, которые принимает __init__. Не нужно передавать self, Python сделает это автоматически. Создание нового объекта называется **созданием экземпляра класса**.

Python_ex239.py

```
1  class Orange:
2      def __init__(self, w, c):
3          self.weight = w
4          self.color = c
5          print("Создано!")

6  or1 = Orange(10, "темный апельсин")
7  print(or1)
```

```
>> Создано!
```

После определения класса, вы создаете экземпляр класса Orange при помощи кода Orange(10, "темный апельсин"), в этом случае выводится строка Создано!. Затем вы выводите сам объект Orange, Python сообщает вам, что это объект Orange, и выдает его местонахождение в памяти (в вашем случае, расположение в памяти не будет совпадать с указанным в данном примере).

Как только вы создали объект, можно получить значение его переменных экземпляра с помощью синтаксиса *имя_объекта.имя_переменной*.

Python_ex240.py

```
1  class Orange:
2      def __init__(self, w, c):
3          self.weight = w
4          self.color = c
5          print("Создано!")

6  or1 = Orange(10, "темный апельсин")
7  print(or1.weight)
8  print(or1.color)
```

```
>> Создано!
>> 10
>> темный апельсин
```

Значение переменной экземпляра можно изменить с помощью синтаксиса *имя_объекта.имя_переменной = новое_значение*.

Python_ex241.py

```
1  class Orange:
2      def __init__(self, w, c):
3          self.weight = w
4          self.color = c
5          print("Создано!")

6  or1 = Orange(10, "темный апельсин")
7  or1.weight = 100
8  or1.color = "светлый апельсин"

9  print(or1.weight)
10 print(or1.color)
```

```
>> Создано!
>> 100
>> светлый апельсин
```

Несмотря на то что значения переменных экземпляра `color` и `weight` были `"темный апельсин"` и `10`, соответственно, вы смогли их изменить, присвоив им значения `"светлый апельсин"` и `100`.

Используя класс `Orange`, вы можете создать множество апельсинов.

Python_ex242.py

```
1  class Orange:
2      def __init__(self, w, c):
3          self.weight = w
4          self.color = c
5          print("Создано!")

6  or1 = Orange(4, "светлый апельсин")
7  or2 = Orange(8, "темный апельсин")
8  or3 = Orange(14, "желтый апельсин")
```

```
>> Создано!
>> Создано!
>> Создано!
```

Апельсин не определяется одними только физическими свойствами вроде цвета и веса. Апельсины делают разные вещи — например, гниют — и вы можете смоделировать их с помощью методов. Ниже показано, как можно наделить объект `Orange` возможностью «гнить».

Python_ex243.py

```
class Orange():
    def __init__(self, w, c):
        """вес в граммах"""
        self.weight = w
        self.color = c
        self.mold = 0
        print("Создано!")

    def rot(self, days, temp):
        self.mold = days * temp

orange = Orange(6, "апельсин")
print(orange.mold)
orange.rot(10, 33)
print(orange.mold)
```

```
>> Создано!
>> 0
>> 330
```

Метод `rot` принимает два параметра: число дней, прошедших с тех пор как кто-то сорвал апельсин, и среднюю температуру за этот период. Когда вы вызываете метод, он использует формулу для увеличения переменной экземпляра `mold` — это работает, поскольку вы можете изменять значение любой переменной экземпляра внутри любого метода. Теперь апельсин может гнить.

В классе можно определять множество методов. Ниже приведен пример построения модели прямоугольника при помощи метода для расчета его площади и другого метода для изменения его размера.

Python_ex244.py

```
class Rectangle():
    def __init__(self, w, l):
        self.width = w
        self.len = l

    def area(self):
        return self.width * self.len
```

```
    def change_size(self, w, l):
        self.width = w
        self.len = l

rectangle = Rectangle(10, 20)
print(rectangle.area())
rectangle.change_size(20, 40)
print(rectangle.area())
```

```
>> 200
>> 800
```

В этом примере объекты `Rectangle` имеют две переменные экземпляра: `len` и `width`. Метод `area` возвращает площадь объекта `Rectangle`, перемножая между собой переменные экземпляра, а метод `change_size` изменяет переменные, присваивая им числа, которые передаются в качестве параметров.

У объектно-ориентированного программирования есть несколько преимуществ. Эта парадигма способствует повторному использованию кода и вследствие этого сокращает количество времени, необходимое на разработку и обслуживание кода. Проблемы разбиваются на множество фрагментов, благодаря чему код становится легче поддерживать. Недостатком объектно-ориентированного программирования можно считать то, что создание программ требует больших усилий, поскольку их разработка включает огромный объем планирования.

Словарь терминов

Глобальное состояние: значение глобальных переменных в программе при ее работе.
Классы: механизм, позволяющий программисту классифицировать и сгруппировывать похожие объекты.
Магический метод: метод, который Python использует в разных ситуациях, например, при создании объекта.
Методы: тело в классах. Методы похожи на функции, только их определяют внутри класса и вызывают только в объекте, созданном классом.
Объектно-ориентированное программирование: парадигма программирования, где вы определяете объекты, которые взаимодействуют друг с другом.
Парадигма программирования: стиль программирования.
Переменные экземпляра: переменные, которые принадлежат объекту.
Побочный эффект: изменение состояния глобальной переменной.
Процедурное программирование: стиль программирования, в котором пишется последовательность шагов по направлению к решению, и каждый шаг изменяет состояние программы.
Создание экземпляра класса: создание нового объекта при помощи класса.
Состояние: это значение переменных в программе во время ее работы.
Функциональное программирование: решает проблемы процедурного программирования с помощью устранения глобального состояния, передавая его от функции к функции.

Экземпляр: каждый объект это экземпляр класса. Каждый экземпляр класса имеет тот же тип, что и все остальные экземпляры этого класса.

Практикум

1. Определите класс `Apple` с четырьмя переменными экземпляра, представляющими четыре свойства яблока.
2. Создайте класс `Circle` с методом `area`, подсчитывающим и возвращающим площадь круга. Затем создайте объект `Circle`, вызовите в нем метод `area` и выведите результат. Воспользуйтесь функцией `pi` из встроенного в `Python` модуля `math`.
3. Создайте класс `Triangle` с методом `area`, подсчитывающим и возвращающим площадь треугольника. Затем создайте объект `Triangle`, вызовите в нем `area` и выведите результат.
4. Создайте класс `Hexagon` с методом `calculate_perimeter`, подсчитывающим и возвращающим периметр шестиугольника. Затем создайте объект `Hexagon`, вызовите в нем `calculate_perimeter` и выведите результат.

Решения: *chall_1.py – chall_4.py*.

Глава 13. Четыре столпа объектно-ориентированного программирования

Хороший дизайн приносит пользу быстрее,
чем увеличивает расходы.

Томас Гейл

В объектно-ориентированном программировании есть четыре главные концепции: инкапсуляция, абстракция, полиморфизм и наследование. Вместе они образовывают **четыре столпа объектно-ориентированного программирования**. Чтобы язык программирования действительно можно было назвать объектно-ориентированным, как Python, Java и Ruby, в нем должны присутствовать все четыре концепции. В этой главе вы узнаете о каждом из столпов объектно-ориентированного программирования.

Инкапсуляция

Инкапсуляция относится к двум концепциям. Первая заключается в том, что в объектно-ориентированном программировании переменные (состояние) и методы (для изменения состояния либо выполнения вычислений, использующих состояние) группируются в единое целое — объект.

Python_ex245.py

```
1 class Rectangle():
2     def __init__(self, w, l):
3         self.width = w
```

```
        self.len = l

    def area(self):
        return self.width * self.len
```

В этом случае переменные экземпляра класса `len` и `width` хранят состояние объекта. Состояние объекта сгруппировано в том же блоке (объекте), что и метод `area`. Метод использует состояние объекта, чтобы вернуть площадь прямоугольника.

Вторая концепция, собственно инкапсуляция, заключается в сокрытии внутренних данных класса для предотвращения получения **клиентом** (кодом вне класса, который использует объект) прямого доступа к этим данным.

Python_ex246.py

```
class Data:
    def __init__(self):
        self.nums = [1, 2, 3, 4, 5]

    def change_data(self, index, n):
        self.nums[index] = n
```

Класс `Data` имеет переменную `nums` экземпляра класса, содержащую список целых чисел. Как только вы создали объект `Data`, есть два способа изменить элементы в `nums` — при помощи метода `change_data` или напрямую получая доступ к переменной `nums` экземпляра с помощью объекта `Data`.

Python_ex247.py

```
class Data:
    def __init__(self):
        self.nums = [1, 2, 3, 4, 5]

    def change_data(self, index, n):
        self.nums[index] = n

data_one = Data()
data_one.nums[0] = 100
print(data_one.nums)

data_two = Data()
data_two.change_data(0, 100)
print(data_two.nums)
```

```
>> [100, 2, 3, 4, 5]
>> [100, 2, 3, 4, 5]
```

Оба способа изменять элементы в переменной экземпляра `nums` работают, но что же произойдет, если вы решите сделать переменную `nums` кортежем вместо списка? Если вы осуществите это изменение, любой код клиента, пытающийся изменить элементы в переменной `nums`, как с `nums[0] = 100`, больше не будет работать, поскольку кортежи неизменяемы.

Многие языки программирования решают эту проблему, разрешая программистам определять **закрытые переменные** и **закрытые методы** — переменные и методы, к которым могут обращаться объекты в коде, реализующем различные методы, но клиент не может. Закрытые переменные и методы полезны, когда у вас есть метод или переменные, используемые внутри класса, но вы планируете позже изменить реализацию своего кода (или желаете сохранить гибкость этой опции) и потому не хотите, чтобы тот, кто использует этот класс, полагался на них (ведь они могут измениться и нарушить код клиента). Закрытые переменные являются примером второй концепции, к которой относится инкапсуляция; закрытые переменные скрывают внутренние данные класса от прямого доступа к ним клиента. В противоположность, **открытые переменные** — это те, к которым клиент может получить доступ.

В Python нет закрытых переменных. Все его переменные — открытые. К решению проблемы, с которой справляются закрытые переменные, Python подходит иным образом — используя конвенции (соглашения) имен. Если у вас есть переменная или метод, к которым вызывающий их не должен получить доступ, перед их именами необходимо добавить нижнее подчеркивание. Программисты Python знают, что если имя метода или переменной начинается с символа подчеркивания, их нельзя использовать (хотя они все еще могут попытаться сделать это на свой страх и риск).

Python_ex248.py

```
class PublicPrivateExample:
    def __init__(self):
        self.public = "безопасно"
        self._unsafe = "опасно"

    def public_method(self):
        # клиенты могут это использовать
        pass

    def _unsafe_method(self):
        # клиенты не должны это использовать
        pass
        self.public = "безопасно"
        self._unsafe = "опасно"
```

Программисты, читающие этот код, понимают, что переменную `self.public` использовать безопасно, но они не должны использовать переменную `self._unsafe`, поскольку она начинается с подчеркивания, — а если все же решает ее использовать, то на свой страх и риск. Человек, занимающийся под-

держкой этого кода, не обязан содержать переменную `self._unsafe`, поскольку вызывающие ее не должны иметь к ней доступ. Программисты знают, что метод `public_method` использовать безопасно, а метод `_unsafe_method` — нет, ведь имя последнего начинается с подчеркивания.

Абстракция

Абстракция — это процесс «отнятия или удаления у чего-то характеристик с целью сведения его к набору основных, существенных характеристик»[8]. В объектно-ориентированном программировании абстракция используется, когда объекты моделируются с использованием классов, а ненужные подробности опускаются.

Скажем, вы создаете модель человека. Человек комплексен — у него есть цвет волос, цвет глаз, рост, вес, этническая принадлежность, пол и многое другое. Если для представления человека вы создадите класс, некоторые из этих данных могут оказаться не важными в контексте проблемы, которую вы пытаетесь решить. Примером абстракции может быть создание класса `Person` с опущением некоторых свойств человека, например, цвета глаз или веса. Объекты `Person`, которые создает ваш класс, являются абстракциями людей. Это представление человека, урезанное до основных характеристик, необходимых для решения конкретной проблемы.

Полиморфизм

Полиморфизмом называют «способность (в программировании) представлять один и тот же интерфейс для разных базовых форм (типов данных)»[9]. Интерфейс — это функция или метод. Ниже приведен пример представления одного и того же интерфейса для разных типов данных.

Python_ex249.py

```
1 print("Привет, мир!")
2 print(200)
3 print(200.1)
```

```
>> Привет, мир!
>> 200
>> 200.1
```

Вы представили один интерфейс, функцию `print`, для трех разных типов данных: строки, целого числа и числа с плавающей точкой. Вам не нужно определять и вызывать три отдельных функции (вроде `print_string` для вывода строк, `print_int` для вывода целых чисел и `print_float` для вывода чисел с плавающей точкой), чтобы вывести три разных типа данных; вместо этого вы использовали функцию `print` для представления одного интерфейса, чтобы вывести их все.

[8] whatis.techtarget.com/definition/abstraction

[9] stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-forand-how-is-it-used

Встроенная функция `type` возвращает тип данных объекта.

Python_ex250.py

```
print("Привет, мир!")
print(200)
print(200.1)
```

```
>> <class 'str'>
>> <class 'int'>
>> <class 'float'>
```

Скажем, вам нужно написать программу, создающую три объекта, которые сами себя рисуют: треугольники, квадраты и круги. Этого можно достичь, определив три различных класса `Triangle`, `Square` и `Circle`, и метод `draw` для каждого из них. `Triangle.draw()` будет рисовать треугольник, `Square.draw()` — квадрат, `Circle.draw()` — круг. Таким образом, каждый объект имеет интерфейс `draw`, знающий, как рисовать самого себя. Вы представили один и тот же интерфейс для трех различных типов данных.

Если бы Python не поддерживал полиморфизм, вам понадобился бы свой метод для рисования каждой фигуры — к примеру, `draw_triangle` для рисования объекта `Triangle`, `draw_square` — для рисования объекта `Square` и `draw_circle` — для рисования объекта `Circle`.

Кроме того, если бы у вас был список этих объектов, и вы хотели бы нарисовать каждый из них, вам бы пришлось узнавать тип каждого объекта, затем вызывать подходящий для этого типа метод, увеличивая программу, делая ее более сложной для чтения и написания, а также менее надежной. И это усложнило бы процесс совершенствования программы, поскольку всякий раз при добавлении новой фигуры вам пришлось бы отслеживать каждое место в коде, где вы рисуете фигуры, проверять их тип данных (чтобы выяснить, какой метод использовать) и вызывать новую функцию рисования. Ниже приведен пример рисования фигур с полиморфизмом и без него.

Python_ex251.py

```
# Не выполнять.

# Рисование фигур
# без полиморфизма
shapes = [tr1, sq1, cr1]
for a_shape in shapes:
    if type(a_shape) == "Треугольник":
        a_shape.draw_triangle()
    if type(a_shape) == "Квадрат":
        a_shape.draw_square()
    if type(a_shape) == "Круг":
        a_shape.draw_circle()
```

```
# Рисуем фигуры
# с помощью полиморфизма
shapes = [tr1,
          sw1,
          cr1]
for a_shape in shapes:
    a_shape.draw()
```

Если бы вы пожелали добавить новую фигуру в список `shapes` без полиморфизма, вам пришлось бы изменить код в цикле `for`, чтобы проверить тип `a_shape`, и вызывать его метод рисования. С помощью единообразного полиморфического интерфейса вы можете впредь добавлять столько классов фигур в список `shapes`, сколько пожелаете, и фигура будет рисовать себя без какого-либо дополнительного кода.

Наследование

Наследование в программировании напоминает наследование в биологии. При генетическом наследовании вы наследуете характеристики вроде цвета глаз от родителей. Аналогично, когда вы создаете класс, он может наследовать методы и переменные от другого класса. Класс, от которого наследуют, называется **родительским**, а класс, который наследует, — **дочерним**. В этом разделе вы будете моделировать фигуры, используя наследование. Вот класс, который моделирует фигуру.

Python_ex252.py

```
class Shape():
    def __init__(self, w, l):
        self.width = w
        self.len = l

    def print_size(self):
        print("""{} на {}
              """.format(self.width,
                         self.len))

my_shape = Shape(20, 25)
my_shape.print_size()
```

```
>> 20 на 25
```

При помощи этого класса вы можете создавать объекты фигур `Shape` с шириной `width` и длиной `len`. К тому же объекты `Shape` имеют метод `print_size`, выводящий их ширину и длину.

Вы можете определить дочерний класс, наследующий от родительского, передав имя родительского класса в качестве параметра дочернему классу при его создании. В следующем примере создается класс `Square`, наследующийся от класса `Shape`.

Python_ex253.py

```
class Shape():
    def __init__(self, w, l):
        self.width = w
        self.len = l

    def print_size(self):
        print("""{} на {}
              """.format(self.width,
                         self.len))

class Square(Shape):
    pass

a_square = Square(20,20)
a_square.print_size()
```

```
>> 20 на 20
```

Поскольку вы передали класс `Shape` в качестве параметра в класс `Square`, класс `Square` наследует переменные и методы класса `Shape`. Тело, которое вы определили в классе `Square`, состояло лишь из ключевого слова `pass`, сообщающего Python ничего не делать.

Благодаря наследованию вы можете создавать объект `Square`, передавать ему ширину и длину и вызывать в нем метод `print_size` без необходимости писать еще какой-либо код (за исключением `pass`) в классе `Square`. Такое сокращение кода важно, поскольку избегание повторений кода уменьшает вашу программу и делает ее лучше управляемой.

Дочерний класс не отличается от любого другого — вы можете определять в нем методы и переменные, не затрагивая родительский класс.

Python_ex254.py

```
class Shape():
    def __init__(self, w, l):
        self.width = w
        self.len = l

def print_size(self):
    print("""{} на {}
          """.format(self.width,
                     self.len))
```

```
 9 class Square(Shape):
10     def area(self):
11         return self.width * self.len

12 a_square = Square(20, 20)
13 print(a_square.area())
```

```
>> 400
```

Когда дочерний класс наследует метод от родительского класса, его можно переопределить, определив новый метод с таким же именем, как у метода, который был унаследован. Способность дочернего класса изменять реализацию метода, унаследованного от его родительского класса, называется переопределением метода.

Python_ex255.py

```
 1 class Shape():
 2     def __init__(self, w, l):
 3         self.width = w
 4         self.len = l

 5 def print_size(self):
 6     print("""{} by {}
 7           """.format(self.width,
 8                      self.len))

 9 class Square(Shape):
10     def area(self):
11         return self.width * self.len

12 def print_size(self):
13     print("""Я {} на {}
14           """.format(self.width,
15                      self.len))

16 a_square = Square(20, 20)
17 a_square.print_size()
```

```
>> Я 20 на 20
```

В этом случае, поскольку вы определили метод `print_size`, новый метод переопределяет родительский метод с таким же именем, и выводит новое сообщение при вызове.

Композиция

Теперь, когда вы узнали о четырех столпах объектно-ориентированного программирования, я раскрою смысл еще одной важной концепции — **композиции**. Композиция создает отношение «имеет», сохраняя объект в другом объекте как переменную. Например, композиция может использоваться для представления отношения между собакой и ее хозяином (собака имеет хозяина). Для этого сначала создайте классы, представляющие собак и людей.

Python_ex256.py

```
class Dog():
    def __init__(self,
                 name,
                 breed,
                 owner):
        self.name = name
        self.breed = breed
        self.owner = owner

class Person():
    def __init__(self, name):
        self.name = name
```

Затем, когда вы создаете объект `Dog`, то передаете объект `Person` в качестве параметра хозяина.

Python_ex257.py

```
# Продолжение
# предыдущего примера

mick = Person("Мик Джаггер")
stan = Dog("Стэнли",
           "Бульдог",
           mick)
print(stan.owner.name)
```

```
>> Мик Джаггер
```

Теперь объект `stan` с именем `"Стэнли"` имеет хозяина — объект `Person` с именем `"Мик Джаггер"`, хранящийся в переменной экземпляра класса `owner`.

Словарь терминов

Абстракция: процесс «отнятия или удаления у чего-то характеристик с целью сведения его к набору основных, существенных характеристик»[10].
Дочерний класс: класс, который наследуется.
Инкапсуляция: относится к двум концепциям. Первая заключается в том, что в объектно-ориентированном программировании переменные (состояние) и методы (для изменения состояния либо выполнения вычислений, использующих состояние) группируются в единое целое — объект. Вторая концепция, собственно инкапсуляция, заключается в сокрытии внутренних данных класса для предотвращения получения клиентом прямого доступа к этим данным.
Клиент: код вне класса, который использует объект.
Композиция: композиция моделирует отношение «имеет», сохраняя объект в другом объекте как переменную.
Наследование: при генетическом наследовании вы наследуете свойства вроде цвета глаз от родителей. Аналогично, когда вы создаете класс, он может наследовать методы и переменные от другого класса.
Переопределение метода: способность дочернего класса изменять реализацию метода, унаследованного от его родительского класса.
Полиморфизм: полиморфизмом называют «способность (в программировании) представлять один и тот же интерфейс для разных базовых форм (типов данных)»[11].
Родительский класс: класс, от которого осуществляется наследование.
Четыре столпа объектно-ориентированного программирования: четыре главные концепции в объектно-ориентированном программировании: наследование, полиморфизм, абстракция и инкапсуляция.

Практикум

1. Создайте классы `Rectangle` и `Square` с методом `calculate_perimeter`, вычисляющим периметр фигур, которые эти классы представляют. Создайте объекты `Rectangle` и `Square` вызовите в них этот метод.
2. В классе `Square` определите метод `change_size`, позволяющий передавать ему число, которое увеличивает или уменьшает (если оно отрицательное) каждую сторону объекта `Square` на соответствующее значение.
3. Создайте класс `Shape`. Определите в нем метод `what_am_i`, который при вызове выводит строку `"Я — фигура."`. Измените ваши классы `Rectangle` и `Square` из предыдущих заданий для наследования от `Square`, создайте объекты `Square` и `Rectangle` и вызовите в них новый метод.
4. Создайте классы `Horse` и `Rider`. Используйте композицию, чтобы смоделировать лошадь с всадником на ней.

Решения: *chall_1.py – chall_4.py*.

[10] whatis.techtarget.com/definition/abstraction
[11] stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-forand-how-is-it-used

Глава 14. Еще об объектно-ориентированном программировании

Относитесь к своему коду как к поэзии
и сводите его к абсолютному минимуму.

Илья Дорман

В этой главе я описываю дополнительные концепции, относящиеся к объектно-ориентированному программированию.

Переменные класса и переменные экземпляра

В Python классы являются объектами. Идея пошла из Smalltalk, влиятельного языка программирования, положившего начало объектно ориентированному программированию. Каждый класс в Python — объект, являющийся экземпляром «типа» класса.

Python_ex258.py

```
1 class Square:
2     pass

3 print(Square)
```

```
>> <class '__main__.Square'>
```

В этом примере класс `Square` является объектом, и вы его вывели.

У классов есть два типа переменных — **переменные класса** и **переменные экземпляра класса**. Переменные, которые вам встречались до этого, были переменными экземпляра класса, определяемыми синтаксисом `self.`*`имя_переменной = значение_переменной`*. Переменные экземпляра класса принадлежат объектам.

Python_ex259.py

```
 1 class Rectangle():
 2     def __init__(self, w, l):
 3         self.width = w
 4         self.len = l

 5     def print_size(self):
 6         print("""{} на {}
 7               """.format(self.width,
 8                          self.len))

 9 my_rectangle = Rectangle(10, 24)
10 my_rectangle.print_size()
```

```
>> 10 на 24
```

В этом примере `width` и `len` — переменные экземпляра класса.

Переменные класса принадлежат объекту, который Python создает для каждого определения класса, и объектам, которые они создают. Переменные класса определяются как обычные переменные (но определение осуществляется внутри класса). Вы можете получить к ним доступ с помощью объектов класса и объектов, созданных объектами класса. Вы получаете к ним доступ так же, как и к переменным экземпляра, добавляя перед именем переменной `self`. Переменные класса полезны — они позволяют обмениваться данными между всеми экземплярами класса без полагания на глобальные переменные.

Python_ex260.py

```
class Rectangle():
    recs = []

    def __init__(self, w, l):
        self.width = w
        self.len = l
        self.recs.append((self.width,
                          self.len))

    def print_size(self):
        print("""{} на {}
              """.format(self.width,
                         self.len))

r1 = Rectangle(10, 24)
r2 = Rectangle(20, 40)
r3 = Rectangle(100, 200)

print(Rectangle.recs)
```

```
>> [(10, 24), (20, 40), (100, 200)]
```

В этом примере вы добавили переменную `recs` в класс `Rectangle`. Вы определили ее вне метода `__init__`, поскольку Python вызывает метод `__init__` только при создании объекта, а вы хотите получить доступ к переменной класса, используя объект класса (что не вызывает метод `__init__`).

Затем вы создали три объекта `Rectangle`. Всякий раз, как создается объект `Rectangle`, код в методе `__init__` присоединяет кортеж, содержащий ширину и длину нового объекта, к списку `recs`. С помощью этого кода каждый новый объект `Rectangle` автоматически добавляется в список `recs`. Благодаря использованию переменной класса вы смогли обменяться данными между различными объектами, созданными классом, без необходимости использовать глобальную переменную.

Магические методы

Каждый класс в Python наследуется от родительского класса `Object`. Python использует методы, унаследованные от `Object`, в различных ситуациях — например, когда вы выводите объект.

Python_ex261.py

```
1 class Lion:
2     def __init__(self, name):
3         self.name = name

4 lion = Lion("Дилберт")
5 print(lion)
```

```
>> <__main__.Lion object at 0x101178828>
```

Когда вы выводите объект `Lion`, Python вызывает в нем магический метод `__repr__`, унаследованный этим объектом от `Object`, и выводит то, что возвращает `__repr__`. Можно переопределить унаследованный метод `__repr__`, чтобы изменить его вывод.

Python_ex262.py

```
1 class Lion:
2     def __init__(self, name):
3         self.name = name

4     def __repr__(self):
5         return self.name

6 lion = Lion("Дилберт")
7 print(lion)
```

```
>> Дилберт
```

Поскольку вы переопределили метод `__repr__`, унаследованный от `Object`, и изменили его таким образом, чтобы он выводил имя объекта `Lion`, то когда вы выводите этот объект, выводится его имя — в этом случае `Дилберт` — вместо чего-то вроде `<__main__.Lion object at 0x101178828>`, что возвращал бы метод `__repr__`.

Операнды в выражении должны иметь магический метод, который оператор может использовать для оценки выражения. Например, в выражении `2 + 2` каждый объект целого числа имеет магический метод `__add__`, вызываемый Python при оценке выражения. Если вы определите метод `__add__` в классе, то сможете использовать создаваемые им объекты как операнды в выражении с оператором сложения.

Python_ex263.py

```
class AlwaysPositive:
    def __init__(self, number):
        self.n = number

    def __add__(self, other):
        return abs(self.n +
                   other.n)

x = AlwaysPositive(-20)
y = AlwaysPositive(10)

print(x + y)
```

```
>> 10
```

Объекты `AlwaysPositive` могут использоваться в качестве операндов в выражении с оператором сложения, поскольку вы определили метод `__add__`. Когда Python определяет значение выражения с оператором сложения, он вызывает метод `__add__` в первом объекте операнда, передает второй объект операнда в `__add__` в качестве параметра и возвращает результат.

В этом случае `__add__` использует встроенную функцию `abs`, чтобы вернуть абсолютное значение двух сложенных в выражении чисел. Поскольку вы определили `__add__` таким образом, два объекта `AlwaysPositive` в выражении с оператором сложения всегда будут давать абсолютную величину суммы двух объектов, следовательно, результат выражения всегда будет положительным.

Ключевое слово `is`

Ключевое слово `is` возвращает значение `True`, если два объекта являются одним и тем же объектом, и `False` в противном случае.

Python_ex264.py

```
class Person:
    def __init__(self):
        self.name = 'Боб'

bob = Person()
same_bob = bob
print(bob is same_bob)

another_bob = Person()
print(bob is another_bob)
```

```
>> True
>> False
```

Когда вы используете ключевое слово `is` в выражении с объектами `bob` и `same_bob` в качестве операндов, выражение принимает значение `True`, так как обе переменные указывают на один и тот же объект `Person`. Когда вы создаете новый объект `Person` и сравниваете его с исходным объектом `bob`, выражение принимает значение `False`, поскольку переменные указывают на различные объекты `Person`.

Используйте ключевое слово `is`, чтобы проверить, присвоено ли переменной значение `None`.

Python_ex265.py

```
x = 10
if x is None:
    print("x равно None :( ")
else:
    print("x не равно None")

x = None
if x is None:
    print("x равно None")
else:
    print("x равно None :( ")
```

```
>> x не равно None
>> x равно None :(
```

Словарь терминов

Закрытый метод: метод, к которому объект может получить доступ, а клиент — нет.
Закрытая переменная: переменная, к которой объект может получить доступ, а клиент — нет.
Открытая переменная: переменная, к которой клиент может получить доступ.
Переменная класса: переменные класса принадлежат объекту класса и объектам, которые он создает.
Переменная экземпляра: переменная экземпляра принадлежит объекту.

Практикум

1. Добавьте переменную `square_list` в класс `Square` так, чтобы всякий раз, когда вы создаете новый объект `Square`, он добавлялся в список.
2. Измените класс `Square` так, чтобы когда вы выводите объект `Square`, выводилось сообщение с длинами всех четырех сторон фигуры. Например, если вы создадите квадрат при помощи `Square(29)` и осуществите вывод, Python должен вывести строку `29 на 29 на 29 на 29`.
3. Напишите функцию, которая принимает два объекта в качестве параметров и возвращает `True`, если они являются одним и тем же объектом, и `False` в противном случае.

Решения: *chall_1.py – chall_3.py*.

Глава 15. Практикум. Часть II

Пока код не запустится, все это лишь разговоры.

Уорд Каннингем

В этой главе вы создадите популярную карточную игру «Пьяница». В этой игре каждый игрок берет из колоды по карте — побеждает тот, у которого карта старше. При создании игры вы будете определять классы, представляющие карту, колоду, игрока и, наконец, саму игру.

Карты

Ниже показан класс, который моделирует игру в карты.

Python_ex266.py

```
class Card:
    suits = ["пикей",
             "червей",
             "бубей",
             "треф"]

    values = [None, None,"2", "3",
              "4", "5", "6", "7",
              "8", "9", "10",
              "валета", "даму",
              "короля", "туза"]

    def __init__(self, v, s):
        """suit и value — целые числа"""
        self.value = v
        self.suit = s

    def __lt__(self, c2):
        if self.value < c2.value:
            return True
        if self.value == c2.value:
            if self.suit < c2.suit:
                return True
            else:
                return False
        return False
    def __gt__(self, c2):
```

```
        if self.value > c2.value:
            return True
        if self.value == c2.value:
            if self.suit > c2.suit:
                return True
            else:
                return False
        return False

    def __repr__(self):
        v = self.values[self.value] + " of " \
        + self.suits[self.suit]
        return v
```

У класса `Card` есть две переменные класса, `suits` и `values`. `suits` — это список строковых значений, представляющих все масти, которые могут быть у карты: `пикей`, `червей`, `бубей`, `треф`. `values` — это список строковых значений, представляющих различные номиналы карт: `2` — `10`, `валета`, `даму`, `короля` и `туза`. Элементы под первыми двумя индексами в списке `values` являются `None` — для того, чтобы строки в списке совпадали с индексами, которые они представляют (так строка `"2"` в списке `values` имеет индекс `2`).

Объекты `Card` имеют две переменные экземпляра — `suit` и `value`, каждая из них представлена целым числом. Вместе эти переменные экземпляра класса представляют вид карты объекта `Card`. К примеру, вы создаете `2 червей` путем создания объекта `Card` и передачи в него параметров `2` (для значения) и `1` (для масти, поскольку индекс червей в списке `suits` — `1`).

Определения в магических методах `__lt__` и `__gt__` позволяют вам сравнивать два объекта `Card` в выражении при помощи операторов больше и меньше. Код в этих методах выясняет, больше или меньше карта, чем другая карта, переданная в качестве параметра. Код в магических методах также обрабатывает ситуации, когда значение карт одинаковое, например, если обе карты — десятки. В таком случае методы используют масти, чтобы избежать ничьей. Масти упорядочены в списке `suits` по возрастанию силы — самая сильная масть располагается последней, то есть она назначена наибольшему индексу, а самая слабая масть назначена наименьшему индексу.

Python_ex267.py

```
card1 = Card(10, 2)
card2 = Card(11, 3)
print(card1 < card2)
```

```
>> True
```

Python_ex268.py

```
card1 = Card(10, 2)
card2 = Card(11, 3)
print(card1 > card2)
```

```
>> False
```

Последний метод в классе `Card` — магический метод `__repr__`. Он использует переменные экземпляра `value` и `suit` для нахождения значения и масти карты в списках `values` и `suit`, и возвращает их, чтобы вы могли вывести карту, которую представляет объект `Card`.

Python_ex269.py

```
card = Card(3, 2)
print(card)
```

```
>> 3 бубей
```

Колода

Теперь нужно определить класс, представляющий колоду карт.

Python_ex270.py

```
from random import shuffle

class Deck:
    def __init__(self):
        self.cards = []
        for i in range(2, 15):
            for j in range(4):
                self.cards.append(Card(i, j))
        shuffle(self.cards)

    def rm_card(self):
        if len(self.cards) == 0:
            return
        return self.cards.pop()
```

Когда вы создаете объект `Deck`, два цикла `for` в `__init__` создают объекты, представляющие все карты в 52-карточной колоде, и добавляют их в список `cards`. Первый цикл перебирает от `2` до `15`, поскольку первое значение для карты — это `2`, а последнее — `14` (туз). При каждом прохождении внутреннего цикла создается новая карта путем использования целого числа из внешнего цикла в качестве значения (например, `14` для туза) и целого числа из внутреннего цикла в качестве масти (например, `2` для червей). Так создаются 52 карты, по карте

на каждую комбинацию масти и значения. После того как метод создает карты, метод `shuffle` из модуля `random` случайным образом перемешивает элементы в списке `cards`, имитируя перетасовку колоды карт.

У нашей колоды есть еще один метод, `rm_card`, который изымает и возвращает карту из списка `cards`, или возвращает `None`, если список пуст. Вы можете использовать класс `Deck` для создания новой колоды карт и вывода каждой карты в ней.

Python_ex271.py

```
deck = Deck()
for card in deck.cards:
    print(card)
```

```
>> 4 пикей
>> 8 червей
...
```

Игрок

Нужен класс для представления каждого игрока, чтобы отслеживать их карты и количество выигранных ими раундов.

Python_ex272.py

```
class Player:
    def __init__(self, name):
        self.wins = 0
        self.card = None
        self.name = name
```

У класса `Player` есть три переменных экземпляра: `wins` для отслеживания количества раундов, выигранных игроком; `card` для представления карты, которую в данный момент держит игрок; `name` для отслеживания имени игрока.

Игра

Наконец, вы создаете класс, представляющий игру.

Python_ex273.py

```
class Game:
    def __init__(self):
        name1 = input("имя игрока 1: ")
        name2 = input("имя игрока 2: ")
        self.deck = Deck()
        self.p1 = Player(name1)
        self.p2 = Player(name2)
```

```
    def wins(self, winner):
        w = "{} забирает карты"
        w = w.format(winner)
        print(w)

    def draw(self, p1n, p1c, p2n, p2c):
        d = "{} кладет {}, а {} кладет {}"
        d = d.format(p1n, p1c, p2n, p2c)
        print(d)

    def play_game(self):
        cards = self.deck.cards
        print("Поехали!")
        while len(cards) >= 2:
            m = "Нажмите X для выхода. Нажмите любую другую
            клавишу для начала игры."
            response = input(m)
            if response == 'X':
                break
            p1c = self.deck.rm_card()
            p2c = self.deck.rm_card()
            p1n = self.p1.name
            p2n = self.p2.name
            self.draw(p1n, p1c, p2n, p2c)
            if p1c > p2c:
                self.p1.wins += 1
                self.wins(self.p1.name)
            else:
                self.p2.wins += 1
                self.wins(self.p2.name)

        win = self.winner(self.p1, self.p2)

        print("Игра окончена. {} выиграл!".format(win))

    def winner(self, p1, p2):
        if p1.wins > p2.wins:
            return p1.name
        if p1.wins < p2.wins:
            return p2.name
        return "Ничья!"
```

Когда вы создаете объект игры, Python вызывает метод `__init__`, а функция `input` принимает имена двух игроков и сохраняет их в переменных `name1` и `name2`. Затем вы создаете новый объект `Deck`, сохраняете его в переменной экземпляра класса `deck` и создаете два объекта `Player`, используя имена в `name1` и `name2`.

Метод `play_game` в классе `Game` начинает игру. В методе есть цикл, который продолжает ведение игры до тех пор, пока в колоде остается две или более карты, и пока переменная `response` не примет значение `X`. При каждом прохождении цикла вы назначаете переменную `response` пользовательскому вводу. Игра продолжается до тех пор, пока либо пользователь не введет `"X"`, либо в колоде останется менее двух карт.

При каждом прохождении цикла вынимаются две карты, и метод `play_game` присваивает первую карту `p1`, а вторую — `p2`. Затем он выводит имя каждого игрока и карты, которую он вынул, сравнивает две карты, выясняя, какая из них старше, увеличивает значение переменной экземпляра `wins` для игрока со старшей картой и выводит сообщение о том, кто победил.

У класса `Game` также есть метод `winner`, который принимает два объекта игроков, «смотрит» на количество раундов, выигранных каждым игроком, и возвращает игрока, который выиграл больше раундов.

Когда в объекте `Deck` заканчиваются карты, метод `play_game` выводит сообщение о том, что игра окончена, вызывает метод `winner` (передавая в него `p1` и `p2`) и выводит сообщение с итогом игры — именем победившего игрока.

«Пьяница»

Ниже приведен полный код игры.

Python_ex274.py

```
from random import shuffle

class Card:
    suits = ["пикей",
             "червей",
             "бубей",
             "треф"]

    values = [None, None,"2", "3",
              "4", "5", "6", "7",
              "8", "9", "10",
              "валета", "даму",
              "короля", "туза"]

    def __init__(self, v, s):
        """suit и value — целые числа"""
```

```
14          self.value = v
15          self.suit = s

16      def __lt__(self, c2):
17          if self.value < c2.value:
18              return True
19          if self.value == c2.value:
20              if self.suit < c2.suit:
21                  return True
22              else:
23                  return False
24          return False

25      def __gt__(self, c2):
26          if self.value > c2.value:
27              return True
28          if self.value == c2.value:
29              if self.suit > c2.suit:
30                  return True
31              else:
32                  return False
33          return False

34      def __repr__(self):
35          v = self.values[self.value] + " " \
36          + self.suits[self.suit]
37          return v

38  class Deck:
39      def __init__(self):
40          self.cards = []
41          for i in range(2, 15):
42              for j in range(4):
43                  self.cards.append(Card(i, j))
44          shuffle(self.cards)

45      def rm_card(self):
46          if len(self.cards) == 0:
47              return
48          return self.cards.pop()
```

```
50 class Player:
51     def __init__(self, name):
52         self.wins = 0
53         self.card = None
54         self.name = name

55 class Game:
56     def __init__(self):
57         name1 = input("имя игрока 1: ")
58         name2 = input("имя игрока 2: ")
59         self.deck = Deck()
60         self.p1 = Player(name1)
61         self.p2 = Player(name2)

62     def wins(self, winner):
63         w = "{} забирает карты"
64         w = w.format(winner)
65         print(w)

66     def draw(self, p1n, p1c, p2n, p2c):
67         d = "{} кладет {}, а {} кладет {}"
68         d = d.format(p1n, p1c, p2n, p2c)
69         print(d)

70     def play_game(self):
71         cards = self.deck.cards
72         print("Поехали!")
73         while len(cards) >= 2:
74             m = "Нажмите X для выхода. Нажмите любую другую
                   клавишу для начала игры."
75             response = input(m)
76             if response == 'X':
77                 break
78             p1c = self.deck.rm_card()
79             p2c = self.deck.rm_card()
80             p1n = self.p1.name
81             p2n = self.p2.name
82             self.draw(p1n, p1c, p2n, p2c)
83             if p1c > p2c:
84                 self.p1.wins += 1
```

```
85                  self.wins(self.p1.name)
86              else:
87                  self.p2.wins += 1
88                  self.wins(self.p2.name)
89          win = self.winner(self.p1, self.p2)
90          print("Игра окончена.{} выиграл!".format(win))
91      def winner(self, p1, p2):
92          if p1.wins > p2.wins:
93              return p1.name
94          if p1.wins < p2.wins:
95              return p2.name
96          return "Ничья!"
97  game = Game()
98  game.play_game()
```

```
>> "имя игрока 1: "
…
```

ЧАСТЬ III

Введение в инструменты программирования

Глава 16. bash

Я не могу представить себе работу, которую предпочел бы компьютерному программированию. Целый день из бесформенной пустоты вы создаете шаблоны и структуры, по пути решая десятки мелких головоломок.

Питер Ван Дер Линден

В этой главе вы научитесь пользоваться **интерфейсом командной строки** под названием **bash**. Интерфейс командной строки — это программа, в которую вы вводите инструкции для выполнения вашей операционной системой. bash — частный случай интерфейса командной строки, и он идет в комплекте с большинством Unix-подобных операционных систем. Впредь я буду использовать термины «интерфейс командной строки» и «**командная строка**» как взаимозаменяемые.

Когда я получил свою первую работу программиста, я допустил ошибку, начав посвящать все свое время практике программирования. Конечно, чтобы программировать профессионально, вы должны быть талантливым программистом. Но у вас также должно быть множество других навыков — например, знание командной строки. Командная строка — это «центр управления» всем, чем вы будете заниматься, не относящимся к написанию кода.

К примеру, позднее в этой книге вы узнаете, как пользоваться системами управления пакетами, чтобы устанавливать программы других людей, а также системами управления версиями для взаимодействия с другими программистами. Вы будете управлять обоими этими инструментами из командной строки. К тому же большинство современного программного обеспечения использует данные из всего Интернета, а большинство мировых веб-серверов работает под Linux. У этих серверов нет пользовательских интерфейсов, вы можете пользоваться ими лишь посредством командной строки.

Командная строка, системы управления пакетами, регулярные выражения и управление версиями — основные инструменты в арсенале программиста. Все, с кем я работал в команде, были экспертами в этих вещах.

Когда вы программируете профессионально, от вас ожидают, что помимо прочего вы будете хорошо ими владеть. У меня ушло много времени, чтобы понять это, и я жалею, что не начал изучение перечисленных методов раньше.

Выполнение примеров

Если вы пользуетесь Ubuntu или Unix, на вашем компьютере уже есть bash. Однако в операционной системе Windows вместо bash поставляется интерфейс **командная строка** (Command Prompt), который нельзя использовать в этой главе. В 64-х разрядной версии Windows 10 с обновлением не ниже Anniversary Update присутствует bash. Инструкции по использованию bash в Windows 10 можно найти на странице **www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/**.

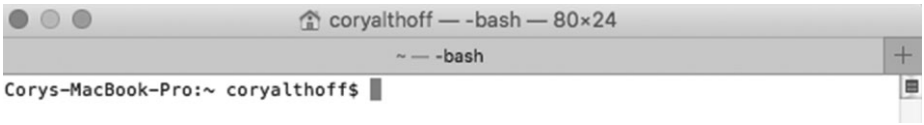
Если у вас другая версия Windows, можете использовать инфраструктуру Amazon AWS для настройки бесплатного веб-сервера под Ubuntu. Операция по настройке сервера проста, а AWS широко используется в программистской среде, так что это снабдит вас ценным опытом. Для дополнительной информации посетите сайт **aws.amazon.com/ru/getting-started/**.

Если на вашем компьютере установлена операционная система Windows, но вы не хотите настраивать сервер, вы можете выполнять примеры в веб-приложении, эмулирующем bash. Ссылка на приложение находится по адресу **www.tutorialspoint.com/execute_bash_online.php**.

В главах 17–18 вы сможете продолжать изучение, используя командную строку Windows. Найти ее можно, введя команду `cmd.exe` в диалоговом окне **Выполнить** (Run).

Запуск bash

Чтобы найти bash на компьютере, выполните поиск по запросу «терминал», щелкнув мышью по значку «Поиск на компьютере и в Интернете» (в Ubuntu), или при помощи поиска Spotlight (в macOS).



Команды

Интерфейс bash похож на оболочку Python. В bash вы набираете команды, аналогичные функциям в Python. Затем вы вводите пробел и указываете параметры, которые хотите передать в команду (если они есть). Нажимаете клавишу **Enter**, и bash возвращает результат. Команда `echo` похожа на функцию `print` в Python.

Всякий раз, когда в этой книге или документации вы видите символ `$`, за которым следует команда, это значит, что в командную строку нужно ввести команду.

bash_ex01.sh

```
$ echo Привет, мир!

>> Привет, мир!
```

Сначала вы набрали в bash команду `echo`, пробел после нее и в качестве параметра — `Привет, мир!`. Когда вы нажали клавишу **Enter**, в окне bash появилась строка `Привет, мир!`.

Из командной строки можно использовать установленные вами программы (например, Python). Введите команду `python3` (веб-приложение bash не поддерживает Python 3, поэтому введите `python` для использования Python 2).

bash_ex02.sh

```
$ python3
```

Теперь вы можете выполнять код Python.

bash_ex03.sh

```
print("Привет, мир!")

>> Привет, мир!
```

Введите команду `exit()`, чтобы выйти из Python.

Последние команды

В bash можно просматривать последние команды, нажимая клавиши ↑ и ↓. Для просмотра списка всех последних команд воспользуйтесь командой `history`.

bash_ex04.sh

```
$ history
```

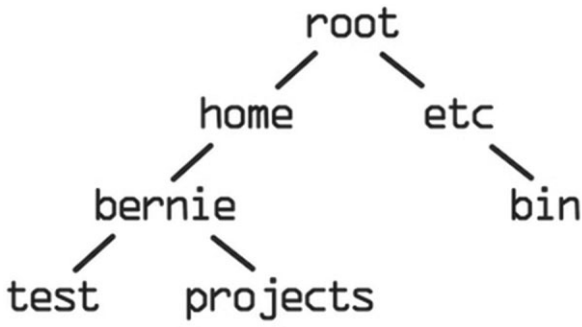
Относительные и абсолютные пути

Операционная система состоит из каталогов и файлов. **Каталог** — синоним слова «папка». У всех каталогов и файлов есть путь — адрес каталога или файла в операционной системе. Когда вы используете bash, то всегда находитесь в каталоге с конкретным путем. Для вывода имени вашего **рабочего каталога** воспользуйтесь командой `pwd` (сокращение от **print working directory** — вывод рабочего каталога). Рабочий каталог — это ваш текущий каталог.

bash_ex05.sh

```
$ pwd

>> /Users/coryalthoff
```

Операционная система представляет свои каталоги, в том числе местонахождение вашего каталога, в виде дерева. В информатике «дерево» является важной концепцией, называемой «структура данных» (об этом поговорим в части IV). В этом дереве корень располагается сверху. У корня есть ветви, у каждой из этих ветвей есть свои ветви, а у тех — свои, и так до бесконечности. Следующее изображение служит примером дерева, представляющего каталоги в операционной системе.



Каждая ветвь дерева, включая корень, представляет каталог. Дерево демонстрирует, как каталоги связаны друг с другом. При использовании bash вы находитесь в определенном месте в дереве вашей операционной системы. **Путь** — способ представить это место. Есть два способа представить путь к файлу или каталогу в Unix-подобной операционной системе — **абсолютный путь и относительный путь**.

Абсолютный путь указывает на место файла или каталога начиная с корневого каталога. Абсолютный путь состоит из отделенных слешами имен каталогов в дереве в порядке приближения их к корню. Абсолютный путь к каталогу *bernie* (см. рисунок выше) такой: */ home/ bernie*. Первый слеш представляет корневой каталог, за ним следует каталог *home*, затем еще один слеш и каталог *bernie*.

Другим способом указания места на компьютере является относительный путь. Относительный путь начинается не с корневого каталога, а с текущего рабочего каталога. Если ваш путь не начинается со слеша, bash понимает, что вы используете относительный путь. Если бы вы находились в каталоге *home* (см. рисунок выше), относительный путь к каталогу *projects* был бы *bernie/ projects*. Если бы вы находились в каталоге *home*, относительный путь к *bernie* был бы просто *bernie*. Если бы вы находились в каталоге *root*, относительный путь к *projects* был бы *home/ bernie/ projects*.

Навигация

По каталогам можно перемещаться, передавая абсолютный или относительный путь в команду `cd` в качестве параметра. Введите команду `cd`, а после нее — абсолютный путь `/`, чтобы перейти в корневой каталог вашей операционной системы.

bash_ex06.sh

```
$ cd /
```

Узнать текущее местоположение можно при помощи команды `pwd`.

bash_ex07.sh

```
$ pwd

>> /
```

Команда `ls` выводит каталоги и папки в текущем рабочем каталоге.

bash_ex08.sh

```
$ ls

>> bin dev initrd.img lost+found ...
```

Вы можете создать новый каталог, передав имя желаемого каталога в команду `mkdir`. Имена каталогов не должны содержать пробелы. Переместитесь в свой домашний каталог (~ — это сокращение для домашнего каталога в Unix-подобных операционных системах) и используйте команду `mkdir`, чтобы создать новый каталог *tstp*.

bash_ex09.sh

```
$ cd ~
$ mkdir tstp
```

Проверьте создание каталога при помощи команды `ls`.

bash_ex10.sh

```
$ ls

>> tstp
```

Теперь используйте команду `cd`, чтобы перейти в каталог `tstp`: передайте в нее в качестве параметра относительный путь к каталогу `tstp`.

bash_ex11.sh

```
$ cd tstp
```

Чтобы перейти на один каталог выше (на один уровень выше по дереву), после команды `cd` введите две точки.

bash_ex12.sh

```
$ cd ..
```

Каталог можно удалить при помощи команды `rmdir`. Воспользуйтесь ею, чтобы удалить каталог `tstp`.

bash_ex13.sh

```
$ rmdir tstp
```

Наконец, удостоверьтесь, что каталог был удален при помощи команды `ls`.

bash_ex14.sh

```
$ ls
```

Флаги

К командам применима концепция **флаги**, которая позволяет при выполнении команды изменять ее поведение. Флаги — это опции для команд, принимающие значения `True` или `False`. По умолчанию изначальное значение всех флагов — `False`. Если вы добавите флаг к команде, `bash` установит значение флага равным `True`, и поведение команды изменится. Чтобы изменить флаг на `True`, перед его именем укажите один (`-`) или два (`--`) дефиса (количество дефисов зависит от используемой операционной системы).

К примеру, вы можете добавить к команде `ls` флаг `–author`, чтобы присвоить флагу `author` значение `True`. Добавление этого флага к команде `ls` влияет на ее поведение. Когда вы добавляете этот флаг к команде `ls`, она выводит все каталоги и файлы в каталоге, но также выводит имя владельца — человека, их создавшего.

В Unix перед флагом добавляется один дефис.

bash_ex15.sh

```
$ ls –author

>> drwx------+ 13 coryalthoff 442B Sep 16 17:25 Pictures
>> drwx------+ 25 coryalthoff 850B Nov 23 18:09 Documents
```

А в Linux — два.

bash_ex16.sh

```
$ ls --author

>> drwx------+ 13 coryalthoff 442B Sep 16 17:25 Pictures
>> drwx------+ 25 coryalthoff 850B Nov 23 18:09 Documents
```

Скрытые файлы

Ваша операционная система, а также многие программы могут хранить данные в скрытых файлах. Скрытые файлы — это файлы, которые по умолчанию не видны пользователю, поскольку их изменение может повлиять на программы, зависящие от этих файлов. Имена скрытых файлов начинаются с точки, например *.hidden*. Скрытые файлы можно просмотреть, добавив флаг `–a` (от *англ.* слова **all** — все) к команде `ls`.

Команда `touch` создает новый файл. Используйте ее для создания скрытого файла *.self_taught*.

bash_ex17.sh

```
$ touch .self_taught
```

Проверьте, отображается ли этот файл, при помощи команд `ls` и `ls –a`.

Вертикальная черта

В Unix-подобных операционных системах символ вертикальной черты (`|`) на-

зывают «**пайп**». Вы можете использовать его для передачи вывода команды в другую команду в качестве ее ввода. Например, можно использовать вывод команды `ls` в качестве ввода команды `less` (убедитесь, что не находитесь в пустом каталоге).

bash_ex18.sh

```
$ ls | less

>> Applications ...
```

Результатом будет текстовый файл с выводом `ls`, открытым в программе `less` (введите символ `q` для выхода).

Переменные окружения

Переменные окружения хранятся в операционной системе и используются программами для получения данных об окружении, в котором они работают, — например, об имени компьютера, на котором запущена программа, или имени пользователя операционной системы, на которой она запущена. В bash новая переменная окружения создается при помощи синтаксиса `export` *`имя_переменной=значение_переменной`*. Чтобы сослаться на переменную окружения, нужно указать перед ее именем символ `$`.

bash_ex19.sh

```
$ export x=100
$ echo $x

>> 100
```

Переменная окружения, созданная таким образом, существует лишь в окне bash, в котором вы ее создали. Если вы закроете окно, в котором создали переменную окружения, затем заново откроете его и введете команду `echo $x`, bash не выведет `100`, поскольку переменной окружения `x` больше не существует.

Вы можете сохранить переменную окружения, добавив ее в скрытый файл *.profile*, расположенный в каталоге *home* в Unix-подобных операционных системах. Воспользуйтесь своим графическим интерфейсом пользователя, чтобы попасть в домашний каталог. Путь к домашнему каталогу можно узнать, введя `pwd ~` в командной строке. Воспользуйтесь текстовым редактором, чтобы создать файл *.profile*. Введите в первой строке файла `export x=100` и сохраните его. Закройте и заново откройте окно bash, и тогда вы сможете вывести переменную окружения `x`.

bash_ex20.sh

```
$ echo $x

>> 100
```

Переменная будет сохранена до тех пор, пока будет содержаться в файле *.profile*. Переменную можно удалить, стерев ее из файла *.profile*.

Пользователи

У операционных систем может быть множество пользователей. Пользователь — человек, использующий операционную систему. Каждому пользователю присвоены имя пользователя (логин) и пароль, позволяющие ему заходить в операционную систему и работать в ней. У каждого пользователя также есть определенные разрешения — набор операций, которые им позволено выполнять. При помощи команды `whoami` можно вывести имя пользователя операционной системы (примеры в этом разделе не будут работать в веб-приложении bash).

```
$ whoami
```

Обычно вы и есть тот пользователь, аккаунт которого создали при установке операционной системы. Но полномочия такого пользователя не самые широкие. Пользователь с наивысшим уровнем разрешений называется пользователем с правами root (пользователем root, суперпользователем). В каждой системе есть суперпользователь, который, к примеру, может создавать и удалять аккаунты других пользователей.

По соображениям безопасности обычно вы не входите в систему как суперпользователь. Вместо этого перед командами, которые вы хотите выполнить как суперпользователь, вы добавляете команду `sudo` (от англ. словосочетания superuser do — выполняет суперпользователь). Ключевое слово `sudo` позволяет выполнять команды от имени суперпользователя без необходимости входа в аккаунт суперпользователя, нанося ущерб безопасности вашей операционной системы. Ниже приведен пример использования команды `echo` с командой `sudo`.

```
$ sudo echo Привет, мир!

>> Привет, мир!
```

Если на вашем компьютере установлен пароль, вам будет предложено его ввести, когда вы используете команду с `sudo`. Команда `sudo` снимает защитные меры, которые оберегают операционную систему от нанесения вреда, поэтому никогда не выполняйте команду с `sudo`, если вы до конца не уверены, что эта команда не навредит вашей операционной системе.

Узнайте больше

В этой главе были описаны лишь основы bash. Чтобы узнать больше об использовании bash, прочитайте статью по адресу **habrahabr.ru/post/47163/**.

Словарь терминов

`$PATH`: когда вы вводите команду в bash, оболочка ищет эту команду во всех каталогах, хранящихся в переменной окружения `$PATH`.
bash: программа, встроенная в большинство Unix-подобных операционных систем, в которую вы вводите инструкции для выполнения вашей операционной системой.

Абсолютный путь: указывает на расположение файла или каталога, начиная с корневого каталога.
Вертикальная черта: символ `|`. В Unix-подобных операционных системах вы можете использовать его для передачи вывода команды в другую команду в качестве ее ввода.
Интерфейс командной строки: программа, в которую вы вводите инструкции для выполнения вашей операционной системой.
Каталог: другое название папки на компьютере.
Командная строка (в Windows): интерфейс командной строки, встроенный в операционную систему Windows.
Командная строка: другое название интерфейса командной строки.
Относительный путь: указывает на расположение файла или каталога, начиная с текущего рабочего каталога.
Переменные окружения: переменные, в которых хранит данные операционная система, а также другие программы.
Пользователь: человек, пользующийся операционной системой.
Путь: способ представить место файла или каталога в операционной системе.
Рабочий каталог: текущий каталог, в котором вы находитесь.
Разрешения: операции, которые разрешено выполнять пользователям.
Суперпользователь (пользователь с правами root): пользователь с наивысшим уровнем полномочий.

Практикум

1. Выведите в bash `Self-taught`.
2. Переместитесь в свой домашний каталог из другого каталога, используя абсолютный и относительный путь.
3. Создайте переменную окружения `$python_projects`, являющуюся абсолютным путем к каталогу, где находятся ваши файлы Python. Сохраните переменную в файле *.profile*, а затем воспользуйтесь командой `cd $python_projects`, чтобы попасть туда.

Решения: *challenge1.sh – challenge3.sh*.

Глава 17. Регулярные выражения

Болтовня ничего не стоит. Покажите мне код.

Линус Торвальдс

Многие языки программирования и операционные системы поддерживают **регулярные выражения** — «последовательности символов, определяющие поисковый шаблон»[12]. Регулярные выражения могут быть полезны благодаря тому, что их можно использовать для выполнения поиска по сложным запросам в файлах или иных данных. К примеру, вы можете использовать регулярное выражение для поиска всех чисел в файле. В этой главе вы научитесь определять и передавать регулярные выражения команде `grep`, используемой в Unix-подобных

[12] ru.wikipedia.org/wiki/Регулярные_выражения

операционных системах, которая ищет совпадения в файле и возвращает найденный текст, соответствующий шаблону. Вы также научитесь использовать регулярные выражения для поиска по шаблону в строках в Python.

Настройка

Для начала создайте файл *zen.txt*. В командной строке (убедитесь, что вы находитесь в каталоге с файлом *zen.txt*) введите команду `python3 -c "import this"`. Команда выведет текст поэмы «Дзен Пайтона» Тима Питерса. Ниже показан ее перевод на русский язык.

Дзен Пайтона
Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Плоское лучше, чем вложенное.
Разреженное лучше, чем плотное.
Читаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
При этом практичность важнее безупречности.
Ошибки никогда не должны замалчиваться.
Если не замалчиваются явно.
Встретив двусмысленность, отбрось искушение угадать.
Должен существовать один — и, желательно, только один — очевидный способ сделать это.
Хотя он поначалу может быть и не очевиден, если вы не голландец.
Сейчас лучше, чем никогда.
Хотя никогда зачастую лучше, чем прямо сейчас.
Если реализацию сложно объяснить — идея плоха.
Если реализацию легко объяснить — идея, возможно, хороша.
Пространства имен — отличная штука! Будем делать их побольше!

Флаг `-c` сообщает Python, что вы собираетесь передать строку, содержащую код Python. Затем Python выполняет код. Результатом выполнения строки `import this` становится вывод текста «Дзен Пайтона» (сообщение, спрятанное в коде, вроде этой философии, называется **пасхальным яйцом**). Введите в bash команду `exit()`, чтобы выйти из Python. Используйте документ *zen.txt*, прилагаемый в качестве файлов примеров для этой книги.

По умолчанию в Ubuntu команда `grep` выводит совпавшие слова, подсвечивая их красным цветом, но в Unix этого не происходит. Если вы работаете в среде macOS, то можете изменить это поведение, установив в bash следующие переменные окружения:

bash_ex21.sh

```
$ export GREP_OPTIONS='--color=always'
$ export GREP_OPTIONS='--color=always'
```

Запомните, что установка переменных окружения в bash происходит на временной основе, так что если вы завершите работу bash, вам придется настраивать переменные окружения заново при следующем запуске. Вы можете добавить переменные окружения в ваш файл *.profile*, чтобы сохранить их на постоянной основе.

Простое совпадение

Команда `grep` принимает два параметра — регулярное выражение и путь к файлу для поиска содержимого по шаблону, определенному в регулярном выражении. Простейшим видом шаблона является простое совпадение, строка слов, совпадающая с такой же строкой слов. Чтобы увидеть пример простого совпадения, введите следующую команду в каталоге, в котором вы создали файл *zen.txt*.

bash_ex22.sh

```
$ grep Красивое zen.txt

>> Красивое лучше, чем уродливое.
```

В выполненной вами команде первый параметр, `Красивое`, — это регулярное выражение, а второй параметр, `zen.txt`, — путь к файлу, в котором нужно искать регулярное выражение. Программа bash вывела строку `Красивое лучше, чем уродливое` и выделила слово `Красивое` красным цветом шрифта, поскольку это то слово, которое совпало с регулярным выражением.

Если в предыдущем примере вы измените регулярное выражение `Красивое` на `красивое`, команда `grep` не найдет совпадения.

bash_ex23.sh

```
$ grep красивое zen.txt
```

Чтобы не учитывать регистр букв, добавьте флаг `-i`.

bash_ex24.sh

```
$ grep -i красивое zen.txt

>> Красивое лучше, чем уродливое.
```

По умолчанию команда `grep` выводит всю строку файла, в котором она нашла совпадение. Чтобы выводить только конкретные слова, совпадающие с переданным шаблоном, добавьте флаг `-o`.

bash_ex25.sh

```
$ grep -o Красивое zen.txt

>> Красивое
```

Вы можете использовать регулярные выражения в Python при помощи встроенной библиотеки `re` (от англ. словосочетания regular expressions — регулярные выражения). В модуле `re` есть метод `findall`. В качестве параметров в него пере-

дается регулярное выражение, затем строка, и он возвращает список со всеми элементами в строке, которые совпадают с шаблоном.

Python_ex275.py

```
import re

l = "Красивое лучше, чем уродливое."

matches = re.findall("Красивое", l)

print(matches)
```

```
>> ['Красивое']
```

В этом примере метод `findall` нашел совпадение и вернул список с совпадением (Красивое) в качестве первого элемента.

Вы можете не учитывать регистр в методе `findall`. Для этого передайте в него значение `re.IGNORECASE` в качестве третьего параметра.

Python_ex276.py

```
import re

l = "Красивое лучше, чем уродливое."

matches = re.findall("красивое",
                     l,
                     re.IGNORECASE)

print(matches)
```

```
>> ['Красивое']
```

Совпадение в начале и в конце

Вы можете создавать регулярные выражения со сложными шаблонами, добавив в них особые символы, которые не используются для поиска совпадения, а определяют правило. Например, для создания регулярного выражения, которое ищет шаблон, только если он встречается в начале строки, можно использовать символ каретки (^).

bash_ex26.sh

```
$ grep ^Если zen.txt

>> Если не замалчиваются явно.
>> Если реализацию сложно объяснить — идея плоха.
>> Если реализацию легко объяснить — идея, возможно, хороша.
```

Аналогично можно использовать символ доллара (`$`), чтобы искать совпадения только в конце строки.

bash_ex27.sh

```
$ grep никогда.$ zen.txt

>> Сейчас лучше, чем никогда.
```

В этом случае команда `grep` проигнорировала строки `Ошибки никогда не должны замалчиваться.` и `Хотя никогда зачастую лучше, чем прямо сейчас.`, так как они хоть и содержат слово никогда, но не заканчиваются им.

Ниже приведен пример использования символа каретки (`^`) в Python (вы должны передать значение `re.MULTILINE` в качестве третьего параметра методу `findall`, чтобы искать совпадения во всех строках многострочного экрана).

Python_ex277.py

```
import re

zen = """Хотя никогда зачастую
лучше, чем прямо сейчас.
Если реализацию сложно
объяснить — идея плоха.
Если реализацию легко объяснить
— идея, возможно, хороша.
Пространства имен — отличная
штука! Будем делать их побольше!
"""

m = re.findall("^Если",
               zen,
               re.MULTILINE)

print(m)
```

```
>> ['Если', 'Если']
```

Поиск совпадений с несколькими символами

Чтобы определить шаблон для поиска совпадений с несколькими символами, поместите эти символы внутрь квадратных скобок в регулярном выражении. Если вы укажете значение `[абв]`, регулярное выражение будет искать совпадения с `а`, `б` или `в`. В следующем примере вместо поиска совпадений в вашем файле *zen.txt*, вы будете искать их в строке, которую добавите к команде `grep` при помощи вертикальной черты.

bash_ex28.sh

```
$ echo Два даа. | grep  -i д[ав]а

>> Два даа
```

Вывод команды `echo` передается в `grep` в качестве ввода и, следовательно, вам не нужно указывать файл в качестве параметра для `grep`. Команда выводит и `Два`, и `даа`, поскольку регулярное выражение находит совпадение с буквой `д`, затем либо с `а`, либо с `в`, и затем — с `а`.

В Python это делается так:

Python_ex278.py

```
1 import re

2 string = " Два даа."

3 m = re.findall("д[ав]а",
4                string,
5                re.IGNORECASE)

6 print(m)
```

```
>> ['Двас, 'даа']
```

Совпадения цифр

С помощью значения `[[:digit:]]` можно искать совпадения цифр.

bash_ex29.sh

```
$ echo 123 хай 34 привет. | grep [[:digit:]]

>> 123 хай 34 привет.
```

В Python для этого используются символы `\d`.

Python_ex279.py

```
1 import re

2 line = "123?34 привет?"

3 m = re.findall("\d",
4                line,
5                re.IGNORECASE)

6 print(m)
```

```
>> ['1', '2', '3', '3', '4']
```

Повторение

Символ звездочки (*) добавляет в регулярные выражения повторение. При его помощи «предшествующий элемент будет найден ноль или более раз»[13]. К примеру, используя звездочку, можно найти совпадения с ту, за которым следует любое количество букв у.

bash_ex30.sh

```
$ echo ту тууу ура. | grep -о ту*

>> ту
>> тууу
```

В регулярных выражениях точка соответствует любому символу. Если вы укажете звездочку после точки, будет выполняться поиск совпадения с любым символом ноль или более раз. Можно использовать точку со звездочкой для поиска всего, что находится в промежутке между двумя символами.

bash_ex31.sh

```
$ echo __привет__всем | grep -о __.*__

>> __привет__
```

Регулярное выражение __.*__ находит все символы между двумя двойными подчеркиваниями, включая сами подчеркивания. Символ звездочки является **жадным** — это значит, что он попытается найти столько текста, сколько сможет. Если вы добавите больше слов с двойными подчеркиваниями, регулярное выражение из предыдущего примера найдет все от первого до последнего подчеркивания.

bash_ex32.sh

```
$ echo __привет__пока__чао__! | grep -о __.*__

>> __привет__пока__чао__
```

Не всегда нужно искать шаблоны жадным образом. Чтобы сделать регулярное выражение **нежадным**, можно указать после звездочки вопросительный знак. Нежадное регулярное выражение ищет наименьшее возможное количество совпадений. В данном случае такое выражение остановилось бы на первом двойном подчеркивании, а не продолжило бы искать совпадения между самым первым и самым последним подчеркиваниями. Для команды grep недоступен нежадный поиск, но в Python для этого можно использовать знак вопроса.

Python_ex280.py

```
1 import re
2 t = "__один__ __два__ __три__"
3 found = re.findall("__.*?__", t)
```

[13] tldp.org/LDP/Bash-Beginners-Guide/html/sect_04_01.html

```
    for match in found:
        print(match)
```

```
>> __один__
>> __два__
>> __три__
```

С помощью «нежадного» поиска на Python можно создать игру «Чепуха» (напомню, что это игра, где игроку предлагается вставить пропущенные в абзаце текста слова).

Python_ex281.py

```
import re

text = """Жирафы любят таскать
различные __СУЩЕСТВИТЕЛЬНОЕ ВО МНОЖЕСТВЕННОМ ЧИСЛЕ__
целый день напролет. Жирафы
также славятся тем, что поедают
прекрасные __СУЩЕСТВИТЕЛЬНОЕ ВО МНОЖЕСТВЕННОМ ЧИСЛЕ__, но
после этого у них часто
болит __ЧАСТЬ_ТЕЛА__. Если же
жирафы находят __ЧИСЛО__
__СУЩЕСТВИТЕЛЬНОЕ ВО МНОЖЕСТВЕННОМ ЧИСЛЕ__, у
них моментально отваливается __ЧАСТЬ ТЕЛА__.
"""

def mad_libs(mls):
    """
    :param mls: В строках
    пользовательский ввод
    должен быть окружен двойными
    подчеркиваниями. Подчеркивания
    нельзя вставлять в подсказку:
    __подсказка_подсказка__ (нельзя);
    __подсказка__ (можно).
    """
    hints = re.findall("__.*?__",
                       mls)

    if hints is not None:
        for word in hints:
            q = "Введите {}".format(word)
```

```
            new = input(q)
            mls = mls.replace(word, new, 1)
        print('\n')
        mls = mls.replace("\n", "")
        print(mls)
    else: print("ошибка ввода")
        mad_libs(text)
```

```
>> Введите __СУЩЕСТВИТЕЛЬНОЕ ВО МНОЖЕСТВЕННОМ ЧИСЛЕ__
```

В данном примере вы используете метод `re.findall`, чтобы получить список всех слов в переменной `text`, окруженных двойными подчеркиваниями (каждое из них подсказывает тип необходимого слова). Затем вы перебираете список и используете каждую подсказку, предлагая пользователю программы вставить новое слово. После этого вы создаете новую строку, заменяя подсказку вставленным пользователем словом. Как только цикл завершается, выводится новая строка со всеми словами, полученными от пользователя.

Управляющие символы

В регулярных выражениях вы можете экранировать символы (игнорировать их значение, просто находя совпадения) так же, как вы делали в случае со строками в Python, указывая обратный слеш \ перед регулярным выражением.

bash_ex33.sh

```
$ echo Я люблю $ | grep  \$
```

```
>> Я люблю $
```

Обычно символ `$` означает, что совпадение допустимо, только если оно находится в конце строки, но поскольку вы экранировали этот знак, регулярное выражение просто найдет совпадение.

Этот же пример в Python:

Python_ex282.py

```
import re

line = "Я люблю $"

m = re.findall("\\$",
               line,
               re.IGNORECASE)

print(m)
```

```
>> ['$']
```

Инструмент для создания регулярных выражений

Процесс создания регулярного выражения для поиска шаблона может отнимать много сил. Посетите сайты **regexr.com** и **www.tutorialspoint.com/python/python_reg_expressions.htm** и познакомьтесь с инструментами, которые помогут вам создавать идеальные регулярные выражения.

Словарь терминов

Жадный: регулярное выражение, которое будет с жадностью пытаться найти столько совпадений, сколько сможет.
Нежадный: нежадное регулярное выражение находит наименьшее количество совпадений.
Пасхальное яйцо: сообщение, спрятанное в коде.
Регулярное выражение: «последовательность символов, определяющая поисковый шаблон»[14].

Практикум

1. Напишите регулярное выражение, которое находит слово `голландец` в «Дзене Пайтона».
2. Создайте регулярное выражение, которое находит все цифры в строке `Москва: 777, 999, 797. Тула: 071, 950, 112.`.
3. Создайте регулярное выражение для нахождения любого слова, которое начинается с произвольного символа, предшествующего буквам `ло`. Затем используйте модуль `re` из Python, чтобы найти сочетания `ало` и `зло` в предложении `Привидение прошуршало и и исчезло в углу`.

Решения: *challenge_1.sh, challenge_2.sh, challenge_3.py и zen.txt.*

Глава 18. Системы управления пакетами

Каждый программист — это автор.

Серкан Лейлек

Система управления пакетами (менеджер пакетов) — это программа, которая устанавливает другие программы и управляет ими. Она может быть полезна, поскольку при создании нового программного обеспечения часто приходится пользоваться другими программами. Например, веб-разработчики часто используют веб-фреймворк — программу, позволяющую создать веб-сайт. Программисты используют системы управления пакетами для установки **веб-фреймворков** и множества других программ. В этой главе вы научитесь использовать менеджер пакетов **pip**.

Пакеты

Пакет — это программное обеспечение, «упакованное» для распространения. Пакет включает файлы, из которых собственно состоит программа, и **метадан-**

[14] ru.wikipedia.org/wiki/Регулярные_выражения

ные — информацию о данных вроде названия программного обеспечения, номера версии и **зависимостей** (программ, от которых зависит должная работа вашей программы). Для того чтобы загрузить и установить пакет на компьютер как программу, можно воспользоваться системой управления пакетами. Система управления пакетами управляет загрузкой всех зависимостей пакета.

Pip

В этом разделе я покажу вам, как использовать pip, менеджер пакетов для Python, чтобы загрузить пакеты Python. Как только вы загрузите пакет при помощи pip, вы сможете импортировать его в Python как модуль. Для начала проверьте, установлен ли pip на вашем компьютере, открыв bash (или командную строку Windows) и введя команду `pip`.

bash_ex34.sh

```
$ pip

>> Usage: pip <command> [options]
Commands:
install Install packages.
download Download packages. ...
```

После ввода команды должен отобразиться синтаксис команды с доступными параметрами. Инструмент pip поставляется в комплекте с новыми версиями Python, но в ранних версиях он отсутствовал. Если после ввода команды `pip` ничего не происходит, значит, на вашем компьютере pip не установлен. Ознакомьтесь с инструкциями по его установке на сайте **pythonworld.ru/osnovy/pip.html**.

Новый пакет можно установить при помощи команды `pip install` *`имя_пакета`*. Pip устанавливает новые пакеты в папку *site-packages* в вашем каталоге Python. Список всех доступных для загрузки пакетов Python вы можете найти на сайте **pypi.python.org/pypi**. Существуют два способа указать, какой пакет вы хотите загрузить, — указав только имя пакета, или имя пакета, за которым следуют два знака равенства (==) и желаемый номер версии. Если вы используете только имя пакета, pip загрузит последнюю версию пакета. Второй вариант позволяет загрузить конкретную версию пакета вместо наиболее актуальной. Ниже показано, как установить `Flask`, пакет Python для создания сайтов на Ubuntu и Unix.

bash_ex35.sh

```
$ sudo pip install Flask==0.11.1

>> Пароль:
>> Successfully installed flask-0.11.1
```

В Windows вам нужно воспользоваться командной строкой от имени администратора. Щелкните правой кнопкой мыши по ярлыку инструмента команд-

ной строки и выберите команду меню **Запуск от имени администратора** (Run as administrator).

В командной строке Windows введите следующее.

bash_ex36.sh

```
C:\Python36\Scripts\pip3.exe install Flask==0.11.1

>> Successfully installed flask-0.11.1
```

При помощи этой команды pip устанавливает модуль `Flask` в папку *site-packages* на вашем компьютере. Обратите внимание, что в вашем случае путь к файлу *pip3.exe* может быть иным!

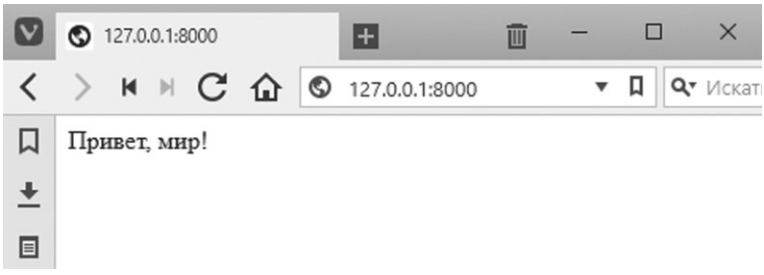
Теперь можете импортировать модуль `Flask` в программу. Создайте новый файл Python, добавьте следующий код и запустите программу.

Python_ex283.py

```
1 from flask import Flask

2 app = Flask(__name__)

3 @app.route('/')
4 def index():
5     return "Привет, мир!"

6 app.run(port='8000')
```

```
>> * Running on http://127.0.0.1:8000/ (Press CTRL+C to quit)
```

Теперь в браузере перейдите по адресу **http://127.0.0.1:8000/** — вы должны увидеть сайт с сообщением `Привет, мир!`.



Модуль `Flask` позволяет быстро создать веб-сервер и веб-сайт. Посетите сайт **flask.pocoo.org/docs/0.11/tutorial**, чтобы узнать больше о том, как работает этот пример.

С помощью команды `pip freeze` можно просматривать перечень установленных пакетов.

bash_ex37.sh

```
pip freeze

>> Flask==0.11.11
...
```

Наконец, программу можно удалить с помощью команды `pip uninstall` *`имя_пакета`*. Удалите `Flask` следующей командой:

bash_ex38.sh

```
sudo pip uninstall flask

>> Proceed (y/n)? y
  Successfully uninstalled Flask-0.11.1
```

Теперь модуль `Flask` удален, в чем можно убедиться при помощи команды `pip freeze`.

Виртуальные окружения

В конечном итоге вам захочется устанавливать пакеты Python в **виртуальное окружение** вместо помещения в папку *site-packages*. Виртуальные окружения позволяют раздельно хранить пакеты Python для разных проектов. Чтобы узнать о виртуальных окружениях больше, перейдите по ссылке **docs.pythonguide.org/en/latest/dev/virtualenvs**.

Словарь терминов

`$PYTHONPATH`: Python ищет модули в списке папок, который хранится в переменной окружения `$PYTHONPATH`.
Apt-get: менеджер пакетов, который поставляется с Ubuntu.
Pip: менеджер пакетов для Python.
PyPI: веб-сайт, который содержит пакеты Python.
Site-packages: папка в `$PYTHONPATH`. Эта папка — место, куда pip устанавливает пакеты.
Веб-фреймворк: программа, помогающая создавать веб-сайты.
Виртуальное окружение: позволяет хранить пакеты Python для разных программистских проектов раздельно.
Зависимости: программы, от которых зависит должная работа программы.
Метаданные: данные о данных.
Пакет: программное обеспечение, «упакованное» для распространения.
Система управления пакетами (менеджер пакетов): программа, устанавливающая другие программы и управляющая ими.

Практикум

Найдите пакет на PyPI (pypi.python.org) и загрузите его при помощи pip.

Решение: *challenge_1.sh*.

Глава 19. Управление версиями

Я отказываюсь делать то, что могут сделать компьютеры.

Олин Шиверс

Создание программного обеспечения — командный вид спорта. Когда вы работаете над проектом совместно с другим человеком (или в команде), вам нужно вносить изменения в **кодовую базу** — папки и файлы, из которых формируется ваше программное обеспечение, — при этом, изменения должны быть синхронизированы. Вы могли бы периодически слать друг другу электронные письма с изменениями и самостоятельно совмещать различные версии, но это было бы слишком утомительно.

Кроме того, что произойдет, если бы вы оба внесете изменения в одну и ту же часть проекта? Как решить, чьи изменения оставлять? Это именно те проблемы, которые решает система управления версиями. **Система управления версиями** — это программа, разработанная с целью помогать вам без труда сотрудничать с другими программистами при работе над проектами.

Git и **SVN** — популярные системы управления версиями. Обычно система управления версиями используется в сочетании с сервисом облачного хранения. В этой главе вы будете использовать Git, чтобы поместить программное обеспечение на **GitHub**, веб-сайт, хранящий код в облаке.

Репозитории

Репозиторий — это структура данных, созданная системой управления версиями, наподобие Git, которая отслеживает все изменения в программистском проекте. **Структура данных** — это способ организовывать и хранить информацию: списки и словари являются примерами структур данных (вы узнаете больше о структурах данных в части IV). На вид репозиторий напоминает каталог с файлами внутри. Вы будете использовать Git, чтобы взаимодействовать со структурой данных, отслеживающей изменения в проекте.

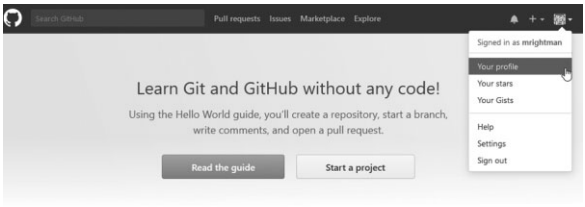
При работе над проектом, управляемым Git, будет создано множество репозиториев (как правило, по одному на каждого человека, принимающего участие в работе). Обычно у каждого человека, который работает над проектом, на компьютере есть репозиторий, называемый **локальным репозиторием** — он отслеживает все изменения, которые этот человек вносит в проект. Также есть **центральный репозиторий**, размещенный на веб-сайте вроде GitHub, с которым обмениваются информацией все локальные репозитории, чтобы поддерживать синхронизацию друг с другом (каждый репозиторий полностью самостоятелен). Программист, работающий над проектом, может обновлять центральный репозиторий согласно изменениям, которые он внес в свой локальный репозиторий, а также обновлять свой локальный репозиторий согласно последним изменениям, которые другие программисты внесли в центральный репозиторий. Если вы работаете над проектом с еще одним программистом, вся система будет выглядеть следующим образом.

Вы можете создать новый центральный репозиторий на сайте GitHub (или из командной строки). Как только вы создадите центральный репозиторий, сможете использовать Git для создания локального репозитория, который будет обмениваться с ним информацией.

Начинаем

Если GitHub изменит оформление сайта, инструкции в этом разделе также изменятся — в этом случае вы найдете новые инструкции на сайте **www.codeschool.com/learn/git**. Для начала вам понадобится создать учетную запись GitHub на странице **github.com/join**. Чтобы создать на GitHub новый репозиторий, войдите в свою учетную запись GitHub (как только создадите ее) и щелкните по кнопке **+** в правом верхнем углу экрана. В появившемся меню выберите пункт **New repository** (Новый репозиторий). Присвойте репозиторию название **hangman.** Установите переключатель в положение **Public** (Публичный) и установите флажок **Initialize the repository with a README** (Инициализировать репозиторий с помощью README-файла). Теперь нажмите кнопку **Create repository** (Создать репозиторий).

Щелкните мышью по кнопке учетной записи в правом верхнем углу и выберите пункт **Your profile** (Ваш профиль).



Вы увидите название своего репозитория — **hangman**. Щелкните по нему. Эта часть сайта — ваш центральный репозиторий. Видите кнопку **Clone Or Download** (Клонировать или скачать)? Когда нажмете на нее, появится ссылка. Сохраните эту ссылку.

Прежде чем продолжить, нужно установить Git. Инструкции по установке вы можете найти на сайте **www.git-scm.com/book/en/v2/Getting-Started-Installing-Git**.

Как только вы установите Git, его можно будет использовать из командной строки. Введите в командной строке команду `git`.

bash_ex39.sh

```
$ git

>> usage: git [--version] [--help] [-C <path>] [-c name=value]
   ...
```

Если ваш вывод такой же, как в этом примере, значит, вы справились с установкой Git.

Теперь при помощи команды `git clone` *ссылка_на_репозиторий* можно загрузить локальный репозиторий на компьютер (используйте сохраненную ранее ссылку). Репозиторий будет загружен в тот каталог, из которого вы выполняете команду. Скопируйте ссылку или выберите команду **Copy to clipboard** (Копировать в буфер обмена) и передайте ее в качестве параметра команде `git clone`.

bash_ex40.sh

```
$ git clone ссылка_на_репозиторий

>> Cloning into 'hangman'... remote: Counting objects: 3,
   done. remote: Total 3 (delta 0), reused 0 (delta 0), pack-
   reused 0 Unpacking objects: 100% (3/3), done. Checking
   connectivity... done.
```

Проверьте, загрузился ли локальный репозиторий с помощью `ls`.

bash_ex41.sh

```
$ ls

>> hangman
```

Должен отобразиться каталог `hangman`. Это ваш локальный репозиторий.

Помещение и извлечение данных

Прежде всего, с помощью Git вы сможете делать две вещи. Первая — обновлять свой центральный репозиторий, внося изменения из локального репозитория, то **есть помещать данные (толкать)**. Вторая — обновлять свой локальный репозиторий, внося изменения из центрального репозитория, то есть **получать данные (тянуть)**.

Команда `git remote -v` (этот часто используемый флаг служит для вывода дополнительной информации) выводит URL-адрес центрального репозитория, в который ваш локальный репозиторий помещает данные и из которого он их получает. Перейдите в свой каталог `hangman` и введите команду `git remote`.

bash_ex42.sh

```
$ cd hangman
$ git remote -v

>> origin ссылка_на_репозиторий/hangman.git (fetch)
>> origin ссылка_на_репозиторий/hangman.git (push)
```

Первая строка вывода — это URL-адрес центрального репозитория, из которого ваш проект будет получать данные, а вторая — URL-адрес центрального репозитория, в который ваш проект будет помещать данные. Скорее всего, в обоих случаях вы будете использовать один и тот же центральный репозиторий, поэтому обе ссылки будут одинаковыми.

Пример помещения данных

В этом разделе вы внесете изменение в созданный вами и скопированный на компьютер локальный репозиторий `hangman`, а затем поместите это изменение в свой центральный репозиторий, размещенный на GitHub.

Переместите ваш файл Python с кодом из задания, выполненного вами в конце части I, в каталог *hangman*. Теперь в вашем локальном репозитории находится файл, не существующий в центральном репозитории — он не синхронизирован с вашим центральным репозиторием. Эту проблему можно решить, поместив в центральный репозиторий изменения, внесенные в локальный репозиторий.

Этот процесс состоит из трех шагов. Сначала вы **индексируете** файлы — сообщаете Git, какие измененные файлы вы хотите отправить в центральный репозиторий.

Команда `git status` отображает текущее состояние вашего проекта относительно вашего репозитория, чтобы вы могли решить, какие файлы индексировать. Эта команда выводит те файлы в локальном репозитории, которые отличаются от файлов в центральном репозитории. Когда файл не проиндексирован, он выделяется красным, когда проиндексирован — зеленым. Убедитесь, что вы находитесь внутри каталога `hangman`, и введите команду `git status`.

bash_ex43.sh

```
$ git status

 >> On branch master Your branch is up-to-date with 'origin/
    master'. Untracked files: (use "git add <file>..." to in-
    clude in what will be committed)
 >> hangman.py
```

Файл `hangman.py` должен быть выделен красным. Проиндексировать файл можно при помощи команды `git add` *файл*.

bash_ex44.sh

```
$ git add hangman.py
```

Теперь используйте команду `git status`, чтобы убедиться, что вы проиндексировали файл.

bash_ex45.sh

```
$ git status

>> On branch master Your branch is up-to-date with 'origin/
   master'. Changes to be committed: (use "git reset HEAD
   <file>..." to unstage)

new file: hangman.py
```

Имя файла `hangman.py` теперь окрашено в зеленый цвет, поскольку вы проиндексировали его.

При помощи синтаксиса `git reset` *`путь_к_файлу`* вы можете отменить индексацию файла, не внося изменений в центральный репозиторий. Отмените индексацию файла `hangman.py` следующим образом.

bash_ex46.sh

```
$ git reset hangman.py
```

Убедитесь, что индексация была отменена с помощью команды `git status`.

```
>> On branch master Your branch is up-to-date with 'origin/
master'. Untracked files: (use "git add <file>..." to include
in what will be committed)

hangman.py
```

Повторно проиндексируйте файл.

bash_ex47.sh

```
$ git add hangman.py
$ git status

>>On branch master Your branch is up-to-date with 'origin/
  master'. Changes to be committed: (use "git reset HEAD
  <file>..."to unstage)

new file: hangman.py
```

Как только вы проиндексировали файлы, которые хотите поместить в центральный репозиторий, можно приступать к следующему шагу, **сохранению состояния** файлов – указанию Git записывать изменения, которые вы внесли в локальный репозиторий. Это делается с помощью синтаксиса `git commit -m` *`ваше_сообщение`*. Данная команда создает версию вашего проекта, сохраняемую Git. Флаг `-m` значит, что вы хотите добавить к состоянию сообщение, кото-

рое поможет вам запомнить вносимые изменения и их причины (это сообщение похоже на комментарий). В качестве следующего шага вы поместите свои изменения в центральный репозиторий на GitHub, где и сможете увидеть свое сообщение.

bash_ex48.sh

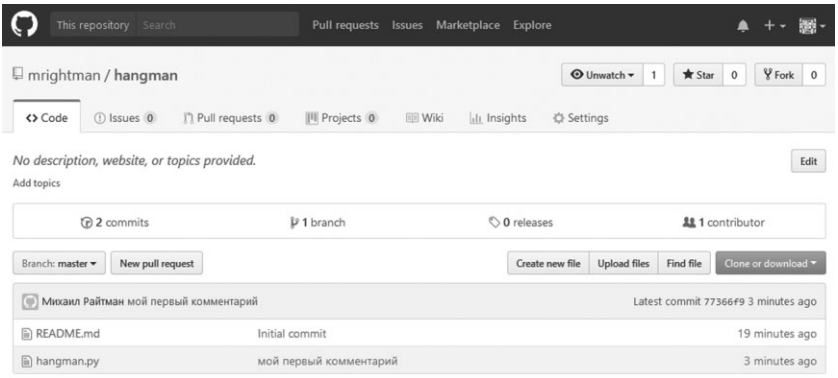
```
$ git commit -m "мой первый комментарий"

>> 1 file changed, 1 insertion(+) create mode 100644 hangman.py
```

Теперь все готово к последнему шагу. Вы можете внести изменения в свой центральный репозиторий с помощью команды `git push origin master`.

bash_ex49.sh

```
$ git push origin master

>> 1 file changed, 1 insertion(+) create mode 100644 hangman.
   py Corys-MacBook-Pro:hangman coryalthoff$ git push ori-
   gin master Counting objects: 3, done. Delta compression
   using up to 4 threads. Compressing objects: 100% (2/2),
   done. Writing objects: 100% (3/3), 306 bytes | 0 bytes/s,
   done. Total 3 (delta 0), reused 0 (delta 0) To https://
   github.com/coryalthoff/hangman.git f5d44da..b0dab51 mas-
   ter -> master
```

После того как вы введете свои имя пользователя и пароль в командной строке, программа `git` поместит измененные файлы на GitHub. Если на этом сайте вы взглянете на свой центральный репозиторий, то увидите файл `hangman.py` и добавленное к состоянию сообщение.



Пример извлечения данных

В этом разделе вы обновите свой локальный репозиторий, получив изменения из центрального репозитория. Вам нужно будет делать это всякий раз, как вы захотите обновить свой локальный репозиторий согласно изменениям, которые другой программист внес в центральный репозиторий.

Отправляйтесь в свой центральный репозиторий, нажмите кнопку **Create new file** (Создать новый файл). Создайте файл с именем **new.py**, а затем нажмите кнопку **Commit new file** (Записать состояние нового файла). Этого файла еще нет в вашем локальном репозитории, так что локальный репозиторий не синхронизирован с центральным. Вы можете обновить локальный репозиторий согласно изменениям в вашем центральном репозитории при помощи команды `git pull origin master`.

bash_ex50.sh

```
$ git pull origin master

>> remote: Counting objects: 3, done. remote: Compressing
   objects: 100% (2/2), done. remote: Total 3 (delta 0), re-
   used 0 (delta 0), pack-reused 0 Unpacking objects: 100%
   (3/3), done. From https://github.com/coryalthoff/hang-
   man b0dab51..8e032f5 master -> origin/master Updating
   b0dab51..8e032f5 Fast-forward new.py | 1 + 1 file changed,
   1 insertion(+) create mode 100644 new.py
```

Git применил изменения в вашем центральном репозитории к локальному репозиторию. Файл *new.py*, который вы создали в центральном репозитории, теперь должен находиться в локальном репозитории. Проверьте это с помощью команды `ls`.

```
$ ls

>> README.md hangman.py new.py
```

Откат версий

Git сохраняет ваш проект каждый раз, как вы сохраняете состояние файла. Вы можете откатиться к любому предыдущему состоянию — «отмотать назад» свой проект. К примеру, можно вернуть проект к состоянию, сделанному на прошлой неделе. Все ваши файлы и папки останутся такими же, какими они были на прошлой неделе. Тогда вы можете перепрыгнуть к более свежему состоянию. У каждого состояния есть **номер** — уникальная последовательность символов, используемая Git для идентификации состояния.

Вы можете просмотреть историю состояний вашего проекта с помощью команды `git log`, выводящей все состояния.

bash_ex51.sh

```
$ git log

>> commit  8e032f54d383e5b7fc640a3686067ca14fa8b43f  Author:
   Cory Althoff <coryedwardalthoff@gmail.com> Date: Thu Dec
   8 16:20:03 2016 -0800
```

```
Create new.py
commit b0dab51849965144d78de21002464dc0f9297fdc Author: Cory
Althoff <coryalthoff@Corys-MacBook-Pro.local> Date: Thu Dec
8 16:12:10 2016 -0800

my first commit
commit   f5d44dab1418191f6c2bbfd4a2b2fcf74ef5a68f   Author:
Cory Althoff <coryedwardalthoff@gmail.com> Date: Thu Dec 8
15:53:25 2016 -0800 Initial commit
```

Вы должны увидеть три состояния. Первое относится к созданию центрального репозитория. Второе — к обновлению центрального репозитория при индексации файла *hangman.py*. Третье — к созданию файла *new.py*. У каждого состояния есть номер. Вы можете изменить состояние вашего проекта, передав номер состояния в качестве параметра команде `git checkout`. Следующим кодом я мог бы откатить свой проект к тому состоянию, в котором он был, когда я впервые его создал: `git checkout f5d44dab1418191f6c2bbfd4a2b2fcf74ef5a68f`.

Команда `git diff`

Команда `git diff` демонстрирует разницу между файлом в локальном и в центральном репозитории. Создайте новый файл *hello_world.py* в локальном репозитории и добавьте в него код `print("Привет, мир!")`.

Теперь проиндексируйте файл.

bash_ex52.sh

```
$ git add hello_world.py
```

Убедитесь, что все произошло должным образом.

bash_ex53.sh

```
$ git status

>> Changes to be committed: (use "git reset HEAD <file>..." to
   unstage) new file: hello_world.py
```

И сохраните состояние.

bash_ex54.sh

```
$ git commit -m "добавление нового файла"

>> 1 file changed, 1 insertion(+) create mode 100644 hello_
   world.py
```

Поместите свои изменения в центральный репозиторий.

bash_ex55.sh

```
$ git push origin master

>> Counting objects: 3, done. Delta compression using up to
   4 threads. Compressing objects: 100% (2/2), done. Writing
   objects: 100% (3/3), 383 bytes | 0 bytes/s, done. Total 3
   (delta 0), reused 0 (delta 0) To https://github.com/co-
   ryalthoff/hangman.git 8e032f5..6f679b1 master -> master
```

Теперь добавьте в файл *hello_world.py* в локальном репозитории вторую строку, `print("Привет!")`. Этот файл отличается от файла в центральном репозиториуме. Введите команду `git diff`, чтобы увидеть разницу.

bash_ex56.sh

```
$ git diff hello_world.py

>> diff --git a/hello_world.py b/hello_world.py index
   b376c99..83f9007 100644 --- a/hello_world.py +++ b/
   hello_world.py -1 +1,2 print("Print, Привет, мир!")
   +print("Привет!")
```

Git выделил код `print("Привет!")` зеленым цветом шрифта, поскольку это новая строка кода. Оператор сложения (+) обозначает, что эта строка — новая. Если бы вы удалили код, удаленная часть была бы выделена красным цветом шрифта, и ей предшествовал бы оператор вычитания (-).

Дальнейшие шаги

В этой главе я описал возможности Git, которыми вы будете пользоваться чаще всего. Как только вы освоите базовые понятия, я советую посвятить время изучению продвинутых возможностей Git, таких как ветвление и слияние, на сайте **www.codeschool.com/learn/git**.

Словарь терминов

Git: популярная система управления версиями.
GitHub: веб-сайт, хранящий код в облаке.
SVN: популярная система управления версиями.
Извлечение данных: обновление локального репозитория согласно изменениям в центральном репозитории.
Индексирование: указание Git, какие файлы (с изменениями) следует поместить в центральный репозиторий.
Кодовая база: папки и файлы, из которых формируется ваше программное обеспечение.
Локальный репозиторий: репозиторий на вашем компьютере.
Номер состояния: уникальная последовательность символов, используемая Git для идентификации состояния.
Помещение данных: обновление центрального репозитория согласно изменениям в локальном репозитории.

Репозиторий: структура данных, созданная системой управления версиями, наподобие Git, которая отслеживает все изменения в вашем программистском проекте.
Система управления версиями: программа, разработанная с целью помогать вам без труда работать над проектами совместно с другими программистами.
Состояние: версия вашего проекта, сохраняемая Git.
Сохранение состояния: написание команды, указывающей Git записывать изменения, которые вы внесли в свой репозиторий.
Структура данных: способ организовывать и хранить информацию. Списки и словари являются примерами структур данных.
Центральный репозиторий: репозиторий, размещенный на веб-сайте вроде GitHub, с которым обмениваются информацией все локальные репозитории для поддержания синхронизации друг с другом.

Практикум

Создайте новый репозиторий на GitHub. Переместите все ваши файлы Python из ранее выполненных заданий в один каталог на компьютере и поместите их в ваш новый репозиторий.

Глава 20. Практикум. Часть III

В наше время осуществилось волшебство мифа и легенды.
С клавиатуры вводится верное заклинание, и экран монитора оживает,
показывая то, чего никогда не было и не могло быть.

Фредерик Брукс

В этой главе вы увидите, насколько мощным является программирование, создав **парсер веб-контента** — программу, которая извлекает данные с веб-сайта. Когда вы создадите парсер контента, вы получите возможность собирать данные из самого объемного информационного хранилища, когда-либо существовавшего. Мощь парсеров контента, а также простота их создания являются одной из причин, по которым я подсел на программирование, надеюсь, на вас это произведет аналогичный эффект.

HTML

Прежде чем приступать к созданию парсера контента, позвольте мне быстро ввести вас в курс дела насчет **HTML** — языка гипертекстовой разметки (англ. hypertext markup language). Наряду с CSS и JavaScript, язык HTML является одной из фундаментальных технологий, с помощью которых программисты строят веб-сайты. HTML — язык, который создает структуру сайта, он состоит из HTML-элементов, используемых браузером для создания макетов веб-страниц. Вы можете построить целый сайт при помощи HTML, и хотя он не будет интерактивным и не будет приятно выглядеть из-за того, что за интерактивность отвечает JavaScript, а за стиль — CSS, это все равно будет настоящий веб-сайт. Ниже показан сайт, который отображает текст `Привет, мир!`.

bash_ex57.sh

```
<!--Это HTML-комментарий.
Сохраните этот файл как index.html-->

<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Мой веб-сайт</title>
</head>
<body>
    Привет, мир!
    <a href="https://www.google.com/>
    щелкните здесь</a>
</body>
</html>
```

Сохраните этот HTML-код в файл. Откройте файл с помощью веб-браузера, щелкнув мышью по файлу (может понадобиться щелкнуть правой кнопкой мыши и изменить программу, которая открывает такие файлы по умолчанию, выбрав браузер, например, Google Chrome). Как только вы откроете файл в браузере, сразу увидите сайт с надписью `Привет, мир!` и ссылкой на Google.

Для отображения этого сайта браузер использует различные **HTML-элементы**, находящиеся в вашем файле HTML. HTML-элемент (для краткости просто элемент) похож на ключевое слово в программировании — он указывает браузеру сделать что-то. У большинства элементов есть начальный и конечный тег, между которыми содержится текст. Например, на вкладке вашего браузера отображается текст, содержащийся между тегами `<title>` и `</title>`. HTML-элементы могут вкладываться внутрь других HTML-элементов; все, что находится между тегами `<head></head>`, — это метаданные о веб-странице, в то время как содержимое между тегами `<body></body>` определяет сам контент страницы. Вместе теги `<a></a>` создают ссылку. HTML-элементы могут содержать данные. К примеру, помещение строки `href="https://www.google.com"` внутрь HTML-элемента `a` дает браузеру понять, на какой сайт нужно добавить ссылку. В языке HTML есть еще много чего, но для создания своего первого парсера контента этих знаний вам будет достаточно.

Парсинг контента с сайта Google Новости

В данном разделе вы построите парсер контента, который собирает все истории с сервиса Google Новости, извлекая все теги <a> из HTML-кода этого сайта. На сайте Google Новости эти теги используются для создания ссылок на различные сайты, так что помимо некоторых дополнительных данных вы соберете все URL-адреса новостей, отображаемых на сайте. Вы воспользуетесь модулем `BeautifulSoup`, чтобы **выполнить парсинг** HTML-кода сайта Google Новости. Парсинг означает принятие формата вроде HTML и придание ему структуры при помощи языка программирования. К примеру, превращение данных в объект. Для начала используйте следующую команду для установки модуля `BeautifulSoup` на Ubuntu и Unix.

bash_ex58.sh

```
$ sudo pip install beautifulsoup4==4.4.1

>> Successfully installed beautifulsoup4-4.4.1
```

И на Windows (запустите командную строку от имени администратора).

bash_ex59.sh

```
C:\Python36\Scripts\pip3.exe install beautifulsoup4==4.4.1

>> Successfully installed beautifulsoup4-4.4.1
```

Для работы с URL-адресами у Python есть встроенный модуль `urllib`. Введите следующий код в новый файл Python:

Python_ex284.py

```
1 import urllib.request
2 from bs4 import BeautifulSoup

3 class Scraper:
4     def __init__(self,
5                  site):
6         self.site = site

7     def scrape(self):
8         pass
```

Метод `__init__` принимает в качестве параметра сайт, с которого нужно собрать данные. Позднее вы передадите в качестве параметра значение "`https://news.google.ru/`". В классе `Scraper` есть метод `scrape`, который вы будете вызывать всякий раз, когда захотите собрать данные с переданного сайта.

Добавьте следующий код в ваш метод `scrape`:

Python_ex285.py

```
def scrape(self):
    r = urllib.request.urlopen(self.site)
    html = r.read()
```

Функция `urlopen()` отправляет запрос на веб-сайт и возвращает объект `Response`, в котором сохранен HTML-код этого сайта вместе с дополнительными данными. Функция `response.read()` возвращает HTML-код из объекта `Response`. Весь HTML-код с сайта содержится в переменной `html`.

Теперь вы готовы выполнить парсинг HTML-кода. Добавьте в функцию `scrape` новую строку кода, которая создаст объект `BeautifulSoup`, и передайте в функцию в качестве параметра переменную `html` и строку `"html.parser"` (поскольку вы выполняете парсинг HTML-контента).

Python_ex286.py

```
def scrape(self):
    r = urllib.request.urlopen(self.site)
    html = r.read()
    parser = "html.parser"
    sp = BeautifulSoup(html, parser)
```

Объект `BeautifulSoup` делает всю трудную работу и выполняет парсинг HTML-кода. Теперь вы можете добавить в функцию `scrape` код, который вызывает метод `find_all` в объекте `BeautifulSoup`. Передайте в качестве параметра значение `"a"` (это сообщит функции, что нужно искать теги `<a></a>`) и метод вернет все URL-адреса, на которые ссылается сайт, в загруженном вами HTML-коде.

Python_ex287.py

```
def scrape(self):
    r = urllib.request.urlopen(self.site)
    html = r.read()
    parser = "html.parser"
    sp = BeautifulSoup(html, parser)
    for tag in sp.find_all("a"):
        url = tag.get("href")
        if url is None:
            continue
        if "html" in url:
            print("\n" + url)
```

Метод `find_all` возвращает итерируемый объект, который содержит найденные методом объекты `tag`. При каждом прохождении цикла `for` переменной `tag` присваивается значение нового объекта `Tag`. У каждого объекта `Tag`

есть много переменных экземпляра, но вам нужно значение лишь переменной `href`, содержащий все URL-адреса. Его можно получить путем вызова метода `get` и передачи в него значения `"href"` в качестве параметра. Наконец, вы проверяете, содержит ли переменная `url` данные; содержит ли она строку `"html"` (вам не нужно выводить внутренние ссылки), если содержит, вы выводите ее. Ниже показан полный код парсера контента.

Python_ex288.py

```
import urllib.request
from bs4 import BeautifulSoup

class Scraper:
    def __init__(self, site):
        self.site = site

    def scrape(self):
        r = urllib.request.urlopen(self.site)
        html = r.read()
        parser = "html.parser"
        sp = BeautifulSoup(html, parser)
        for tag in sp.find_all("a"):
            url = tag.get("href")
            if url is None:
                continue
            if "html" in url:
                print("\n" + url)

news = "https://news.google.ru/"
Scraper(news).scrape()
```

Когда вы запустите программу, вывод будет выглядеть примерно так:

```
>> https://regnum.ru/russian/fd-south/crimea/alushta.html
>> https://www.gazeta.ru/business/2017/11/01/10965968.shtml
>> https://www.segodnya.ua/regions/krym/v-okkupirovannom-kry-
   mu-zayavili-o-diversii-vzorvan-gazoprovod-1068509.html
>> http://www.vesti.ru/doc.html?id=2949606&cid=9
```

Теперь, когда вы умеете собирать ссылки с сайта Google Новости, ваши возможности безграничны. Вы можете написать программу, которая будет анализировать наиболее часто употребляемые в заголовках слова. Можете разработать программу, которая будет оценивать «настроения» заголовков, чтобы узнать, существует ли корреляция с фондовым рынком. Благодаря парсингу контента вам доступна информация со всего мира, и, я надеюсь, что вас это поражает так же сильно, как и меня.

Словарь терминов

HTML: язык, который организовывает структуру сайта.
HTML-элемент: аналогичен ключевому слову в программировании — инструктирует браузер сделать что-либо.
Парсер веб-контента: программа, извлекающая данные с веб-сайта.
Парсинг: парсинг означает принятие формата вроде HTML и придание ему структуры при помощи языка программирования. К примеру, превращение данных в объект.

Практикум

Модифицируйте ваш парсер контента так, чтобы он сохранял найденные ссылки в файл.

Решение: *challenge_1.py*.

ЧАСТЬ IV

Введение в информатику

Глава 21. Структуры данных

Вообще-то я утверждаю, что разница между плохим программистом и хорошим заключается в том, что именно он считает более важным — свой код или свои структуры данных. Плохие программисты беспокоятся о коде. Хорошие программисты беспокоятся о структурах данных и их отношениях.

Линус Торвальдс

Структуры данных

Структура данных — это формат, используемый для хранения и организации информации. Структуры данных — фундаментальное понятие в программировании, они встроены в большинство языков. Вы уже знаете, как пользоваться некоторыми встроенными в Python структурами данных — списками, кортежами и словарями. В этой главе вы научитесь создавать еще две структуры данных — стеки и очереди.

Стеки

Стек — это структура данных. Как и в список, в стек можно добавлять элементы и удалять их из него, но в отличие от списка, вы можете добавлять и удалять только последний элемент. Если у вас есть список `[1, 2, 3]`, вы можете удалить из него любой элемент. Если у вас есть такой же стек, вы можете удалить лишь последний элемент, `3`, и тогда стек будет иметь вид `[1, 2]`. Теперь можно удалить `2`, а после этого — `1`, и стек станет пустым. Удаление элемента из стека обозначается термином **pop**. Если вы поместите `1` обратно в стек, он будет иметь вид `[1]`, а если затем поместите двойку, то `[1, 2]`. Добавление элемента в стек обозначается словом **push**. Такой вид структуры данных, при котором последний поме-

щенный элемент является первым извлекаемым, называется структурой данных «**последним вошел — первым вышел**» (**LIFO**, last-in-first-out).

LIFO можно представить как стопку тарелок. Если вы сложите пять тарелок одну на другую и захотите вытащить ту, что в самом низу, вам сначала придется убрать все верхние. Каждый фрагмент данных в стеке сродни тарелке, чтобы получить к нему доступ, нужно избавиться от данных сверху.

В этом разделе вы построите стек. В Python есть библиотека с обеими структурами данных, описываемыми в этой главе, но чтобы понять, как они работают, нужно создать их самостоятельно. У стека будет пять методов: `is_empty`, `push`, `pop`, `peek` и `size`. Метод `is_empty` возвращает значение `True` если ваш стек пуст, и `False` в противном случае. Метод `push` добавляет элемент на «вершину» стека, `pop` удаляет и возвращает элемент с «вершины» стека, `peek` возвращает этот элемент, но не удаляет его. Метод `size` возвращает целое число, представляющее количество элементов в стеке. Вот как выглядит реализация стека в Python:

Python_ex289.py

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        last = len(self.items)-1
        return self.items[last]

    def size(self):
        return len(self.items)
```

Если вы создадите новый стек, он будет пуст, и метод `is_empty` вернет значение `True`.

Python_ex290.py

```
stack = Stack()
print(stack.is_empty())
```

```
>> True
```

Если вы добавите новый элемент в стек, метод `is_empty` вернет значение `False`.

Python_ex291.py

```
stack = Stack()
stack.push(1)
print(stack.is_empty())
```

```
>> False
```

Вызовите метод `pop`, чтобы удалить элемент из стека, тогда метод `is_empty` снова вернет значение `True`.

Python_ex292.py

```
stack = Stack()
stack.push(1)
item = stack.pop()
print(item)
print(stack.is_empty())
```

```
>> 1
>> True
```

Наконец, вы можете заглянуть в содержимое стека и узнать его размер.

Python_ex293.py

```
stack = Stack()

for i in range(0, 6):
    stack.push(i)

print(stack.peek())
print(stack.size())
```

```
>> 5
>> 6
```

Изменение порядка символов строки при помощи стека

Стек может изменять направление итерируемого элемента на обратное, поскольку все, что вы поместите в стек, отделяется в обратном порядке. В этом разделе вы решите часто задаваемую на собеседованиях задачу — изменение порядка следования символов в строке с помощью стека путем помещения строки в стек и дальнейшего ее извлечения.

Python_ex294.py

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        last = len(self.items)-1
        return self.items[last]

    def size(self):
        return len(self.items)

stack = Stack()
for c in "Привет":
    stack.push(c)

reverse = ""

for i in range(len(stack.items)):
    reverse += stack.pop()

print(reverse)
```

```
>> тевирП
```

Сначала вы прошли через каждый символ строки "Привет" и поместили его в стек. Затем вы перебрали стек, взяли каждый элемент из стека и поместили в переменную reverse. Как только итерирование завершилось, буквы в исходном слове расположились в обратном порядке, и программа вывела строку тевирП.

Очереди

Очередь — еще одна структура данных. Она тоже похожа на список; вы можете добавлять в нее элементы и удалять их. Очередь также напоминает стек, поскольку добавлять и удалять элементы можно лишь в определенном порядке. В отличие от стека, где первый помещенный элемент является последним извлекаемым, очередь представляет собой структуру данных вида «**первым вошел —**

первым вышел» (**FIFO**, first-in-first-out): первый помещенный элемент является первым извлекаемым.

Представьте FIFO как очередь людей, желающих купить билеты в кино. Первый человек в очереди — это первый человек, который купит билеты, второй человек в очереди — второй, кто купит билеты, и так далее.

В этом разделе вы построите очередь, используя четыре метода: `enqueue`, `dequeue`, `is_empty` и `size`. `enqueue` добавляет новый элемент в очередь; `dequeue` удаляет элемент из очереди; `is_empty` проверяет, пуста ли очередь, и возвращает соответственно `True` либо `False`; `size` возвращает количество элементов в очереди.

Python_ex295.py

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Если вы создадите новую пустую очередь, метод `is_empty` вернет значение `True`.

Python_ex296.py

```
a_queue = Queue()
print(a_queue.is_empty())
```

```
>> True
```

Добавьте в очередь элементы и узнайте ее размер.

Python_ex297.py

```
a_queue = Queue()

for i in range(5):
    a_queue.enqueue(i)

print(a_queue.size())
```

```
>> 5
```

Удалите каждый элемент из очереди.

Python_ex298.py

```
a_queue = Queue()

for i in range(5):
    a_queue.enqueue(i)

for i in range(5):
    print(a_queue.dequeue())

print()

print(a_queue.size())
```

```
>> 0
>> 1
>> 2
>> 3
>> 4
>>
>> 0
```

Очередь за билетами

При помощи очереди можно смоделировать ситуацию, когда люди выстроились в ряд и ждут, чтобы купить билеты на фильм.

Python_ex299.py

```
import time
import random

class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

def enqueue(self, item):
        self.items.insert(0, item)

def dequeue(self):
        return self.items.pop()
```

```
def size(self):
        return len(self.items)

    def simulate_line(self, till_show, max_time):
        pq = Queue()
        tix_sold = []

        for i in range(10):
            pq.enqueue("кинофанат " + str(i))

    t_end = time.time() + till_show
    now = time.time()
    while now < t_end and not pq.is_empty():
        now = time.time()
        r = random.randint(0, max_time)
        time.sleep(r)
        person = pq.dequeue()
        print(person)
        tix_sold.append(person)

    return tix_sold

queue = Queue()
sold = queue.simulate_line(5, 1)
print(sold)
```

```
>> кинофанат 0
...
>> ['кинофанат 0', 'кинофанат 1', 'кинофанат 2', ...
```

Вначале вы создали функцию `simulate_line`, которая моделирует продажу билетов людям в очереди. Функция принимает два параметра: `till_show` и `max_time`. Первый параметр является целым числом, представляющим число секунд, оставшихся до того момента, когда начнется фильм и уже не останется времени на покупку билетов. Второй параметр также является целым числом, представляющим максимальное количество времени в секундах, которое уходит на покупку билета одним человеком.

Внутри функции вы создаете новую пустую очередь и пустой список. Список будет отслеживать людей, купивших билет. Затем вы заполняете очередь одной сотней строк, начиная с `"кинофанат 0"` и заканчивая `"кинофанат 99"`. Каждая строка в очереди представляет человека в очереди, желающего купить билет.

Во встроенном модуле `time` есть функция `time`. Она возвращает число с плавающей точкой, представляющее число секунд, прошедших с начала **эпохи**, момента во времени (1 января 1970 года), который используется в качестве ссылки.

Если я вызову функцию `time` прямо сейчас, она вернет `1481849664.256039` — число секунд, прошедших с начала эпохи. Если спустя одну секунду я снова вызову эту функцию, число с плавающей точкой, возвращаемое ей, будет увеличено на `1`.

Переменная `t_end` находит результат сложения значения функции `time` и числа секунд, переданного в переменной `till_show`. Этот результат обозначает момент в будущем.

Цикл `while` будет продолжать итерирование до тех пор, пока либо функция `time` не вернет значение, большее, чем значение `t_end`, либо пока очередь не станет пустой.

После этого вы используете функцию `sleep` из встроенного модуля `time`, чтобы случайное количество секунд между `0` и `max_time` интерпретатор Python ничего не делал. Вы указываете Python остановить выполнение кода, чтобы сымитировать время, требующееся на продажу билета, а случайное количество времени используется для имитации того, что на каждую продажу билета уходит разное время.

После паузы, вызванной функцией `sleep`, вы извлекаете из очереди строку, представляющую человека, и помещаете ее в список `tix_sold`, который представляет человека, уже купившего билет.

Результатом этого кода будет функция, которая умеет продавать билеты людям в очереди, продавая их больше или меньше в зависимости от переданных параметров и случайного значения.

Словарь терминов

FIFO: сокращение от англ. first-in-first-out — «первым вошел — первым вышел».
LIFO: сокращение от англ. last-in-first-out — «последним вошел — первым вышел».
Pop: удаление элемента из стека.
Push: добавление элемента в стек.
Очередь: структура данных вида «первым вошел — первым вышел».
Стек: структура данных вида «последним вошел — первым вышел».
Структура данных «первым вошел — первым вышел»: структура данных, в которой первый помещенный элемент является первым извлекаемым.
Структура данных «последним вошел — первым вышел»: вид структуры данных, при котором последний помещенный элемент является первым извлекаемым.
Структура данных: формат, используемый для хранения и организации информации.
Эпоха: момент во времени, используемый в роли ссылки.

Практикум

1. Измените порядок символов строки `"позавчера"` при помощи стека.
2. Используйте стек, чтобы создать новый список с элементами списка `[1, 2, 3, 4, 5]` в обратном порядке.

Решения: *challenge1.py и challenge2.py*.

Глава 22. Алгоритмы

Алгоритм подобен кулинарному рецепту.

Васим Латиф

Эта глава представляет собой упрощенное введение в алгоритмы. Алгоритм — это серия шагов, которым следуют при решении проблемы. Проблема может заключаться в поиске в списке или выводе слов песни «99 бутылок пива на стене».

FizzBuzz

Наконец-то пришло время научиться решать FizzBuzz, известный тест, используемый на собеседованиях для отсеивания кандидатов.

Напишите программу, которая выводит числа от 1 до 100. Но вместо чисел, кратных трем, выводите «Fizz», вместо кратных пяти выводите «Buzz», вместо кратных и трем, и пяти — «FizzBuzz».

Для решения этой задачи нужно придумать способ проверить, кратно ли число трем, пяти, и трем, и пяти, либо ни трем, ни пяти. Если число кратно трем, это значит, что оно делится на три без остатка. То же самое касается пяти. Оператор деления по модулю (`%`) возвращает остаток. Задачу можно решить, перебирая числа и проверяя, кратно ли число и трем и пяти, только трем, только пяти, или не кратно ни тому, ни другому.

Python_ex300.py

```
def fizz_buzz():
    for i in range(1, 101):
        if i % 3 == 0 and i % 5 == 0:
            print("FizzBuzz")
        elif i % 3 == 0:
            print("Fizz")
        elif i % 5 == 0:
            print("Buzz")
        else:
            print(i)

fizz_buzz()
```

```
>> 1
>> 2
>> Fizz
...
```

Сначала вы перебираете числа от 1 до 100. Затем вы проверяете, делится ли число и на `3`, и на `5`. Это важно сделать в первую очередь, так как если число делится и на 3, и на 5, нужно вывести `FizzBuzz` и перейти к следующей итерации

цикла. Если бы вы сначала проверяли, делится ли число только на 3 или только на 5, и обнаружили бы такое число, вы бы не могли вывести Fizz или Buzz и перейти к следующей итерации цикла, поскольку это число все еще могло бы делиться и на 3, и на 5. А в этом случае вывод Fizz или Buzz был бы неверен, ведь нужно было бы выводить FizzBuzz.

Как только вы проверили, делится ли число и на 3, и на 5, порядок этих двух тестов больше не важен, ведь вам известно, что число не делится на оба этих числа. Если число делится либо на 3, либо на 5, вы останавливаете алгоритм и выводите Fizz или Buzz. Если число проходит мимо первых трех условий, значит, оно не делится ни на 3, ни на 5 (ни на оба этих числа одновременно), и его можно вывести.

Последовательный поиск

Поисковый алгоритм находит информацию в структуре данных, например в списке. **Последовательный поиск** — это простой поисковый алгоритм, который проверяет каждый элемент в структуре данных на предмет его соответствия тому, что алгоритм ищет.

Если вы когда-то играли в карты и искали в колоде определенную карту, то, вероятно, осуществляли последовательный поиск. Вы перебирали каждую карту в колоде, одну за другой, и если карта была не той, которую вы искали, вы переходили к следующей карте. Когда вы наконец находили желаемую карту, то останавливались. Если вы перебрали всю колоду и не нашли карту, вы также останавливались, поскольку понимали, что карты там нет. Ниже приведен пример последовательного поиска в Python.

Python_ex301.py

```
def ss(number_list, n):
    found = False
    for i in number_list:
        if i == n:
            found = True
            break
    return found

numbers = range(0, 100)
s1 = ss(numbers, 2)
print(s1)
s2 = ss(numbers, 202)
print(s2)
```

```
>> True
>> False
```

Вначале вы присвоили переменной `found` значение `False`. Эта переменная отслеживает, нашел ли алгоритм искомое число. Затем вы перебираете каждое число в списке и проверяете, является ли это число искомым. Если является, вы присваиваете переменной `found` значение `True`, выходите из цикла и возвращаете значение переменной `found` (которой теперь присвоено значение `True`).

Если вы не нашли желаемое искомое число, то переходите к следующему числу в списке. Если вы перебрали весь список, то возвращаете переменную `found`, которая примет значение `False`, если число отсутствует в списке.

Палиндром

Палиндром — это слово, которое одинаково записывается в обоих направлениях. Вы можете написать алгоритм, который будет проверять, является ли слово палиндромом, меняя порядок букв в нем на обратный и сравнивая измененное слово с исходным. Если измененное и исходное слово записываются одинаково, значит, исходное слово было палиндромом.

Python_ex302.py

```
1 def palindrome(word):
2     word = word.lower()
3     return word[::-1] == word

4 print(palindrome("Мама"))
5 print(palindrome("Мам"))
```

```
>> False
>> True
```

Метод `lower` переводит прописные символы в проверяемом слове в нижний регистр (строчное написание). Для Python `М` и `м` — это два разных символа, а вам нужно, чтобы он рассматривал их как один и тот же символ.

Код `word[::-1]` меняет порядок букв в слове на обратный. `[::-1]` — это синтаксис, используемый Python для возвращения среза целого итерируемого объекта в обратном порядке. Вы меняете порядок букв в слове на обратный, чтобы сравнить измененное слово с исходным, если они одинаковые, функция возвращает `True`, поскольку исходное слово является палиндромом. В противном случае функция возвращает `False`.

Анаграмма

Анаграмма — это слово, созданное путем перестановки букв другого слова. Слово «тапок» является анаграммой слова «капот», поскольку каждое из этих слов может быть получено путем перестановки букв другого слова. Определить, являются ли два слова анаграммами, можно отсортировав буквы каждого из них в алфавитном порядке и проверив, одинаковые ли они.

Python_ex303.py

```
def anagram(w1, w2):
    w1 = w1.lower()
    w2 = w2.lower()
    return sorted(w1) == sorted(w2)

print(anagram("тапок", "капот"))
print(anagram("лист", "древо"))
```

```
>> True
>> False
```

Сначала вы вызываете метод `lower` в обоих словах, чтобы исключить влияние регистра на результат. Затем оба слова передаются в метод Python `sorted`. Этот метод возвращает слова с буквами, отсортированными в алфавитном порядке. Наконец, вы сравниваете результаты. Если отсортированные слова совпадают, ваш алгоритм возвращает `True`, если нет — `False`.

Подсчет вхождений символов

В этом разделе вы напишете алгоритм, возвращающий число — количество раз, которое каждый символ встречается в строке. Алгоритм будет перебирать строку символ за символом и в словаре отслеживать, сколько раз встречается каждый символ.

Python_ex304.py

```
def count_characters(string):
    count_dict = {}
    for c in string:
        if c in count_dict:
            count_dict[c] += 1
        else:
            count_dict[c] = 1
    print(count_dict)

count_characters("Династия")
```

```
>> {'Д': 1, 'и': 2, 'н': 1, 'а': 1, 'с': 1, 'т': 1, 'я': 1}
```

В этом алгоритме вы перебираете каждый символ в строке `"Династия"`, переданной в качестве параметра `string`. Если символ уже содержится в словаре `count_dict`, значение символа увеличивается на `1`.

В противном случае вы добавляете символ в словарь и устанавливаете его значение на `1`. К концу цикла `for` словарь `count_dict` содержит пары ключ-значение для каждого символа строки. Значение каждого ключа — это число вхождений соответствующего символа в строке.

Рекурсия

Рекурсия — это способ решения проблем, который предполагает разбивание проблемы на все меньшие и меньшие фрагменты, пока проблема не решается элементарно. До сих пор вы решали проблемы, используя **итерационные алгоритмы**. Итерационные алгоритмы решают проблемы путем повторения шагов снова и снова, обычно при помощи цикла. **Рекурсивные (рекуррентные) алгоритмы** полагаются только на функции, которые вызывают сами себя. Любую проблему, которую можно решить итерационно, можно решить рекурсивно, но иногда рекурсивный алгоритм представляет собой более изящное решение.

Рекурсивный алгоритм создается внутри функции. Функция должна содержать **базовый случай** — условие, завершающее рекурсивный алгоритм, не позволяющее ему продолжаться вечно. Внутри функции функция вызывает себя сама. Каждый раз, как функция вызывает себя сама, она продвигается ближе к базовому случаю. В конце концов, выполняется условие базового случая, проблема решается, и функция перестает вызывать себя. Алгоритм, следующий таким правилам, отвечает трем законам рекурсии.

1. Рекурсивный алгоритм должен содержать базовый случай.
2. Рекурсивный алгоритм должен изменять свое состояние и двигаться по направлению к базовому случаю.
3. Рекурсивный алгоритм должен вызывать сам себя[15].

Ниже показан рекурсивный алгоритм, выводящий слова известной народной песни «99 бутылок пива на стене».

Python_ex305.py

```
def bottles_of_beer(bob):
    """ Печатает текст песенки про 99 бутылок пива.
        :param bob: Должно быть целым числом.
    """
    if bob < 1:
        print("""Нет бутылок пива на стене. Нет бутылок
              пива.""")
        return
    tmp = bob
    bob -= 1
    print("""{} бутылок пива на стене. {} бутылок пива.
         Возьми одну, пусти по кругу, {} бутылок пива на
         стене.
          """.format(tmp,
                     tmp,
                     bob))
```

[15] interactivepython.org/runestone/static/pythonds/Recursion/TheThreeLawsofrecursion.htm

```
    bottles_of_beer(bob)

bottles_of_beer(99)
```

```
>> 99 бутылок пива на стене. 99 бутылок пива.
>> Возьми одну, пусти по кругу, 98 бутылок пива на стене.
```

В этом примере первый закон рекурсии был выполнен благодаря следующему базовому случаю:

Python_ex306.py

```
if bob < 1:
print("""Нет бутылок пива на стене. Нет бутылок пива.""")
return
```

Когда значение переменной `bob` становится меньше `1`, функция перестает вызывать сама себя.

Строка кода `bob -= 1` отвечает второму закону рекурсии, поскольку уменьшение переменной `bob` приводит к движению по направлению к базовому случаю. В данном примере вы передали в вашу функцию число `99` в качестве параметра. Базовый случай выполняется, когда переменная `bob` меньше `1`, а каждый раз, когда функция вызывает себя, она двигается к базовому случаю.

Последний закон рекурсии выполняется благодаря этому.

Python_ex307.py

```
bottles_of_beer(bob)
```

Эта строка кода проверяет, что пока базовый случай не выполнен, ваша функция будет вызывать себя. Каждый раз, как функция вызывает себя, она передает себе параметр, который был уменьшен на `1`, и таким образом продвигается к базовому случаю. В первый раз, когда функция вызывает себя в этой строке кода, она передает себе в качестве параметра `98`, затем `97`, затем `96`, пока, в конечном итоге, не передаст себе параметр меньше `1`, что выполнит базовый случай и выведет строку `Нет бутылок пива на стене. Нет бутылок пива`. Тогда функция сталкивается с ключевым словом `return`, что останавливает алгоритм.

Как известно, рекурсия — одна из самых сложных концепций для начинающих программистов. Если поначалу она будет казаться вам запутанной, не волнуйтесь — просто продолжайте практиковаться. И помните: чтобы понять рекурсию, сначала нужно понять рекурсию.

Словарь терминов

Алгоритм: серия шагов, которым следуют для решения проблемы.
Анаграмма: слово, созданное путем перестановки букв другого слова.
Базовый случай: условие, завершающее рекурсивный алгоритм.
Итерационный алгоритм: решает проблемы путем повторения шагов снова и снова, обычно используя цикл.

Палиндром: слово, которое одинаково записывается в обоих направлениях.
Поисковый алгоритм: алгоритм, который находит информацию в структуре данных, например в списке.
Последовательный поиск: простой поисковый алгоритм для поиска информации в структуре данных, который проверяет каждый элемент в ней на предмет его соответствия тому, что алгоритм ищет.
Рекурсивный (рекуррентный) алгоритм: рекурсивные алгоритмы используют функции, которые вызывают сами себя.
Рекурсия: способ решения проблем, который предполагает разбивание проблемы на все меньшие и меньшие фрагменты, пока проблема не решается элементарно.

Практикум

Зарегистрируйте учетную запись на сайте **leetcode.com** и попытайтесь решить на нем три алгоритмические задачи легкого уровня.

ЧАСТЬ V

Получение работы

Глава 23. Лучшие практические советы по программированию

Всегда пишите код так, будто сопровождать его будет
склонный к насилию психопат, который знает, где вы живете.

Джон Вудс

Производственный код — это код используемого людьми продукта. Отправить программное обеспечение в **производство** — значит вывести его в мир. В этой главе я рассказываю о нескольких общих принципах программирования, которые помогут вам в написании окончательного, готового к производству кода. Многие из этих принципов были описаны в «*Программисте-прагматике*», книге Энди Ханта и Дейва Томаса, которая значительно улучшила качество моего кода.

Написание кода — крайнее средство

Ваша работа как инженера-программиста заключается в том, чтобы писать как можно меньше кода. Когда у вас возникает проблема, вашей первой мыслью должно быть не «Как я могу решить ее?», а «Решил ли уже кто-то эту проблему, и если да, то могу ли я использовать это решение?». Если вы пытаетесь решить распространенную проблему, есть высокий шанс, что кто-то уже в ней разобрался. Начните с поиска решения онлайн, и только после того, как выяснится, что вашу проблему еще никто не решил, сами приступайте к решению.

НПС

НПС (Не повторяй себя) — это принцип программирования, согласно которому в программе не следует повторно использовать один и тот же или сильно похо-

жий код. Вместо этого нужно поместить код в одну функцию, обрабатывающую множество ситуаций.

Ортогональность

Ортогональность — еще один важный программистский принцип, ставший известным благодаря книге «Программист-прагматик». Хант и Томас дают такое объяснение: «Этот термин был введен в информатике для обозначения некой разновидности независимости или несвязанности. Два или более объекта ортогональны, если изменения, вносимые в один из них, не влияют на любой другой. В грамотно спроектированной системе код базы данных будет ортогональным к интерфейсу пользователя: вы можете менять интерфейс пользователя без воздействия на базу данных и менять местами базы данных, не меняя интерфейса». Чтобы применить этот принцип на практике, хорошенько запомните, что «А не должно влиять на Б». Если у вас есть два модуля, `module a` и `module b`, то `module a` не должен вносить изменения в `module b` и наоборот. Если вы спроектируете систему, где А влияет на Б, Б влияет на В, В влияет на Г, — все очень быстро выйдет из-под контроля и система станет неуправляемой.

У каждого фрагмента данных должно быть одно представление

Один фрагмент данных следует хранить в одном месте. Скажем, вы разрабатываете программное обеспечение, обрабатывающее телефонные номера. Если у вас есть две функции, использующие список кодов регионов, проследите, чтобы все коды были помещены в один список — не должно быть двух дублирующих друг друга списков для каждой функции. Используйте глобальную переменную, которая будет хранить коды регионов. Или еще лучше, храните данные в файле или базе данных.

Проблема дублирования данных в том, что на определенном этапе вам понадобится изменить эти данные. Тогда вам придется вспоминать все места с дубликатами, чтобы изменить также и их. Если вы измените список кодов регионов в одной функции и забудете о другой, ваша программа не будет работать должным образом. Этого можно избежать, если у каждого фрагмента данных будет одно представление.

У функции должна быть одна задача

У каждой написанной вами функции должна быть одна, и только одна, задача. Если вам покажется, что функции становятся слишком длинными, спросите себя — возможно, они выполняют больше одной задачи? Ограничение функций на выполнение одной задачи дает несколько преимуществ. Код будет легче читаться, ведь имя вашей функции укажет именно на то, чем она занимается. Если код не работает, то будет легче осуществить отладку — поскольку когда каждая функция отвечает за конкретную задачу, их можно быстро изолировать и определить, какая из них не работает. Как говорили многие известные программисты; «Столько сложностей в программном обеспечении происходит от попыток заставить одну вещь делать две вещи».

Если на это уходит много времени, вероятно, вы совершаете ошибку

Если вы не работаете над чем-то очевидно сложным и не работаете с огромным количеством данных, а ваша программа очень долго загружается, вероятно, вы делаете что-то не так.

Делайте все самым лучшим способом

Если в процессе программирования вы подумали: «Я знаю, что существует лучший способ сделать это, но я уже не хочу останавливаться и выяснять, как это сделать», не продолжайте кодинг. Остановитесь. Вы должны сделать это лучшим способом.

Соблюдайте соглашения

Если вы потратите время на изучение соглашений нового языка программирования, то сможете быстрее читать код на этом языке. PEP 8 представляет собой сборник методических рекомендаций по написанию кода на Python — ознакомьтесь с ним. Там есть правила по переносу кода на новые строки. На русском языке документ PEP 8 доступен по адресу **pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html**.

Используйте мощную IDE

До сих пор для написания кода вы использовали IDLE — интерактивную среду разработки (IDE), которая поставляется с Python. Но IDLE — лишь одна из многих доступных IDE, и я не советую использовать ее долгое время, поскольку она не слишком мощная. Например, при открытии проекта Python в более удобной среде разработки для каждого файла открывается своя вкладка. В IDLE для каждого файла нужно открывать отдельное окно, что усложняет процесс переключения между файлами и может быть утомительно.

Я пользуюсь оболочкой PyCharm, разработанной компанией JetBrains. Доступны бесплатная и профессиональная (платная) версии. Ниже приведен список возможностей PyCharm, которые существенно сэкономят мое время:

1. Если вы захотите увидеть определение переменной, функции или объекта, у PyCharm есть ссылки, переносящие вас к определившему их коду (даже если он находится в другом файле). Также есть ссылки для возвращения на исходную страницу.
2. В PyCharm присутствует возможность сохранения истории, и это значительно повысило мою производительность. PyCharm автоматически сохраняет последнюю версию проекта после каждого изменения. Вы можете использовать PyCharm как локальную систему управления версиями, не помещая данные в облачный репозиторий. Вам не нужно ничего делать, все происходит автоматически. Раньше, когда я не знал об этой возможности, я решал проблему, менял решение, затем у меня появлялась необходимость откатиться к прежнему решению, но если я не помещал исходное решение на GitHub, оно пропадало, и мне приходилось начинать все с начала. Используя эту опцию,

вы можете откатить время на 10 минут назад и заново загрузить ваш проект в том самом состоянии, в котором он тогда находился. Если вы снова передумаете, можете перемещаться туда-обратно между различными решениями столько раз, сколько захотите.

3. Во время работы вы, вероятно, часто копируете и вставляете код. В PyCharm вместо копирования и вставки вы можете перетаскивать код вверх и вниз по текущей странице.
4. PyCharm поддерживает системы управления версиями, такие как Git и SVN. Вместо использования командной строки вы можете использовать Git из PyCharm. Чем меньше переключений между вашей IDE и командной строкой, тем вы более производительны.
5. В PyCharm есть встроенная командная строка и оболочка Python.
6. В PyCharm есть встроенный **отладчик**. Отладчик — это программа, позволяющая остановить выполнение кода и просмотреть программу строка за строкой, отмечая значения переменных в разных частях программы.

Если вы хотите научиться пользоваться PyCharm, на сайте JetBrains есть руководство, доступное по адресу **www.jetbrains.com/help/pycharm/2016.1/quick-start-guide.html**.

Логирование

Логирование — это практика записи данных во время работы программного обеспечения. Логирование можно использовать для отладки программы и получения дополнительной информации о том, что происходило во время работы вашей программы. Python поставляется с модулем `logging`, позволяющим записывать процессы работы в консоль или файл.

Когда что-то в вашей программе работает неправильно, нельзя оставлять это без внимания — нужно запротоколировать информацию о случившемся, чтобы изучить ее позже. Логирование также полезно при сборе и анализе данных. Например, можно настроить веб-сервер, чтобы он производил журналирование данных, включая дату и время, каждый раз, как он получает запрос. Все ваши журналы нужно сохранить в базе данных и создать программу для анализа этих данных, построив граф, отображающий количество посещений вашего сайта в день.

Блогер Хенрик Варне пишет: «Одно из отличий хорошего программиста от плохого заключается в том, что хороший программист использует логирование, а также другие инструменты, позволяющие с легкостью отладить программу, когда все рушится». О том, как использовать модуль `logging`, можно прочесть на сайте **docs.python.org/3/howto/logging.html**.

Тестирование

Под **тестированием** программы подразумевается процесс проверки того, что она «соответствует требованиям, сопровождавшим ее проектирование и разработку, корректно отвечает на все виды ввода, выполняет свои функции за приемлемое количество времени, пригодна к использованию, может быть установлена и запущена в предполагаемых окружениях, а также достигает результата, требуе-

мого от нее заинтересованными сторонами». Для тестирования своих программ программисты пишут другие программы.

В производстве тестирование является обязательным. Ни одна программа, отправляемая в производство, не может считаться полноценной до тех пор, пока для нее не будут написаны тесты. Впрочем, если вы написали небольшую программу, которую больше не собираетесь использовать, ее тестирование может быть тратой времени. Если же вы пишете программы, которые будут использоваться другими людьми, вы обязаны написать тесты. Как говорили некоторые известные программисты, «Непротестированный код — нерабочий код». Узнайте, как использовать модуль Python `unittest` на странице **docs.python.org/3/library/unittest.html**.

Анализ кода

При **анализе кода** кто-либо читает ваш код и составляет на него отзыв. Следует делать столько обзоров кода, сколько возможно — особенно если вы программист-самоучка. Даже если вы последуете всем советам, изложенным в этой главе, вы все равно будете делать что-то неправильно. Нужно, чтобы кто-то опытный прочел ваш код и указал на ошибки, которые вы затем устраните.

Code Review — сайт, где можно провести анализ кода сообществом программистов. Любой желающий может зайти на Code Review и опубликовать свой код; другие участники сообщества Stack Exchange просматривают этот код, составляют отзыв, указывая, что было сделано хорошо, и предлагают полезные советы по его улучшению. Посетите Code Review по адресу **codereview.stackexchange.com**.

Безопасность

Программисты-самоучки часто игнорируют тему безопасности. Скорее всего, на собеседованиях вас не будут спрашивать об этом; безопасность не важна для программ, которые вы пишете во время обучения. Однако, как только вы получите свою первую должность, вы будете непосредственно ответственны за безопасность вашего кода. В этом разделе я дам некоторые советы по поддержанию безопасности кода.

Ранее вы научились использовать команду `sudo` для работы от имени суперпользователя. Никогда не запускайте программу из командной строки, используя `sudo`, если в этом нет необходимости, поскольку если программа будет скомпрометирована, хакер получит доступ к правам root. Также, если вы управляете веб-сервером, следует отключить возможность входа в систему от имени суперпользователя. Все хакеры знают о существовании учетной записи суперпользователя, так что при атаке на систему это будет легкая цель.

Никогда не забывайте, что ввод пользователя может быть вредоносным. Несколько видов вредоносных атак построены на эксплойтах, использующих ввод, поэтому нужно исходить из того, что ввод всех пользователей вредоносен, и программировать в соответствии с этим.

Еще одной стратегией поддержки вашего программного обеспечения в безопасности может считаться минимизирование **поверхности атаки** — различных областей программы, откуда хакеры могут извлечь данные или атаковать

вашу систему. Уменьшая поверхность атаки, вы снижаете вероятность появления в вашей программе уязвимостей. Далее перечислю несколько стратегий по уменьшению поверхности атаки: не храните конфиденциальные данные без необходимости; предоставляйте пользователям самый низкий уровень доступа из возможных; используйте как можно меньше сторонних библиотек (меньше кода, меньше эксплойтов); избавьтесь от возможностей, которые больше не используются.

Отключение функции авторизации от имени суперпользователя, настороженное отношение к пользовательскому вводу, минимизирование поверхности атаки — важные шаги на пути к обеспечению безопасности программ. Но это только начало. Всегда пытайтесь думать как злоумышленник. Как бы он смог использовать ваш код в своих интересах? Такой способ мышления поможет найти уязвимости, которые иначе вы могли проглядеть. Аспектов, касающихся безопасности, существует куда больше, чем я смогу раскрыть в этой книге, так что непрерывно размышляйте о безопасности и изучайте ее. Брюс Шнайер сказал лучше всех: «Безопасность — это состояние души».

Словарь терминов

Анализ кода: когда другой программист читает ваш код и составляет на него отзыв.
Логирование: практика записи данных во время работы программного обеспечения.
НПС: принцип программирования «Не повторяй себя».
Ортогональность: «Этот термин был введен в информатике для обозначения некой разновидности независимости или несвязанности. Два или более объекта ортогональны, если изменения, вносимые в один из них, не влияют на любой другой. В грамотно спроектированной системе код базы данных будет ортогональным к интерфейсу пользователя: вы можете менять интерфейс пользователя без воздействия на базу данных и менять местами базы данных, не меняя интерфейса».
Отладчик: программа, позволяющая остановить выполнение кода и просмотреть вашу программу строка за строкой, отмечая значения переменных в разных частях программы.
Поверхность атаки: различные области программы, откуда хакеры могут извлечь данные или атаковать вашу систему.
Производственный код: код используемого людьми продукта.
Производство: отправить программное обеспечение в производство — значит вывести его в мир.
Тестирование: процесс проверки того, что программа «соответствует требованиям, сопровождавшим ее проектирование и разработку, корректно отвечает на все виды ввода, выполняет свои функции за приемлемое количество времени, пригодна к использованию, может быть установлена и запущена в предполагаемых окружениях, а также достигает результата, требуемого от нее заинтересованными сторонами».

Глава 24. Ваша первая работа программистом

Остерегайтесь словосочетания «реальный мир».
Обращение к нему говорящего — это всегда предложение не спорить
с подразумеваемыми им допущениями.

Эдсгер В. Дейкстра

Заключительная часть этой книги посвящена тому, чтобы помочь вам в вашей карьере. Чтобы получить первую работу на должности программиста, нужно приложить дополнительные усилия, но если вы последуете моему совету, у вас не возникнет с этим сложностей. К счастью, после того, как вы получите свою первую работу и наберетесь немного опыта, при поиске следующей работы менеджеры по персоналу уже сами будут обращаться к вам.

Выберите путь

Когда вы подаете заявку на должность программиста, от вас ожидают владения определенным набором технологий, который изменяется в зависимости от конкретной области. Хотя во время обучения программированию вполне можно быть универсалом (программистом, который занимается всем подряд), да и получить работу, будучи универсалом, тоже можно, вам все же стоит сконцентрироваться на той области программирования, которая вам нравится больше всего и в которой вы хотите стать экспертом. Это упростит получение работы мечты.

Мобильная и веб-разработка — одни из самых популярных направлений для программистов. В них есть два вида специальностей: front-end и back-end. Front-end приложения это та часть, которую вы видите, — например, графический интерфейс веб-приложения. Back-end — это то, чего вы не видите; часть, обеспечивающая графический интерфейс данными. Названия программистских вакансий будут выглядеть примерно так: «Программист Python (back-end)». Это значит, нужен кто-то занимающийся back-end программированием веб-сайта и знакомый с Python. Описание вакансии будет содержать список технологий, с которыми должен быть знаком идеальный кандидат, а также некоторые дополнительные навыки.

В некоторых компаниях есть отдельные команды, одна из которых занимается front-end разработкой, а другая — back-end. Где-то нанимают только разработчиков полного стека (full stack developer) — программистов, которые одинаково хорошо работают и с front-end, и с back-end; впрочем, это касается только компаний, занимающихся созданием сайтов или приложений.

Существует множество других областей программирования, в которых можно работать, таких как обеспечение безопасности, инженерия платформ и наука о данных. Описания вакансий на сайтах поиска работы могут пригодиться при изучении требований в различных областях программирования. Раздел Job Board на сайте Python (**www.python.org/jobs**) — поможет вам в этом. Ознакомьтесь с требованиями и технологиями в новых вакансиях, чтобы понять,

на что вам стоит обратить внимание для повышения собственной конкурентоспособности.

Получите начальный опыт

Прежде чем вас возьмут на работу программистом, вы должны будете получить соответствующий опыт. Но как это сделать, если без опыта никуда нельзя устроиться? Эта проблема решается несколькими способами. Вы можете начать деятельность в области открытого программного обеспечения, создав свой проект или занявшись любым из тысяч open source проектов на GitHub.

Другой вариант — стать программистом-фрилансером. Создайте профиль на сайте наподобие Upwork и начните подавать заявки на мелкие работы. Советую найти знакомого, которому нужно написать программу — пусть он зарегистрируется на Upwork, официально наймет вас, а после оставит отличный отзыв. Пока у вас не будет хотя бы одного хорошего отзыва на сайтах типа Upwork, получить заказ будет проблематично. Как только люди увидят, что вы справились с заданием, получение работы значительно упростится, ведь у вас уже будет кредит доверия.

Запишитесь на собеседование

Обретя опыт программирования при работе фрилансером либо в open source, можете записываться на собеседование. Самый эффективный способ — использовать сайт LinkedIn. Если у вас нет учетной записи в LinkedIn, заведите ее и начните работать с потенциальными работодателями по сети. В верхней части своего профиля кратко опишите себя, выделив программистские навыки. Например, многие пишут здесь что-то вроде «Языки программирования: Python, JavaScript», чтобы специалистам по набору персонала было легче найти их учетные записи по ключевым словам. Обязательно укажите свой опыт в open source или фрилансе в качестве последнего места работы.

Когда вы заполните профиль, начинайте связываться с менеджерами по техническому персоналу — их в LinkedIn очень много. Менеджеры по персоналу постоянно ищут молодые таланты, так что они будут жаждать пообщаться с вами. Как только они примут ваш запрос, поинтересуйтесь, нет ли у них открытых вакансий.

Собеседование

Если менеджер по персоналу сочтет вас подходящим кандидатом на открытую должность, он отошлет вам сообщение в LinkedIn, пригласив на собеседование по телефону. Это собеседование будет проходить с менеджером по персоналу, так что оно вряд ли будет техническим, хотя некоторые менеджеры по персоналу задавали мне технические вопросы, ответы на которые выучили на предыдущих собеседованиях. Разговор будет идти о технологиях, которыми вы владеете, вашем предыдущем опыте, а также о том, сможете ли вы адаптироваться к корпоративной культуре компании.

Если вы хорошо себя проявите, вас пригласят на второй раунд — техническое собеседование по телефону, — во время которого вы будете общаться с членами

команды инженеров. Они будут задавать те же вопросы, что и во время первого собеседования, но в этот раз также предложат вам пройти технический тест. Инженеры сообщат адрес сайта, на котором опубликованы вопросы по программированию, и попросят ответить на них по телефону.

Если вы справитесь, вам должны назначить третье собеседование, которое обычно проводится лично в офисе компании. Там вы снова встретитесь с инженерами, которые расспросят вас о ваших навыках и предложат пройти еще кое-какие технические тесты. Вас могут попросить остаться на обед, чтобы проверить, как вы взаимодействуете с командой. В третьем раунде и проводят знаменитые тесты с белой доской или листом бумаги и просят накидать программу. Если компания, в которой вы проходите собеседование, прибегает к таким тестам, вас попросят решить несколько программистских задач. Советую купить доску и предварительно попрактиковаться, поскольку решать задачи на доске гораздо сложнее, чем на компьютере.

Подготовьтесь к собеседованию

В большинстве случаев на собеседованиях по программированию все внимание уделяется двум вещам — структурам данных и алгоритмам. Поэтому для успешного его прохождения улучшите свои навыки в этих областях компьютерной науки. К тому же это поможет вам стать лучшим программистом.

Вы можете еще больше сузить спектр вопросов, если посмотрите на ситуацию со стороны интервьюера. Подумайте о том, в каком он положении; интервьюеры говорят, что программное обеспечение не бывает совершенным, и они правы. У вашего интервьюера наверняка полно работы, и он не желает посвящать много времени отбору кандидатов. Будет ли он пытаться придумать оригинальные вопросы? Скорее всего, нет. Вместо этого он наберет в поиске Google «вопросы по программированию для собеседования» и будет задавать вам первые же, которые найдет. Из-за этого вопросы на собеседованиях раз за разом повторяются, появились даже ресурсы, где можно потренироваться отвечать на них! Я настоятельно рекомендую ресурс LeetCode — там я нашел все вопросы, которые мне когда-либо задавали на собеседованиях по программированию.

Глава 25. Работа в команде

> Нельзя создавать отличное программное обеспечение
> без отличной команды, а большинство программистских команд
> напоминают неблагополучные семьи.
>
> *Джим Маккарти*

Будучи программистом-самоучкой, привыкаешь программировать в одиночку. Но как только вы начнете работать в компании, вам нужно будет научиться взаимодействовать с командой. Даже если вы создадите собственную компанию, в конце концов, вы наймете дополнительных программистов, и тогда все же придется научиться работать в команде. Программирование — командный вид спорта, и, как и во всяком командном виде спорта, здесь нужно уметь ладить со сво-

ими товарищами по команде. В этой главе представлены советы по успешной работе в командной среде.

Освойте базис

При приеме на работу от вас ожидают хорошего владения навыками, описанными в этой книге. Ее недостаточно просто прочесть — вы должны быть экспертом в представленных концепциях. Ваши товарищи по команде не будут в восторге, если им постоянно придется помогать вам с основами.

Не задавайте вопросы, ответы на которые можете найти в Google

Как программисту-самоучке, вам нужно будет много чего изучить — и у вас появится множество вопросов. Задавать вопросы — это отличный способ учиться, но убедитесь, что вы задаете правильные вопросы. Задавайте вопрос, только если потратили минимум пять минут, пытаясь самостоятельно найти ответ в Google. Если вы задаете слишком много вопросов, ответы на которые могли бы найти сами, это будет раздражать ваших товарищей по команде.

Изменение кода

Тем, что вы читаете эту книгу, вы доказали, что стремитесь к постоянному самосовершенствованию. К сожалению, не все в вашей команде будут разделять энтузиазм относительно улучшения программистских навыков. У многих программистов нет желания продолжать учиться — их устраивает и неоптимальное решение задач.

Некачественный код особенно распространен в стартапах, где оперативное написание кода часто важнее качества. Если вы оказались в подобной ситуации, действуйте осторожно. Редактирование чьего-то кода может больно ударить по эго автора. Даже хуже, если вы будете тратить много времени на изменение чужого кода, у вас не останется времени на работу над новыми проектами, и со стороны может показаться, что вы трудитесь недостаточно усердно. Лучший способ избежать таких неприятностей — это осторожно спросить на собеседовании об инженерной культуре в компании. Если вы по-прежнему попадаете в такие ситуации, прислушайтесь к Эдварду Йордану: «Если вы считаете, что менеджмент не в курсе того, что происходит, или ваша организация производит низкокачественное программное обеспечение, за которое вам стыдно, — уходите».

Синдром самозванца

Все программисты порой ощущают себя перегруженными, и вне зависимости от того, насколько усердно вы работаете, вам будут попадаться вещи, которых вы не знаете. Будучи программистом-самоучкой, особенно легко почувствовать, что вы не дотягиваете по уровню, когда кто-то просит вас сделать что-то, о чем вы никогда не слышали. Или вы можете обнаружить, что в программировании существует множество концепций, которые вам не понятны — такое случается со всеми, не только с вами.

Я удивился, когда мой друг из Стэнфордского университета со степенью магистра по компьютерным наукам сказал мне, что тоже чувствует подобное. Он сказал, что каждый в его программе сталкивался с синдромом самозванца. При этом было два вида реакции: люди либо покорно признавали, что они чего-то не знают, и начинали этому учиться, либо делали вид, что во всем прекрасно разбираются (что на самом деле было не так) и не собирались учиться. Помните, что вы оказались там, где оказались, благодаря тяжелому труду, и нет ничего плохого в том, что вы не знаете всего — никто не знает. Просто смиренно и беспрерывно изучайте все, чего не понимаете, и тогда вас будет не остановить.

Глава 26. Дальнейшее обучение

> Разница между лучшими программистами и просто хорошими является существенной. Показатели креативности, скорости, изобретательности или способности быстро решать проблемы первых на порядок лучше при любом способе измерения.
>
> *Рэндалл Э. Стросс*

В статье Дэвида Биттоу «НПП: Не переставай программировать» дается отличный совет насчет получения работы инженера-программиста. Название говорит само за себя — не переставай программировать. Статью можно найти на сайте **medium.com/always-be-coding/abc-always-be-coding-d5f8051afce2**. Если вы соедините НПП с моим акронимом НПУ — не переставай учиться — у вас обязательно получится выдающаяся карьера. В этой главе я расскажу о некоторых программистских ресурсах, которые мне показались полезными.

Классика

Существует несколько книг по программированию, которые считаются обязательными к прочтению.

- «Программист-прагматик» Энди Ханта и Дэвида Томаса;
- «Приемы объектно-ориентированного проектирования. Паттерны проектирования» Эриха Гаммы, Ральфа Джонсона, Джона Влиссидеса и Ричарда Хелма (шаблоны проектирования — важная тема, которую у меня не было возможности раскрыть);
- «Совершенный код» Стива Макконнелла;
- «Компиляторы: принципы, технологии и инструменты» Альфреда Ахо, Джеффри Ульмана, Моники Лам и Рави Сети;
- «Алгоритмы: построение и анализ» Томаса Кормена, Чарльза Лейзерсона, Рональда Ривеста и Клиффорда Штайна.

Онлайн-курсы

Онлайн-уроки — еще один способ улучшать навыки программирования. Все мои рекомендации по этой теме можно найти на сайте **theselftaughtprogrammer.io/courses**.

Платформа Hacker News

Hacker News — это социально-новостная платформа, расположенная на сайте технологического бизнес-инкубатора Y Combinator по адресу **news.ycombinator.com**. Здесь можно следить за последними трендами и технологиями.

Глава 27. Следующие шаги

Полюби ту скромную профессию, которой ты овладел, и будь доволен ею.

Марк Аврелий

Прежде всего — спасибо вам за покупку этой книги. Я надеюсь, она помогла вам стать лучше в программировании. Теперь, когда вы закончили, пришло время приниматься за дело. Куда отправиться дальше? Структуры данных и алгоритмы. Заходите на сайт LeetCode и практикуйтесь в алгоритмах. А затем снова практикуйтесь! В этой главе я выкладываю заключительные мысли о том, как вы можете продолжить совершенствоваться в программировании (сразу, как только закончите практиковаться в алгоритмах).

Найдите себе наставника

Наставник выведет ваши навыки программирования на новый уровень. Одна из наиболее неприятных деталей в обучении программированию заключается в том, что вы можете выполнять множество вещей неоптимальным способом и даже не догадываться об этом. Ранее я упоминал, что с этим можно бороться при помощи обзоров кода. Наставник может анализировать ваш код, помогая вам усовершенствовать процесс написания кода, а также советовать книги и объяснять программистские концепции, которые вам не понятны.

Копайте глубже

В программировании есть термин «черный ящик», обозначающий нечто, что вы используете, при этом не понимая, как оно работает. Когда вы только начинаете программировать, для вас все — черный ящик. Один из главных способов стать лучше в программировании — открывать каждый найденный черный ящик и пытаться понять, как он работает. Один мой друг рассказал о моменте восторга (Бинго!), когда он понял, что командная строка сама по себе является программой. Когда я говорю «копать глубже», я подразумеваю открытие черного ящика.

Написание этой книги помогло мне копнуть глубже. Я столкнулся с определенными концепциями, которые, как я полагал, были мне понятны, и не мог их объяснить. Мне пришлось копнуть глубоко. Не останавливайтесь на одном ответе, прочтите все объяснения, которые сможете отыскать. Задавайте вопросы в Интернете и читайте различные мнения.

Еще один способ копать глубже — создавать вещи, которые вы хотите понять лучше. Есть проблемы с пониманием управления версиями? На досуге создайте простую систему управления версиями. Выделение времени на подобные дела —

грамотная инвестиция. Это улучшит понимание всего, с чем бы вы ни испытывали сложности.

Другие советы

Однажды я набрел на форум, где обсуждались различные способы стать лучше в программировании. Ответ, набравший больше всего голосов, звучал так: занимайтесь чем-то помимо программирования. Мне это показалось правильным — чтение книг вроде «Кода таланта» Дэниела Койла позволило мне стать лучшим программистом благодаря тому, что автор излагает именно то, что вам нужно для овладения любым навыком. Следите за вещами, не связанными с программированием, которые можно использовать в этой области.

Последний совет, который мне хотелось бы вам дать, — проводите как можно больше времени за чтением кода других людей. Это один из лучших способов улучшить свои навыки программиста. Во время обучения старайтесь поддерживать баланс между чтением и написанием кода. Чтение чужого кода поначалу будет сложным, но это важно, ведь вы можете очень многому научиться у других программистов.

Я надеюсь, вы получили от чтения этой книги настолько же сильное удовольствие, как я получил от ее написания.

Желаю удачи в вашем дальнейшем путешествии.

Предметный указатель

КОГДА ВЫ ДАРИТЕ КНИГУ, ВЫ ДАРИТЕ ЦЕЛЫЙ МИР

ХОТИТЕ ЗНАТЬ БОЛЬШЕ?

Заходите на сайт:
https://eksmo.ru/b2b/

Звоните по телефону:
+7 495 411-68-59, доб. 2261

ВАШ ЛОГОТИП НА ОБЛОЖКЕ

ВАШ ЛОГОТИП НА КОРЕШКЕ

ОБРАЩЕНИЕ К КЛИЕНТАМ НА ОБЛОЖКЕ

#САМ СЕБЕ ПРОГРАММИСТ.

Как научиться программировать и устроиться в eBay?

Автор этой книги знает, как!

Всего за год он самостоятельно освоил программирование на Python и стал разработчиком в одной из ведущих мировых IT–компаний.

Этот самоучитель отличается от большинства других книг по программированию для начинающих, ведь уже после прочтения первой главы вы напишете собственную небольшую программу, а к концу книги – нет, вы не станете супер-программистом, потому что для этого нужны годы упорного труда, – но вы будете уверенно писать код.
//Более того, автор расскажет вам, как успешно проходить собеседования на должность программиста в любую компанию.

Уже видите себя в кресле разработчика? Тогда вперед!

«Если у вас нет опыта программирования или вы не писали ничего сложнее “Hello world”, эта книга станет хорошим стартом для первого погружения в мир разработки. На примере Python автор рассматривает базовые конструкции и концепции программирования, которые применимы и к другим популярным языкам, и дает множество советов о том, как развиваться дальше. Если вы хотите стать профессиональным разработчиком, то каждую из рассмотренных тем нужно будет разбирать глубже. Но получив благодаря книге базовое представление о разработке, вы сможете понять, в каких направлениях можно развиваться дальше, и сфокусироваться на них.»

Алексей Кудрявцев,
разработчик мобильных приложений, Avito

ISBN 978-5-04-090834-9

БОМБОРА

Бомбора — это новое название Эксмо Non-fiction, лидера на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bomborabooks
www.bombora.ru