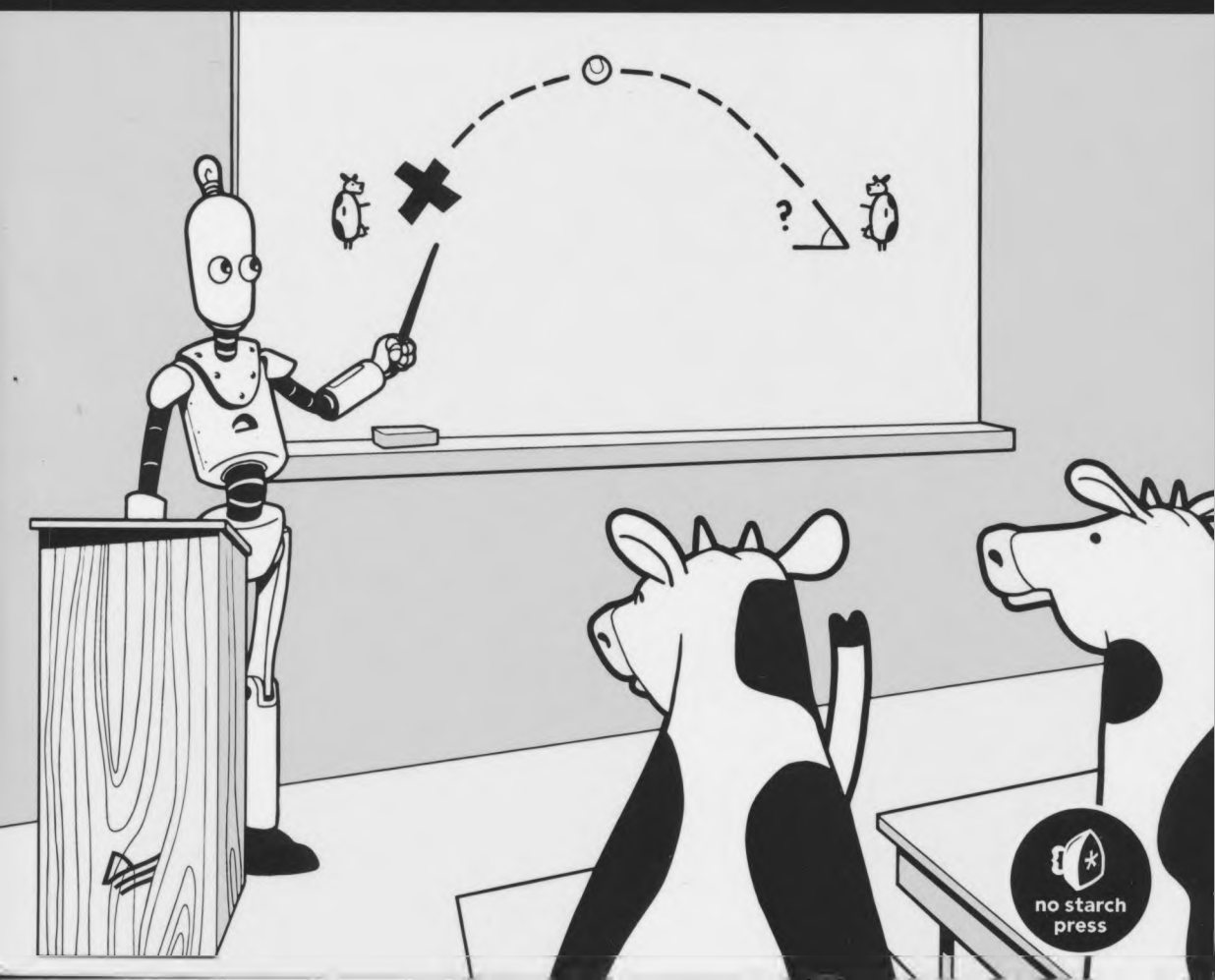
PYTHON
БЕЗ ПРОБЛЕМ

РЕШАЕМ РЕАЛЬНЫЕ ЗАДАЧИ
И ПИШЕМ ПОЛЕЗНЫЙ КОД

ДАНИЭЛЬ ЗИНГАРО



LEARN TO CODE BY SOLVING PROBLEMS

A Python Programming Primer

by Daniel Zingaro

no starch
press

San Francisco

PYTHON
БЕЗ ПРОБЛЕМ

РЕШАЕМ РЕАЛЬНЫЕ ЗАДАЧИ И ПИШЕМ ПОЛЕЗНЫЙ КОД

ДАНИЭЛЬ ЗИНГАРО

ПИТЕР®

Санкт-Петербург · Москва · Минск

2023

ББК 32.973.2-018.1
УДК 004.43
З-63

Зингаро Даниэль

З-63 Python без проблем: решаем реальные задачи и пишем полезный код. — СПб.: Питер, 2023. — 336 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1920-2

Компьютеры — это мощные машины для решения задач, способные делать практически все, если им дать правильные инструкции. Вот тут-то и приходит на помощь программирование. Эта книга поможет начинающим питонистам сразу создавать программы, поскольку знакомит с языком через решение задач, которые использовались на реальных соревнованиях по кодингу.

Практикуясь в использовании основных функций, функций и методов, вы разберетесь со структурами данных, алгоритмами и другими основополагающими аспектами программирования, полезными на любом языке.

К концу книги вы не только овладеете Python, но и научитесь тому типу мышления, который необходим для решения задач. Потому что языки программирования приходят и уходят, а способ решения проблем — нет!

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1718501324 англ.

ISBN 978-5-4461-1920-2

Краткое содержание

Оглавление

Моему отцу, который научил понимать компьютеры, и моей матери, которая научила понимать людей.

Об авторе

Доктор Даниэль Зингаро — адъюнкт-профессор информатики и преподаватель в Университете Торонто, обладатель множества наград. Его основная область исследований — образование в сфере информатики и вопросы, связанные с тем, как студенты осваивают (пусть порой и плохо) информатику. Даниэль написал книгу *Algorithmic Thinking*[1] (No Starch Press, 2021), которая помогает студентам изучать и использовать алгоритмы и структуры данных.

О научном редакторе

Люк Савчак — редактор-фрилансер и программист-любитель. Ему нравится превращать прозу в стихи, он написал пособие по нарезанию нужного количества кусочков торта и заумную версию Boggle, разобраться в которой сможет только преподаватель математики. Сейчас он преподает французский и английский языки на окраине Торонто. Люк также пишет стихи и сочиняет музыку для фортепиано, чем с удовольствием зарабатывал бы себе на жизнь, если бы мог. Его сайт: https://sawczak.com/.

[1] *Зингаро Д.* Алгоритмы на практике. — СПб.: Питер, 2023.

Благодарности

Неужели мне снова довелось поработать с ребятами из No Starch Press? Барбара Йен и Билл Поллок предложили мне написать эту книгу. Алекс Фрид, ведущий редактор, помог в работе над рукописью. Я благодарю всех, кто участвовал в создании книги, включая редактора Кима Вимпсетта, выпускающего редактора Кэсси Андредис и дизайнера Роба Гейла. Мне очень повезло с ними.

Спасибо Университету Торонто, который предоставил мне время и место для написания книги. Я благодарю Люка Савчака, моего научного редактора, за исчерпывающую рецензию.

Я благодарен всем, кто занимался решением задач, описанных в этой книге, и соревновательным программированием в целом. Спасибо администраторам DMOJ за поддержку моей работы.

Благодарю своих родителей за то, что они поистине *способны на все*. А меня они просили лишь об одном — учиться.

Спасибо Дояли за то, что позволила мне уделить достаточно времени написанию книги и понимала, как много для этого требуется.

Наконец, благодарю всех вас за то, что вы купили эту книгу и хотите учиться.

Введение

Компьютеры нужны нам для выполнения каких-то задач и решения проблем. Наверняка вы использовали текстовый редактор, чтобы написать эссе или письмо. Возможно, вы пользовались программами для работы с электронными таблицами, чтобы свести дебет с кредитом. Быть может, с помощью графического редактора чуть-чуть обрабатывали свои фотографии. Сейчас трудно себе представить, как можно было бы сделать все это без компьютера. И текстовые процессоры, и электронные таблицы, и графические редакторы невероятно полезны.

Эти программы — инструменты общего назначения, предназначенные для выполнения широкого круга задач. Вот только написали их другие люди, не мы. А что делать, если имеющаяся программа не дает нам всего необходимого?

В книге наша цель — понять свой компьютер и выйти за рамки того, что пользователь может сделать с помощью уже существующих программ. Если точнее, вы будете писать собственные программы. Но это не будет текстовый или графический редактор либо же редактор таблиц. Это огромные задачи, которые, к счастью, до нас уже решили. Вы всего лишь научитесь писать небольшие программы для решения задач, с которыми в ином случае не справились бы. Я помогу вам научиться давать компьютеру команды, которые подскажут ему, как шаг за шагом решить вашу задачу.

Чтобы отдать компьютеру команду, мы пишем код на *языке программирования*. Он определяет правила написания кода и действия компьютера в ответ на этот код.

Вы будете учиться программировать на языке Python. Это серьезный навык, который не стыдно будет упомянуть в резюме. Но помимо освоения самого Python, вы научитесь тому типу мышления, который необходим для решения проблем

с помощью компьютера. Языки программирования появляются и исчезают, а вот алгоритмический подход к задачам — нет. Я надеюсь, что эта книга поможет вам превратиться из конечного пользователя в программиста и вы получите удовольствие от изучения предложенного материала.

Интернет-ресурсы

Ресурсы для работы с книгой, включая примеры кода и дополнительные упражнения, можно скачать, перейдя по ссылке https://nostarch.com/learn-code-solving-problems/ .

Для кого предназначена книга

Эта книга для всех, кто хочет научиться писать компьютерные программы для решения своих задач.

Во-первых, вы, возможно, слышали о языке программирования Python и хотите научиться писать код на нем. В следующем разделе я объясню, почему именно Python отлично подходит в качестве первого языка программирования для изучения. Здесь вы много узнаете о Python и позже сможете прочесть более сложные книги об этом языке.

Во-вторых, если вы не слышали о Python или просто хотите узнать, что такое программирование, не волнуйтесь, вам эта книга тоже подойдет! Она научит вас понимать суть программирования. У программистов есть свои методы разбивать задачи на небольшие части и находить их решения с помощью кода. На этом уровне не имеет значения, какой именно язык программирования используется, потому что мышление программиста не привязано к какому-либо определенному языку.

И в-третьих, вам может быть интересно изучить какой-нибудь другой язык программирования, например C ++, Java, Go или Rust. Многое из того, что вы узнаете в ходе изучения Python, будет полезно при усвоении других языков программирования. Кроме того, Python сам по себе заслуживает изучения. А почему именно он — об этом дальше.

Зачем изучать Python

За годы преподавания я понял, что Python отлично подходит в качестве первого языка программирования. По сравнению с кодом на других языках код Python часто бывает лучше структурирован и более удобочитаем. Как только вы привыкнете к нему, окажется, что зачастую он читается почти как обычный английский!

Кроме того, у Python есть то, чего нет в других языках, например мощные инструменты для управления данными и их хранения. Многие из этих инструментов мы будем использовать в книге.

Python — это не только отличный язык для обучения, но еще и один из самых востребованных языков программирования в мире. Его используют для написания веб-приложений, игр, средств визуализации, ПО для машинного обучения и многого другого.

В итоге получаем язык, который и для обучения хорош, и для профессиональной деятельности полезен. О большем и мечтать нельзя!

Установка Python

Чтобы начать программировать на Python, нужно его сперва установить. С этого и начнем.

В основном используются две версии языка: Python 2 и Python 3. Python 2 — это более старая версия, которая больше не поддерживается. В этой книге мы будем работать с Python 3, именно ее следует установить на свой компьютер.

Python 3 далеко шагнул по сравнению с Python 2, но и в версии 3 он постоянно меняется. Первой версией Python 3 был Python 3.0. Затем выпустили Python 3.1, Python 3.2 и т. д. На момент написания книги последней версией Python 3 был Python 3.9. Для решения примеров из этой книги достаточно будет версии Python 3.6, но я рекомендую установить последнюю версию Python и поработать с ней.

Чтобы установить Python, выполните приведенные далее инструкции для вашей операционной системы.

Windows

В ОС Windows Python по умолчанию не установлен. Чтобы установить его, на сайте https://www.python.org/ перейдите в раздел Downloads. Вам будет предложено загрузить последнюю версию Python для Windows. Щелкните на ссылке, чтобы загрузить Python, а затем запустите установщик. На одном из первых экранов процесса установки выберите опцию Add Python 3.9 to PATH или Add Python to environment variables — это значительно упрощает запуск. (При обновлении Python вам может потребоваться нажать кнопку Customize installation, чтобы увидеть этот вариант.)

macOS

В macOS по умолчанию Python 3 не установлен. Чтобы установить его, на сайте https://www.python.org/ перейдите в раздел Downloads. Вам будет предоставлена возможность загрузки последней версии Python для macOS. Щелкните на ссылке, чтобы загрузить Python, а затем запустите установщик.

Linux

В Linux Python 3 установлен по умолчанию, но это может быть более старая версия, чем нам нужно. Инструкции по установке могут различаться в зависимости от того, какой дистрибутив Linux вы используете. Установить новейшую версию Python можно с помощью диспетчера пакетов.

Как читать эту книгу

Если вы прочитаете эту книгу от корки до корки за один присест, то мало чему научитесь. Это все равно что пытаться научиться играть на фортепиано, пригласив домой пианиста и посмотрев на него часик-другой, а затем выгнать его, радостно напевая песенку. Практические навыки так не усваиваются.

Я расскажу, как стоит читать эту книгу.

- **Распределите работу равномерно.** Усваивать информацию большими порциями гораздо менее эффективно, чем разбив работу на маленькие фрагменты. Как только почувствуете усталость, сделайте перерыв. Никто не сможет сказать вам, сколько времени нужно работать до перерыва. И точно так же никто не может предугадать, сколько времени потребуется на изучение всей книги. Это зависит от ваших возможностей.
- **Делайте паузы, чтобы проверить, все ли вы поняли.** Читая что-то новое, мы часто думаем, что поняли новый материал лучше, чем это есть на самом деле. Приходится тратить время на согласование того, что мы знаем и что мы думаем, что знаем. Именно поэтому я добавил к ключевым моментам каждой главы парочку вопросов с несколькими вариантами ответов, которые помогут вам понять, что к чему. Отнеситесь к ним серьезно! Прочтите каждый вопрос и дайте ответ, не проверяя ответы на компьютере. Затем прочтите мой ответ и пояснение к нему. Это позволит убедиться, что вы на правильном пути. Если же ответили неправильно или ответили правильно, но по неправильной причине, устраните пробелы в знаниях, прежде чем двинуться дальше. Возможно, вам придется еще поупражняться с соответствующей функцией

Python или перечитать материал из книги, поискать в Интернете дополнительную информацию или примеры.

- **Практикуйтесь в программировании.** Практика во время чтения поможет вам закрепить понимание ключевых моментов. Но вам нужно нечто большее, чтобы искусно решать задачи и стать программистом. Вам нужно попрактиковаться в использовании Python для решения задач, которых в этой книге не будет, которые окажутся новыми и незнакомыми. В конце каждой главы приведен список практических упражнений. Постарайтесь выполнить как можно больше.

Чтобы научиться программировать, требуется время. Не расстраивайтесь, если вы будете двигаться слишком медленно или станете делать слишком много ошибок. И не пугайтесь напыщенных павлинов, которых немало в Интернете. Лучше побольше общайтесь с людьми, которые могут вам помочь.

Сайты с задачами для программистов

За основу для этой книги я взял задачи из онлайн-задачников для программистов. На таких сайтах есть репозитории задач по программированию, которые решают любители со всего мира. Вы можете отправить туда свое решение в виде кода Python — и сайт протестирует его. Если код даст правильный ответ для всех тестов, то, скорее всего, ваше решение правильное. Если же он дает неправильный ответ для одного или нескольких тестовых примеров, значит, его требуется доработать.

Есть ряд причин, по которым именно сайты с задачами для программистов нравятся мне больше других сайтов для обучения программированию.

- **Быстрая обратная связь.** Быстрая и целенаправленная обратная связь имеет решающее значение на ранних этапах обучения программированию. Ответ можно получить сразу же, как только вы отправите свой код.
- **Качественные задачи.** Я считаю, что задачи по программированию на таких сайтах действительно хороши. Одни использовались на различных соревнованиях по программированию. Другие написаны людьми, которые связаны с такими сайтами или просто хотят помочь всем желающим в обучении.
- **Количество задач.** В онлайн-задачнике вы найдете сотни задач. Для этой книги я отобрал лишь малую их часть. Если вам нужно побольше практики, поверьте, на таком сайте вы найдете все необходимое.
- **Помощь сообщества.** Сайт с задачами для программистов позволяет пользователям читать комментарии и отвечать на них. Если вы застряли на какой-то проблеме, просмотрите комментарии — и найдете в них подсказки, оставленные другими. Если это не помогает, можете написать собственный комментарий и попросить людей о помощи. Но даже после успешного реше-

ния задачи ваше обучение не заканчивается, ведь онлайн-задачник позволяет просматривать код, присланный другими. Изучите уже опубликованные материалы и сравните их со своим решением. Всегда есть несколько способов справиться с задачей. Возможно, поначалу вы будете принимать решения интуитивно, но позже вам откроются новые возможности. Общение — важный шаг в овладении программированием.

Создание учетной записи

По ходу книги мы будем пользоваться сразу несколькими сайтами с задачами для программистов. Все дело в том, что на каждом из них размещены уникальные задачи, которых нет на других сайтах, так что потребуется несколько онлайн-задачников, чтобы охватить все темы, которые я хотел бы осветить.

Далее приведены сайты, с которыми мы будем работать:

- DMOJ — https://dmoj.ca/;
- Timus — https://acm.timus.ru/;
- USACO (USA Computing Olympiad) — http://usaco.org/.

На каждом сайте вам нужно будет создать учетную запись, прежде чем вы сможете отправлять код. Поговорим о процессе создания учетной записи и больше узнаем о каждом из сайтов.

DMOJ

Сайтом DMOJ мы будем пользоваться по ходу книги чаще всего. Вам стоит уделить побольше времени его изучению и узнать о том, что на нем имеется.

Чтобы создать учетную запись на сайте DMOJ, перейдите по ссылке https://dmoj.ca/ и нажмите кнопку Sign up. На странице регистрации введите свои имя пользователя, пароль и адрес электронной почты. Здесь же можете выбрать язык программирования по умолчанию. В этой книге мы будем применять исключительно язык программирования Python, поэтому выберите Python 3. Нажмите Register!, чтобы завершить создание учетной записи. После регистрации вы сможете заходить на DMOJ с помощью своих имени пользователя и пароля.

Все задачи в книге начинаются с указания веб-сайта, на котором вы найдете саму задачу и код для доступа к ней. Например, первая задача, над которой мы будем работать в главе 1, находится на DMOJ и доступна по коду `dmopc15c7p2`. Чтобы найти ее в DMOJ, щелкните на кнопке Problems, введите код `dmopc15c7p2` в поле поиска и нажмите Go. Перед вами появится соответствующая задача. Щелкнув на заголовке, вы должны увидеть саму задачу.

Когда будете готовы отправить свой код Python с решением, найдите задачу и щелкните на кнопке Submit solution. На открывшейся странице вставьте код в текстовое поле и нажмите Submit!. Он будет оценен, и вы увидите результаты.

Timus

Чтобы создать учетную запись на сайте Timus, перейдите по ссылке https://acm.timus.ru/ и нажмите кнопку Register. На открывшейся странице регистрации введите свои имя, пароль, адрес электронной почты и прочую запрашиваемую информацию. Нажмите кнопку Register, чтобы создать учетную запись. Затем проверьте свою электронную почту на наличие сообщения от Timus со своим уникальным идентификатором. При отправке кода Python он вам понадобится.

На этом сайте пока нет возможности установить язык программирования по умолчанию, поэтому при каждой отправке кода обязательно выбирайте доступную версию Python 3.

С сайтом Timus мы будем работать лишь один раз, в главе 6, поэтому пока что разговор о нем отложим.

USACO

Чтобы создать учетную запись на сайте USACO, перейдите по ссылке http://usaco.org/ и нажмите на кнопку Register for New Account. На открывшейся странице регистрации введите свои имя пользователя, адрес электронной почты и прочую информацию. Нажмите кнопку Submit, чтобы создать учетную запись. Затем проверьте свою электронную почту на наличие сообщения от USACO, содержащего пароль. Получив его, вы можете заходить на USACO с помощью своих имени пользователя и пароля.

На этом сайте пока нет возможности установить язык программирования по умолчанию, поэтому при каждой отправке кода обязательно указывайте доступную версию Python 3. Вам также нужно будет выбрать файл с кодом Python, а не вставлять сам код в текстовое поле.

Мы не будем использовать сайт USACO до главы 7, поэтому пока что отложим разговор о нем.

Об этой книге

Все главы книги построены вокруг двух-трех задач с одного из сайтов с задачами для программистов. Фактически глава будет начинаться с постановки первой задачи, и уже после этого вы начнете изучать Python! Моя цель — побудить вас изучить функции Python, необходимые для решения задачи. Не беспокойтесь, если решение задачи не придет вам в голову сразу после прочтения ее описания (ведь если вы

еще не можете решить задачу, значит, читаете нужную книгу!). Если же понимаете, что нужно делать, чтобы ее решить, то все отлично. Мы будем изучать Python и вместе решать задачи. Следующие задачи будут нацелены на использование дополнительных возможностей Python или закрепление и расширение того, что вы узнали в первой задаче. Каждая глава завершается упражнениями, которые вы должны будете выполнить самостоятельно, чтобы попрактиковаться в применении вновь обретенных навыков.

Приведу краткое описание каждой главы.

- **Глава 1. Приступим к работе.** Существует немало вводных понятий, которые нужно будет изучить, прежде чем вы сможете решать какие-либо задачи с помощью Python. В этой главе мы рассмотрим их, начиная с написания простейшего кода Python, затем перейдем к работе со строками и числами, использованию переменных, чтению ввода и записи вывода.
- **Глава 2. Принятие решений.** В этой главе вы узнаете об операторе `if`, который позволяет программе решать, что делать, в зависимости от истинности или ложности того или иного условия.
- **Глава 3. Повторяющийся код: определенные циклы.** Многие программы работают некоторое время, пока не закончатся входные данные. В этой главе мы изучим цикл `for`, который позволяет программам обрабатывать входные данные, пока работа не будет выполнена.
- **Глава 4. Повторяющийся код: неопределенные циклы.** Иногда мы не знаем заранее, сколько раз программа должна будет выполнить некоторые действия. Циклы `for` для такого рода задач не подходят. В этой главе мы поговорим о цикле `while`, который повторяет код, пока выполняется определенное условие.
- **Глава 5. Упорядоченные значения и списки.** Списки в Python позволяют использовать одно имя и ссылаться с его помощью на целую последовательность данных. Списки помогают организовать данные, в Python предусмотрены удобные операции обработки списков, такие как сортировка и поиск. В этой главе вы узнаете все о списках.
- **Глава 6. Пишем собственные функции.** Большая программа со значительным количеством кода без должной его организации может стать громоздкой. В этой главе вы узнаете о функциях, которые помогают разбивать программы на небольшие автономные фрагменты кода. Применение функций позволяет писать программы, которые намного легче читаются и редактируются. Мы также поговорим о нисходящем проектировании — подходе к разработке программ с функциями.
- **Глава 7. Чтение из файлов и запись в них.** Для передачи данных программам или сохранения результатов их работы удобно использовать файлы. В этой главе вы узнаете, как читать данные и записывать их в файлы.

- **Глава 8. Организация данных с помощью множеств и словарей.** Когда мы начинаем решать все более сложные задачи, важно правильно организовать хранение данных. В этой главе поговорим о двух новых инструментах для хранения данных в Python — множествах и словарях.
- **Глава 9. Разработка алгоритмов полного поиска.** Программисты не придумывают решение каждой задачи с нуля. Вместо этого они стараются тем или иным способом использовать общий шаблон решения — *алгоритм*. В этой главе вы узнаете об алгоритмах полного поиска, которые можно задействовать для решения широкого круга задач.
- **Глава 10. «О большое» и эффективность программ.** Иногда нам удается написать программу, которая выполняет задачу правильно, но слишком медленно, из-за чего оказывается бесполезной. В этой главе мы обсудим, как определить эффективность программ, и поговорим об инструментах, которые можно использовать для написания более эффективного кода.

От издательства

В книге приведены ссылки на англоязычные сайты-задачники. Вы можете работать с их переводами, воспользовавшись сайтом https://translate.yandex.ru/translate.

Например, задача по адресу https://dmoj.ca/problem/ccc06j1 на русском языке будет выглядеть так: https://translated.turbopages.org/proxy_u/en-ru.ru.c31992eb-62d7fa12-e0216655-74722d776562/https/dmoj.ca/problem/ccc06j1.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Приступим к работе

Программирование — это написание кода для решения некоторой задачи. Этим мы и займемся. Поэтому не будем сначала изучать концепции Python, а затем формулировать задачи. Напротив, я сформулирую задачу, а уже на ее основе стану вводить концепции, которые нужно изучить.

В этой главе мы решим две задачи: определим количество слов в строке (этим же занимаются функции подсчета слов в текстовом редакторе) и вычислим объем конуса. Для этого вам придется познакомиться с несколькими концепциями Python. В какой-то момент вы можете почувствовать, что вам нужно больше подробностей, чтобы полностью понять приведенный материал и то, как все это сочетается друг с другом при разработке программы на Python. Не волнуйтесь: в следующих главах мы вернемся к наиболее важным понятиям и подробно остановимся на них.

Что мы будем делать

Как упоминалось во введении, мы будем решать задачи из области соревновательного программирования с использованием языка Python. Все представленные задачи по соревновательному программированию можно найти на одном из сайтов с задачами для программистов. Я предполагаю, что вы следовали инструкциям, приведенным во введении: установили Python и зарегистрировались на сайтах.

Мы напишем программы для решения всех приведенных задач. У каждой из них есть входные данные (ввод) определенного типа, которые программа будет

получать, и ожидаемые выходные данные (вывод), тоже определенного типа. Будем считать, что программа правильно решает задачу, если может принимать любые допустимые входные данные и выдает в ответ правильные выходные данные.

В целом возможных входных данных могут быть миллионы или миллиарды. Каждый вариант входных данных называется *экземпляром задачи*. Например, в первой задаче, которую мы решим, входные данные — это строка текста, например `hello there` или `bbaabbb aa abab`. Наша цель — подсчитать и вывести количество слов в строке. Одна из самых крутых вещей в программировании заключается в том, что зачастую небольшой объем универсального кода позволяет решить бесконечное количество типовых задач. И не имеет значения, будет в строке два слова, три или 50 — программа всегда будет делать это правильно.

Наши программы будут выполнять три задачи.

- **Чтение входных данных.** Необходимо определить конкретный экземпляр задачи, которую требуется решить, поэтому сначала мы считываем предоставленные входные данные.
- **Обработка.** Мы обрабатываем входные данные, чтобы определить правильные выходные данные.
- **Запись вывода.** Решив задачу, выдаем желаемый вывод.

Границы между шагами бывают размытыми — иногда нам, возможно, придется чередовать обработку с получением результата. Но все равно работа в целом делится на эти три этапа.

Вы, вероятно, ежедневно пользуетесь программами, работающими в соответствии с моделью «ввод — обработка — вывод». Рассмотрим программу-калькулятор: вы вводите формулу (входные данные), программа обрабатывает числа (обработка), а затем отображает ответ (выходные данные). Или вспомним о поисковых системах в Интернете: вы вводите поисковый запрос (входные данные), поисковая система определяет наиболее релевантные результаты (обработка) и отображает их (выходные данные).

Сравните приведенные примеры программ с интерактивными программами, которые также выполняют ввод, обработку и вывод. Например, для набора текста этой книги я использую текстовый редактор. Когда я набираю символ, редактор в ответ добавляет его в мой документ. И мне не нужно печатать сразу весь документ, чтобы увидеть результат, — редактор интерактивно отображает его по мере печатания. В этой книге мы не будем писать интерактивные программы, но если после изучения этой книги вы заинтересуетесь их созданием, то радуйтесь: Python определенно подходит для этого.

Тексты всех задач можно найти здесь и на сайте. Однако они не всегда совпадают, потому что я переписал их ради единообразия внутри книги. Но не волнуйтесь: в моих формулировках вы найдете ту же информацию, что и в официальной постановке задачи.

Оболочка Python

Для каждой задачи из книги мы хотим написать программу и сохранить ее в отдельном файле. Но для начала неплохо было бы знать, какую программу писать! Для решения многих задач вам нужно будет изучить пару новых функций Python.

Лучший способ поэкспериментировать с функциями Python — использовать оболочку Python. Это интерактивная среда, в которой можно ввести `Python` и нажать клавишу Enter, а Python в ответ выведет результат. Как только вы получите достаточно знаний, чтобы решить текущую задачу, можно будет перестать работать с оболочкой и начать писать решение в текстовом файле.

Для начала создайте на рабочем столе новую папку и назовите ее `programming`. Будем использовать ее для хранения результатов работы по ходу изучения книги.

Сейчас мы откроем эту папку и запустим оболочку Python. Чтобы запустить оболочку Python в своей операционной системе, выполните следующие действия.

Windows

Если вы работаете в Windows, сделайте следующее.

1. Удерживая нажатой клавишу Shift, щелкните правой кнопкой мыши на папке с вашими программами, например `programming`.
2. В появившемся меню выберите пункт Open PowerShell window here (Открыть окно PowerShell здесь). Если в контекстном меню нет такого пункта, выберите Open command window here (Открыть окно команд здесь).
3. В появившемся окне вы увидите строку, которая заканчивается знаком «больше» (>). Это приглашение операционной системы, и теперь она ждет, когда вы наберете команду. Вводить здесь нужно именно команды операционной системы, а не код Python. Обязательно нажимайте клавишу Enter после каждой команды.
4. Вы находитесь в папке `programming`. Можете ввести команду `dir` (от англ. directory — «папка»), если хотите посмотреть содержимое папки. Впрочем, пока вы не увидите никаких файлов, потому что мы их еще не создавали.
5. Введите команду `python` для запуска оболочки Python.

Запустив оболочку Python, вы увидите что-то вроде этого:

```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:30:23)
[MSC v.1928 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Здесь важно, чтобы в первой строке была указана версия Python не ниже 3.6. Если у вас установлена более старая версия, особенно 2.x, или Python не загружается вообще, установите последнюю версию Python, следуя инструкциям, приведенным во введении.

Внизу этого окна вы увидите приглашение вида `>>>`. Здесь мы будем писать код Python. Никогда не вводите символы `>>>` сами. Закончив писать программу, вы можете нажать сочетание клавиш Ctrl+Z, а затем Enter, чтобы выйти.

macOS

В macOS сделайте следующее.

1. Откройте приложение Terminal. Для этого нажмите сочетание клавиш Command+Пробел, введите слово `terminal`, а затем дважды щелкните на результате.
2. В открывшемся окне вы увидите строку, оканчивающуюся символом доллара (`$`). Это приглашение операционной системы, теперь можете ввести команду. Здесь вводятся именно команды операционной системы, а не код Python. Обязательно нажимайте клавишу Enter после каждой команды.
3. Можете ввести команду `ls`, чтобы получить список файлов, находящихся в текущей папке. Вы увидите свой рабочий стол.
4. Введите команду `cd Desktop`, чтобы перейти в папку рабочего стола. Команда `cd` означает *change directory*, то есть «перейти в другую папку».
5. Введите команду `cd programming`, чтобы перейти в папку `programming`.
6. Теперь введите команду `python3`, чтобы запустить оболочку Python (можно попробовать ввести просто `python`, но в результате может запуститься старая версия Python 2, которая не подходит для работы с этой книгой).

Когда вы запустите оболочку Python, появится что-то вроде этого:

```
Python 3.9.2 (default, Mar 15 2021, 17:23:44)
[Clang 11.0.0 (clang-1100.0.33.17)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Здесь важно, что в первой строке указана версия Python не ниже 3.6. Если у вас более старая версия, особенно 2.x, или Python не загружается вообще, установите последнюю версию Python, следуя инструкциям, данным во введении.

Внизу этого окна увидите приглашение >>>. Здесь вы набираете код Python. Никогда не вводите символы >>> самостоятельно. Если закончили программирование, можете нажать сочетание клавиш Ctrl+D, чтобы выйти.

Linux

Порядок действий для Linux таков.

1. Щелкните правой кнопкой мыши на папке `programming`.
2. В появившемся меню выберите пункт Open in Terminal (Открыть в терминале) (а можно сначала открыть консоль и перейти в папку `programming`, если вам так удобнее).
3. В открывшемся окне вы увидите строку, оканчивающуюся символом доллара (`$`). Это приглашение операционной системы, теперь можете ввести команду. Здесь вводятся именно команды операционной системы, а не код Python. Обязательно нажимайте клавишу Enter после каждой команды.
4. Вы находитесь в папке `programming`. Можете ввести команду `ls`, чтобы просмотреть ее содержимое. Но пока никаких файлов в ней быть не должно, потому что мы их еще не создавали.
5. Теперь введите команду `python3`, чтобы запустить оболочку Python (можно попробовать ввести просто `python`, но в результате может запуститься старая версия Python 2, которая не подходит для работы с этой книгой).

Когда вы запустите оболочку Python, появится что-то вроде этого:

```
Python 3.9.2 (default, Feb 20 2021, 20:57:50)
[GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Важно, чтобы версия Python в первой строке была не ниже 3.6. Если у вас более старая версия, особенно 2.x, или Python не загружается вообще, установите последнюю версию Python, следуя инструкциям, приведенным во введении.

Внизу этого окна вы увидите приглашение Python вида >>>. Здесь мы будем писать код Python. Никогда не вводите символы >>> сами. Закончив писать программу, можете нажать сочетание клавиш Ctrl+D, чтобы выйти.

Задача 1. Количество слов

Пришло время для нашей первой задачи! На Python напишем небольшую программу для подсчета слов. Вы узнаете, как считывать входные данные пользователя, обрабатывать их для решения задачи и выводить результат. А также узнаете, как управлять текстом и числами в своих программах, использовать встроенные операции Python и сохранять промежуточные результаты на пути к решению.

Это задача с сайта DMOJ, код `dmopc15c7p2`.

Постановка задачи

Подсчитайте количество слов во входных данных. В этой задаче словом будет считаться любая последовательность строчных букв. Например, «привет» — это слово, но и всякие бессвязные наборы букв вроде «ыфафыва» тоже считаются словами.

Входные данные

Входными данными будет одна строка текста, состоящая из строчных букв и пробелов. Между каждой парой слов будет ровно один пробел, при этом перед первым словом и после последнего слова их не будет.

Максимальная длина строки — 80 символов.

Выходные данные

Количество слов в строке.

Строки

Фундаментальным строительным блоком программ на Python являются *значения*. У каждого значения есть *тип*, определяющий операции, которые над ним можно проводить. В задаче подсчета слов мы работаем со строкой текста. В Python текст хранится как строковое значение, поэтому нам нужно побольше узнать о том, что это значит. Чтобы решить задачу, требуется узнать количество слов в тексте, а для обозначения количества следует узнать о числовых значениях. Начнем со строк.

Представление строк

Строка — это тип в Python, который используется для работы с текстом. Чтобы записать строковое значение, нужно поместить его в одинарные кавычки. В оболочке Python введите следующее:

```
>>> 'hello'
'hello'
>>> 'a bunch of words'
'a bunch of words'
```

Оболочка Python повторяет за мной все, что я ввел.

А что, если в строке будет одинарная кавычка как ее часть?

```
>>> 'don't say that'
  File "<stdin>", line 1
    'don't say that'
         ^
SyntaxError: invalid syntax
```

Одиночная кавычка в слове `don't` завершает строку. Остальная часть уже не войдет в строку, поэтому наш ввод вызывает синтаксическую ошибку. *Синтаксическая ошибка* означает, что мы нарушили правила Python и написали неправильный код Python.

Чтобы исправить положение, можем воспользоваться двойными кавычками, которые тоже подходят для разделения строк:

```
>>> "don't say that"
"don't say that"
```

Но если в рассматриваемой строке нет одинарных кавычек, то применять двойные кавычки без причины я не буду.

Строковые операторы

Именно строки подойдут для хранения текста, количество слов в котором мы хотим подсчитать. Чтобы считать слова или вообще что-либо делать со строками, придется научиться работать с ними.

Над строками можно выполнять множество операций. Для некоторых из них используются специальные символы, вставляемые между операндами. Например, оператор + служит для конкатенации строк:

```
>>> 'hello' + 'there'
'hellothere'
```

Ах да, нам же нужен пробел между словами. Добавим его в конец первой строки:

```
>>> 'hello ' + 'there'
'hello there'
```

А еще есть оператор *, который размножает строку на указанное количество раз:

```
>>> '-' * 30
'------------------------------'
```

Здесь `30` — это целое число. О числах подробнее поговорим в ближайшее время.

ПРОВЕРИМ ЗНАНИЯ

Что выведет следующий код?

```
>>> '' * 3
```

А. `''''''`

Б. `''`

В. Этот код вызывает синтаксическую ошибку (недопустимый код Python).

Ответ: **Б.** `''` — это пустая строка — строка без символов. Трехкратное повторение пустой строки остается пустой строкой!

Строковые методы

Метод — это операция, выполняемая над определенным типом значений. У строк, в частности, много методов. Например, есть метод с именем `upper`, который превращает все буквы строки в прописные:

```
>>> 'hello'.upper()
'HELLO'
```

Информация, которую мы получаем от метода, называется *возвращаемым значением метода*. Например, в предыдущем примере мы могли бы сказать, что метод `upper` вернул строку `'HELLO'`.

Выполнение метода над значением называется *вызовом* метода. Вызывается он с помощью *точечной нотации* (`.`) между значением и именем метода. Также нужно после имени метода указать круглые скобки. У некоторых методов, например у `upper`, эти скобки остаются пустыми.

А для некоторых методов можно передать в скобках информацию. Остальные методы требуют информации и без нее просто не работают. Информация, которую мы включаем при вызове метода, называется его *аргументами*.

Например, у строк есть метод `strip`. Если вызывать его без аргументов, он удаляет из строки все начальные и конечные пробелы:

```
>>> ' abc'.strip()
'abc'
>>> '   abc       '.strip()
'abc'
>>> 'abc'.strip()
'abc'
```

Но мы также можем передать ему в качестве аргумента строку. Если сделать это, аргумент скажет методу, какие именно символы нужно удалить слева и справа:

```
>>> 'abc'.strip('a')
'bc'
>>> 'abca'.strip('a')
'bc'
>>> 'abca'.strip('ac')
'b'
```

Поговорим еще об одном строковом методе — `count`. Мы передаем ему строковый аргумент, и он сообщает, сколько вхождений этого аргумента найдено в строке:

```
>>> 'abc'.count('a')
1
>>> 'abc'.count('q')
0
>>> 'aaabcaa'.count('a')
5
>>> 'aaabcaa'.count('ab')
1
```

Если вхождения строки-аргумента перекрываются, учитывается только первое:

```
>>> 'ababa'.count('aba')
1
```

В отличие от прочих методов, которые я описал, метод `count` пригодится для поставленной задачи с подсчетом слов.

Сами подумайте: строка состоит из нескольких слов. Обратите внимание на то, что после каждого слова стоит пробел. Фактически, если бы вам нужно было подсчитать количество слов вручную, пробелы могли бы подсказать, где заканчивается каждое слово. А что, если мы посчитаем количество пробелов в строке? Для этого можем передать методу `count` символ пробела. Это выглядит так:

```
>>> 'this is a string with a few words'.count(' ')
7
```

Мы получили значение 7. Это не соответствует количеству слов, ведь их в строке восемь, но значение близко. Почему мы получили 7 вместо 8?

Причина в том, что пробел стоит после каждого слова, кроме последнего. Это значит, что при подсчете пробелов не учитывается последнее слово. Чтобы внести поправки, нам нужно научиться обращаться с числами.

Целые числа и числа с плавающей точкой

Любое *выражение* состоит из значений и операторов. Рассмотрим, как писать числовые значения и комбинировать их с операторами.

В Python есть два разных типа для работы с числами: целые числа (без дробной части) и числа с плавающей точкой (с дробной частью).

Целочисленные значения пишутся без десятичной точки. Вот несколько примеров:

```
>>> 30
30
>>> 7
7
>>> 1000000
1000000
>>> -9
-9
```

Значение само по себе — простейший вид выражения.

Привычные нам математические операторы позволяют работать с целыми числами. Оператор `+` выполняет сложение, `-` — вычитание, `*` — умножение. С их помощью можно писать более сложные выражения.

```
>>> 8 + 10
18
>>> 8 - 10
-2
>>> 8 * 10
80
```

Обратите внимание на пробелы вокруг операторов. Хотя для Python выражения `8+10` и `8 + 10` одинаковы, написание с пробелами упрощает чтение.

В Python есть два оператора деления! Оператор `//` выполняет целочисленное деление, при котором остаток отбрасывается, а результат округляется вниз:

```
>>> 8 // 2
4
>>> 9 // 5
1
>>> -9 // 5
-2
```

Если хотите получить остаток от деления, используйте оператор деления по модулю, который обозначается символом %. Например, деление 8 на 2 выполняется без остатка:

```
>>> 8 % 2
0
```

При делении 8 на 3 в остатке 2:

```
>>> 8 % 3
2
```

Оператор /, в отличие от //, не округляет результат:

```
>>> 8 / 2
4.0
>>> 9 / 5
1.8
>>> -9 / 5
-1.8
```

И вот тут-то полученные результаты уже не являются целыми числами! Они пишутся через дробную точку и принадлежат другому типу Python, называемому *float* (число с плавающей точкой). Чтобы писать значения с плавающей точкой, нужно добавить собственно точку:

```
>>> 12.5 * 2
25.0
```

Но пока вернемся к целым числам, а о числах с плавающей точкой поговорим в задаче про объем конуса позже в этой главе.

Когда мы используем в выражении несколько операторов, Python определяет порядок их применения согласно правилам приоритета. У каждого оператора есть приоритет. Ровно так же, как делали вы, решая примеры в школьных тетрадях, Python сперва выполняет операции умножения и деления (у них более высокий приоритет), а потом сложения и вычитания (более низкий приоритет):

```
>>> 50 + 10 * 2
70
```

Опять же, как и при вычислениях на бумаге, операции в круглых скобках имеют наивысший приоритет. Зная это, мы можем заставить Python выполнять операции в любом нужном нам порядке:

```
>>> (50 + 10) * 2
120
```

Программисты часто добавляют в выражения круглые скобки, даже если это особо и не требуется. Дело в том, что в Python довольно много операторов и в процессе написания легко запутаться в приоритетах, что чревато ошибками, а это плохо.

Вы можете спросить: а есть ли у целых чисел и чисел с плавающей точкой методы, как у строк. Есть! Но они не так уж и полезны. Например, есть метод, который сообщает нам, сколько памяти компьютера выделено на целое число. Чем оно больше, тем больше памяти ему требуется:

```
>>> (5).bit_length()
3
>>> (100).bit_length()
7
>>> (99999).bit_length()
17
```

В этом случае вокруг чисел нужны круглые скобки, иначе точечная нотация будет воспринята как десятичная точка и мы получим синтаксическую ошибку.

Переменные

Теперь вы умеете писать строковые и числовые значения. Неплохо бы еще получить возможность где-то хранить результаты вычислений, чтобы можно было воспользоваться ими позже. В задаче о подсчете слов было бы удобно где-нибудь хранить строку слов и подсчитанное количество слов в ней.

Оператор присваивания

Переменная — это имя, которое ссылается на некоторое значение. Всякий раз, когда мы позже используем имя переменной, оно будет заменяться значением, на которое ссылается эта переменная. Чтобы она ссылалась на значение, применяется *оператор присваивания*. Он состоит из переменной, знака равенства (`=`) и выражения. Python вычисляет выражение и заставляет переменную ссылаться на результат. Пример оператора присваивания:

```
>>> dollars = 250
```

Имя `dollars` всегда заменяется числом `250`:

```
>>> dollars
250
>>> dollars + 10
260
>>> dollars
250
```

Переменная одновременно ссылается только на одно значение. Как только мы укажем оператор присваивания и назначим ей другое значение, старое будет забыто:

```
>>> dollars = 250
>>> dollars
250
>>> dollars = 300
>>> dollars
300
```

В программе можно завести сколько угодно переменных. В больших программах обычно существуют сотни переменных. Пример использования двух переменных:

```
>>> purchase_price1 = 58
>>> purchase_price2 = 9
>>> purchase_price1 + purchase_price2
67
```

Обратите внимание: я выбрал такие имена переменных, чтобы из них было понятно, что в них хранится. Эти две переменные, например, хранят стоимости двух покупок. Было бы проще ввести имена переменных `p1` и `p2`, но, читая код через несколько дней, мы, вероятно, уже не поймем, что они означают!

Можно поместить в переменные строки:

```
>>> start = 'Monday'
>>> end = 'Friday'
>>> start
'Monday'
>>> end
'Friday'
```

Можно использовать их так же, как и переменные, которые ссылаются на числа, — в более крупных выражениях:

```
>>> start + '-' + end
'Monday-Friday'
```

Имена переменных Python должны начинаться со строчной буквы и могут содержать другие буквы, цифры и символы подчеркивания для разделения слов.

Изменение значений переменных

Предположим, у нас есть переменная `dollars`, которая ссылается на значение:

```
>>> dollars = 250
```

Теперь мы хотим увеличить значение так, чтобы переменная `dollars` стала равна `251`. Не сработает:

```
>>> dollars + 1
251
```

Результат равен `251`, но это значение пропало, нигде не сохранившись:

```
>>> dollars
250
```

Нам нужен оператор присваивания, который позволяет сохранить результат:

```
>>> dollars = dollars + 1
>>> dollars
251
>>> dollars = dollars + 1
>>> dollars
252
```

Учащиеся часто путают оператор присваивания `=` с символом равенства. Не повторяйте этой ошибки! Оператор присваивания — это команда, заставляющая переменную ссылаться на значение выражения, а не утверждение о том, что два значения равны.

ПРОВЕРИМ ЗНАНИЯ

Каким окажется значение переменной y после выполнения приведенного фрагмента кода?

```
>>> x = 37
>>> y = x + 2
>>> x = 20
```

А. `39`

Б. `22`

В. `35`

Г. `20`

Д. `18`

Ответ: **А.** Присваивание значения y выполняется лишь раз и дает значение `39`. Выражение `x = 20` меняет значение переменной x и на значение y никак не влияет.

Подсчет слов с использованием переменной

Давайте резюмируем все, что вы уже знаете о решении задачи подсчета слов.

- Вы узнали о строках и о том, что их можно применять для хранения какого-то текста.
- Узнали, что у строк есть метод `count`, который позволит подсчитать количество пробелов между словами в строке. Получим значение на единицу меньше нужного.
- Вы узнали о целых числах и об операторе сложения, который можно использовать для увеличения числа на 1.
- Узнали о переменных и об операторе присваивания, которые помогают сохранять значения.

Обобщив все это, можно сделать переменную ссылкой на строку и затем подсчитать количество слов:

```
>>> line = 'this is a string with a few words'
>>> total_words = line.count(' ') + 1
>>> total_words
8
```

Переменные `line` и `total_words` здесь не нужны, можно обойтись без них:

```
>>> 'this is a string with a few words'.count(' ') + 1
8
```

Применение переменных для сбора промежуточных результатов — хорошая практика для сохранения читабельности кода. Как только наши программы станут длиннее нескольких строк, без переменных будет уже не обойтись.

Чтение ввода от пользователя

Одна из проблем с написанным ранее кодом заключается в том, что он работает только с конкретной строкой, которую мы ввели. Он сообщает, что в этой строке столько-то слов, и на этом все. Чтобы узнать, сколько слов в другой строке, нам придется заменить текущую строку на новую. Если мы хотим решить задачу о подсчете слов, нужна программа, которая сможет работать с *любой* строкой, переданной ей в качестве входных данных.

Чтобы считать строку ввода, нужна функция `input`. *Функции* похожи на методы: мы вызываем их, передав какие-то аргументы, а они возвращают значение. Различие между методом и функцией заключается в том, что в функциях не используется точечная нотация. Вся информация передается в них через аргументы.

Далее приведен пример вызова функции ввода с последующим вводом информации, в данном случае слова `testing`:

```
>>> input()
testing
'testing'
```

Введя функцию `input()` и нажав клавишу Enter, вы не получите приглашение для ввода. Вместо этого Python ждет, когда вы что-нибудь напечатаете на клавиатуре и нажмете клавишу Enter. Затем функция `input` возвратит введенную вами строку. Как обычно, если мы нигде не сохраним эту строку, она будет потеряна. Воспользуемся оператором присваивания, чтобы сохранить введенное значение:

```
>>> result = input()
testing
>>> result
'testing'
>>> result.upper()
'TESTING'
```

Обратите внимание, что в последней строке я использовал метод `upper` на значении, возвращаемом `input`. Так делать можно, потому что `input` возвращает строку, а `upper` — строковый метод.

Вывод результата

Вы уже успели убедиться в том, что ввод выражений в оболочке Python приводит к выводу результата:

```
>>> 'abc'
'abc'
>>> 'abc'.upper()
'ABC'
>>> 45 + 9
54
```

Это подарок от встроенной оболочки Python. Предполагается, что если вы набираете выражение, то, вероятно, захотите увидеть его значение. Но при запуске программы Python вне оболочки Python этого удобства нет. Вместо этого мы должны явно вызывать функцию `print` всякий раз, когда хотим вывести что-нибудь на экран. Функция `print` также работает из оболочки:

```
>>> print('abc')
abc
>>> print('abc'.upper())
ABC
>>> print(45 + 9)
54
```

Обратите внимание на то, что строки, выводимые с помощью функции `print`, не заключаются в кавычки. Это удобно, так как мы, скорее всего, не захотим задействовать кавычки при общении программы с пользователями!

Приятная особенность функции `print` заключается в том, что вы можете передать ей сколько угодно аргументов и все они будут выведены и разделены пробелами:

```
>>> print('abc', 45 + 9)
abc 54
```

Решение задачи: полная программа на Python

Теперь мы готовы решить задачу подсчета слов и написать полноценную программу на Python. Выйдите из оболочки Python и вернитесь в командную строку операционной системы.

Запуск текстового редактора

Откройте текстовый редактор, в котором будете писать код. Следуйте инструкциям для вашей ОС.

Windows

В Windows мы будем использовать Блокнот — простейший текстовый редактор. В командной строке операционной системы перейдите в папку `programming`, если вы еще не в ней. Затем введите команду `notepad word_count.py` и нажмите Enter. Поскольку файла `word_count.py` не существует, Блокнот выведет запрос, хотите ли вы создать новый файл `word_count.py`. Нажмите кнопку Yes (Да), после чего можно будет начать писать программу на Python.

macOS

В macOS вы можете работать в любом текстовом редакторе, который вам нравится. Скорее всего, у вас уже установлен TextEdit. С помощью команд операционной системы перейдите в папку `programming`, если вы еще не там. Затем введите следующие две команды, нажимая клавишу Enter после каждой:

```
$ touch word_count.py
$ open -a TextEdit word_count.py
```

Команда `touch` создает пустой файл, чтобы текстовый редактор мог его открыть. Теперь можно писать программу на Python.

Linux

В Linux вы можете использовать любой текстовый редактор, который вам нравится. Скорее всего, у вас уже установлен редактор gedit. В командной строке операционной системы перейдите в папку `programming`, если вы еще не в ней. Затем введите команду `gedit word_count.py` и нажмите клавишу Enter. Теперь вы готовы писать программу на Python.

Программа

Запустив текстовый редактор, вы можете начать писать код программы на Python. Он показан в листинге 1.1.

Листинг 1.1. Программа подсчета слов

```
❶ line = input()
❷ total_words = line.count(' ') + 1
❸ print(total_words)
```

Цифры ❶, ❷ и ❸ вводить не надо. Они приведены, чтобы мы могли проанализировать код, и не являются его частью.

Сперва мы получаем от пользователя строку текста и присваиваем ее переменной ❶. Получаем строку, для которой можем применить метод `count`. Затем прибавляем 1 к количеству пробелов, чтобы учесть последнее слово в строке, а с помощью переменной `total_words` ссылаемся на этот результат ❷. Последнее, что нужно сделать, — вывести значение, на которое ссылается переменная `total_words` ❸.

Обязательно сохраните файл, когда закончите вводить код.

Запуск программы

Чтобы запустить программу, воспользуемся командой `python` в командной строке операционной системы. Вы уже знаете, что ввод отдельно взятой команды `python` запускает оболочку Python, но сейчас нам это не нужно. Мы хотим указать Python запустить программу в `word_count.py`. Для этого перейдите в папку программирования и введите `python word_count.py`. Здесь и на протяжении всей книги при необходимости применяйте команду `python3` вместо `python`.

Теперь программа ожидает, чтобы вы что-то напечатали. Введя несколько слов и нажав клавишу Enter, вы должны увидеть программу в деле. Например, наберите следующее:

```
this is my first python program
```

Программа должна выдать ответ 6.

Если вместо этого Python выдает ошибку, вернитесь к коду и убедитесь, что все ввели правильно. Python требует точности. Даже отсутствующая кавычка или скобка приведет к ошибкам.

Не расстраивайтесь, если запустить эту программу удастся не сразу. Для запуска первой программы может потребоваться много работы. Мы должны иметь возможность вводить программу в файл, вызывать Python для ее запуска и исправлять любые ошибки, возникающие из-за того, что что-то сделали неправильно. Но процедура запуска программ не меняется независимо от того, насколько они сложны, поэтому время, которое вы сейчас потратите, окупится во время работы над остальной частью книги.

Отправка на сайт

Поздравляю! Надеюсь, вам было приятно запустить на компьютере свою первую программу Python. Но как мы узнаем, что все работает правильно? Верный ли она выдает результат для всех возможных строк? Мы можем протестировать ее вручную еще на нескольких строках, но лучше всего отправить код онлайн-судье. Он автоматически запустит на нашем коде несколько тестов и сообщит, правильно ли они пройдены.

Перейдите на сайт https://dmoj.ca/ и авторизуйтесь на нем. (Если у вас нет учетной записи DMOJ, создайте ее, следуя инструкциям, данным во введении.) Щелкните на вкладке Problems и найдите код задачи с подсчетом слов `dmopc15c7p2`. Щелкните на выдаче поиска, чтобы загрузить задачу — она называется Not a Wall of Text.

Вы увидите текст задачи в том виде, как ее написал автор. Нажмите кнопку Submit Solution и вставьте свой код в текстовую область. Обязательно выберите язык программирования Python 3. Наконец, нажмите на кнопку Submit.

DMOJ запустит на коде тесты и покажет результаты. Для каждого теста будет выведен код состояния. Код *AC* означает *«принято»*, и именно такой результат нам нужен. Еще есть код *WA* (неправильный ответ) и *TLE* (превышен лимит времени). Если вы видите один из них, дважды проверьте вставленный код и убедитесь, что он точно соответствует коду из вашего текстового редактора.

Если все тесты будут пройдены, коду будет выставлена оценка 100/100 и мы получим за работу 3 балла.

Решая следующие задачи, мы станем придерживаться подхода, который использовали для задачи подсчета слов. Во-первых, будем исследовать функционал

оболочки Python, изучая новые функции Python по мере необходимости. Затем напишем программу, решающую задачу. После этого протестируем ее на собственном компьютере на своих тестовых примерах. И наконец, отправим код на сайт. Если какие-либо тестовые примеры не пройдут, будем пытаться устранить проблему.

Задача 2. Объем конуса

В предыдущей задаче нам нужно было прочитать строку из пользовательского ввода. Здесь же потребуется читать из ввода целые числа. Для этого нужен дополнительный шаг, а именно — извлечь целое число из строки. Попутно мы узнаем чуть больше о том, как в Python решаются математические задачи.

Задача с сайта DMOJ, код `dmopc14c5p1`.

Постановка задачи

Вычислите объем правильного конуса.

Входные данные

Входные данные состоят из двух строк текста. В первой вводится целое число r — радиус конуса. Во второй вводится целое число h — высота конуса. Значения r и h находятся в диапазоне от 1 до 100, то есть их минимальные значения равны 1, а максимальные — 100.

Выходные данные

Выводится объем правильного конуса с радиусом r и высотой h. Формула для расчета объема: $(\pi r^2 h) / 3$.

Усложняем математику

Пусть у нас есть переменные r и h, задающие радиус и высоту конуса соответственно:

```
>>> r = 4
>>> h = 6
```

Теперь вычислим объем по формуле $(\pi r^2 h) / 3$. Подставив в нее радиус 4 и высоту 6, получим $(\pi \cdot 4^2 \cdot 6) / 3$. Подставив значение числа $\pi = 3{,}14159$, получим 100,531. А как сделать это в Python?

Доступ к PI

Чтобы работать с числом π, нам придется завести подходящую переменную. Далее приведено достаточно точное значение числа `PI`:

```
PI = 3.141592653589793
```

Это скорее *константа*, чем переменная, поскольку изменять ее значение в коде мы не будем. По соглашению об именах в Python константы пишутся большими буквами.

Степень

В формуле $(\pi r^2 h) / 3$ единственным неясным моментом является возведение в квадрат. Поскольку $r^2 = r \cdot r$, мы можем использовать умножение, а не возведение в степень:

```
>>> r
4
>>> r * r
16
```

Но лучше применить возведение в степень напрямую. Всегда лучше писать максимально понятный код. Кроме того, в один прекрасный день вам, возможно, придется вычислять более крупные степени, а десять умножений громоздить неудобно. Оператор возведения в степень в Python — `**`:

```
>>> r ** 2
16
```

Итого, полная формула:

```
>>> (PI * r ** 2 * h) / 3
100.53096491487338
```

Отлично — это близко к ожидаемому нами результату 100,531!

Обратите внимание на то, что здесь мы нашли число с плавающей точкой. Как уже говорилось в этой главе, результат применения оператора `/` — число с плавающей точкой.

Преобразование между строками и целыми числами

В конечном итоге нам нужно будет считать из входных данных радиус и высоту, а затем использовать эти значения для расчета объема. Давайте попробуем:

```
>>> r = input()
4
>>> h = input()
6
```

Функция `input` всегда возвращает строку, даже если пользователь вводит целое число:

```
>>> r
'4'
>>> h
'6'
```

Наличие кавычек как раз и говорит о том, что эти значения являются строками. Для математических вычислений использовать строки нельзя. Если попробуем, получим ошибку:

```
>>> (PI * r ** 2 * h) / 3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Когда мы применяем значения неправильного типа, генерируется ошибка `TypeError`. В частности, Python не поддерживает использование оператора `**` между переменной-строкой `r` и числом `2`. Оператор `**` чисто математический и со строками не работает.

Чтобы преобразовать строки в целые числа, можно взять функцию Python `int`:

```
>>> r
'4'
>>> h
'6'
>>> r = int(r)
>>> h = int(h)
>>> r
4
>>> h
6
```

Теперь снова можем использовать эти значения в нашей формуле:

```
>>> (PI * r ** 2 * h) / 3
100.53096491487338
```

Выходит, если у вас есть строка, состоящая из цифр, можно применить функцию `int`, чтобы преобразовать ее в целочисленное значение. Функция действует при наличии в строке начальных или конечных пробелов, а вот с отличными от цифр символами уже не работает:

```
>>> int(' 12 ')
12
>>> int('12x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12x'
```

Чтобы преобразовать возвращаемую функцией `input` строку в целое число, мы можем сначала присвоить переменной возвращаемое `input` значение, а затем преобразовать его в целое число:

```
>>> num = input()
82
>>> num = int(num)
>>> num
82
```

или сделать и то и другое сразу:

```
>>> num = int(input())
82
>>> num
82
```

Здесь аргумент, передаваемый функции `int`, — это строка, возвращаемая функцией `input`. Функция `int` принимает эту строку и возвращает ее как целое число.

Если же нам когда-нибудь потребуется превратить целое число в строку, это можно будет сделать с помощью функции `str`:

```
>>> num = 82
>>> 'my number is ' + num
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> str(num)
'82'
>>> 'my number is ' + str(num)
'my number is 82'
```

Объединить строку и целое число нельзя. Функция `str` возвращает `'82'` вместо `82`, и это число можно использовать в конкатенации строк.

Решение задачи

Теперь мы готовы определить объем конуса. Создайте текстовый файл `cone_volume.py` и введите в него код из листинга 1.2.

Листинг 1.2. Вычисление объема конуса

```
❶ PI = 3.141592653589793

❷ radius = int(input())
❸ height = int(input())

❹ volume = (PI * radius ** 2 * height) / 3

❺ print(volume)
```

Чтобы разделить его на логические части, я добавил в код пустые строки. Python игнорирует их, но нам они облегчают чтение и восприятие кода.

Обратите внимание на то, что я использовал описательные имена переменных: `radius` вместо `r`, `height` вместо `h` и `volume`. Однобуквенные имена переменных годятся для математических формул, но в остальном коде нужно применять имена переменных, несущие больше информации.

Первым делом задаем переменной `PI` приближенное значение числа π ❶. Затем считываем радиус ❷ и высоту ❸ из ввода пользователя и преобразуем оба значения из строк в целые числа. С помощью формулы объема конуса вычислим эту величину ❹. Наконец, выводим результат ❺.

Сохраните файл `cone_volume.py`.

Запустите свою программу с помощью команды `python cone_volume.py`, а затем введите значения радиуса и высоты. Можно с помощью калькулятора убедиться, что все работает правильно!

Что произойдет, если вместо радиуса или высоты мы введем что-нибудь невнятное? Проверим. Запустите программу и введите следующее:

```
xyz
```

Вы должны увидеть ошибку:

```
Traceback (most recent call last):
  File "cone_volume.py", line 3, in <module>
    radius = int(input())
ValueError: invalid literal for int() with base 10: 'xyz'
```

Не слишком удобно. Но в рамках обучения программированию об этом можно не думать. Все тестовые примеры на сайте будут соответствовать заданному формату входных данных, поэтому с подобными проблемами вы не столкнетесь.

И да, DMOJ задолжал нам 3 балла, потому что мы закончили писать правильный код для этой задачи. Можете отправлять работу!

Резюме

Лед тронулся! Вы решили первые две задачи, написав для них код на Python. Изучили основные понятия программирования: значения, типы, строки, целые числа, методы, переменные, оператор присваивания, ввод и вывод.

Когда усвоите этот материал (возможно, с помощью приведенных далее упражнений), переходите к главе 2. В ней вы узнаете, как научить программы принимать решения. Больше не будем писать программы, которые идут просто сверху вниз. Они станут более гибкими и начнут решать задачи по-разному в зависимости от обстоятельств.

Упражнения

В конце каждой главы будут приведены упражнения, которые вы можете попробовать проработать. Я призываю вас выполнить как можно больше заданий.

Некоторые из этих упражнений могут занять немало времени. Делая одни и те же ошибки, вы можете разочароваться в Python. Но, как и в случае с любым другим навыком, который вы когда-либо осваивали, здесь необходима целенаправленная практика. В начале работы над упражнением я рекомендую решить несколько примеров вручную. Тогда вы будете знать, в чем состоит задача и что должна делать ваша программа. Можно писать код и без плана, но тогда придется одновременно собирать в кучу мысли и думать о написании кода.

Если полученный код не работает, задайте себе вопрос: а как именно он должен работать? Какие его строки виноваты в том, что появляются ошибки? Есть ли другой, возможно, более простой подход, который вы могли бы опробовать?

Решения упражнений вы найдете на сайте книги (https://nostarch.com/learn-code-solving-problems/). Но лучше не пользоваться ими, если только после нескольких неудачных попыток вы окончательно не отчаялись выполнить упражнение самостоятельно. Если посмотрите на решение и узнаете, как решить поставленную задачу, устройте перерыв и попробуйте сделать это самостоятельно с нуля. Зачастую

существует несколько способов решения. Если реализованное вами работает правильно, но отличается от моего, это не значит, что кто-то из нас ошибается. Напротив, для вас это хорошая возможность сравнить свой код с моим и узнать альтернативные методы решения.

1. DMOJ, задача A Spooky Season с кодом `wc16c1j1`.
2. DMOJ, задача A New Hope с кодом `wc15c2j1`.
3. DMOJ, задача Next in Line с кодом `ccc13j1`.
4. DMOJ, задача How's the Weather? с кодом `wc17c1j2` (тут повнимательнее с направлением конвертации!).
5. DMOJ, задача An Honest Day's Work с кодом `wc18c3j1`. (Подсказка: как определить количество крышек от бутылок и общее количество краски, необходимое для них?)

Примечания

Задача подсчета количества слов взята из сборника DMOPC (DMOJ Monthly Open Programming Competition) от 15 апреля 2016 года. Задача вычисления объема конуса взята из DMOPC от 14 марта 2015 года.

2

Принятие решений

Большинство программ, которыми мы пользуемся в обычной жизни, ведут себя по-разному в зависимости от тех или иных обстоятельств, возникающих во время их выполнения. Например, когда текстовый редактор спрашивает нас, хотим ли мы сохранить выполненную работу, он принимает решение в зависимости от нашего ответа: сохраняет работу, если мы отвечаем «да», и не сохраняет, если отвечаем «нет». В этой главе поговорим об операторе `if`, который позволяет программам принимать решения.

Мы попробуем решить две задачи: определить результат баскетбольного матча и выяснить, принадлежит ли номер телефона телемаркетологу.

Задача 3. Команда-победитель

Нам потребуется вывести сообщение, содержание которого будет зависеть от результата баскетбольного матча. Для этого нужен оператор `if`. Вы также узнаете, как можно хранить в программе истинные и ложные значения и манипулировать ими.

Задача с сайта DMOJ, код `ccc19j1`.

Постановка задачи

В баскетболе очки даются за трехочковый, двухочковый и штрафной бросок с одним очком.

Мы посмотрели баскетбольный матч между командами «Яблоки» и «Бананы» и записали количество успешных трех-, двух- и одноочковых бросков каждой команды. Нужно определить, какая команда выиграла или же игра закончилась ничьей.

Входные данные

Вводятся шесть строк. Первые три — очки «Яблок», последние три — очки «Бананов».

- В первой строке указано количество успешных трехочковых бросков команды «Яблок».
- Во второй строке указано количество успешных двухочковых бросков команды «Яблок».
- В третьей строке указано количество успешных штрафных бросков команды «Яблок».
- В четвертой строке указано количество успешных трехочковых бросков команды «Бананы».
- В пятой строке указано количество успешных двухочковых бросков команды «Бананы».
- В шестой строке указано количество успешных штрафных бросков команды «Бананы».

Все эти строки — целые числа от 0 до 100.

Выходные данные

На выходе будет один символ.

- Если «Яблоки» набрали больше очков, чем «Бананы», выведите `A` (A — Apples, то есть «Яблоки»).
- Если «Бананы» набрали больше очков, чем «Яблоки», выведите `B` (B — Bananas, то есть «Бананы»).
- Если команды набрали одинаковое количество очков, выведите букву `T` (tie — «ничья»).

Условное выполнение

Мы значительно упростим себе жизнь, если воспользуемся знаниями, которые получили в главе 1. С помощью функций `input` и `int` считаем каждое из шести целых чисел. Создадим переменные, чтобы сохранить эти значения. Мы можем умножить количество успешных трехочковых бросков на 3, а количество успешных двухочковых — на 2. С помощью функции `print` выведем нужную букву.

Вы пока не знаете, как научить программу принимать решение об исходе игры. Покажу, зачем это вообще нужно, с помощью двух примеров.

Сначала рассмотрим тестовый пример:

```
5
1
3
1
1
1
```

«Яблоки» набрали 5 · 3 + 1 · 2 + 3 = 20 очков, а «Бананы» — 1 · 3 +1 · 2 + 1 = 6 очков. «Яблоки» выиграли, поэтому выведем букву

```
A
```

Во втором примере баллы команд поменяем местами:

```
1
1
1
5
1
3
```

На этот раз выиграли «Бананы», так что нужно вывести другую букву:

```
B
```

Наша программа должна уметь сравнивать общее количество баллов, набранных «Яблоками» и «Бананами», и на основании полученного результата выводить `A`, `B` или `T`. Для принятия подобных решений можно использовать оператор `if`. *Условие* — это истинное или ложное выражение, а оператор `if` задействует условия, чтобы определить, что делать дальше. Операторы `if` так и называются — *операторы условного выполнения*, так как на выполнение нашей программы влияют различные условия.

Сначала давайте введем новый тип данных, который позволяет хранить истинные или ложные значения, и посмотрим, как создавать выражения этого типа. Именно их мы позднее будем передавать оператору `if`.

Логический тип

Если мы передадим любое выражение функции `type`, в ответ она сообщит вам тип переданного выражения:

```
>>> type(14)
<class 'int'>
```

```
>>> type(9.5)
<class 'float'>
>>> type('hello')
<class 'str'>
>>> type(12 + 15)
<class 'int'>
```

Один из типов Python, с которым мы еще не встречались, — это тип Boolean (`bool`). В отличие от целых чисел, строк и чисел с плавающей точкой, у которых миллиарды возможных значений, у этого типа есть лишь два логических значения: `True` и `False`. Именно с их помощью записываются условия:

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

Что с этими значениями можно делать? Например, для работы с числами у нас были математические операторы, такие как `+` и `-`, которые позволяли объединять значения в более сложные выражения. А для работы с логическими значениями понадобятся новые операторы.

Операторы сравнения

Больше ли число 5 числа 2? Меньше ли число 4, чем 1? Ответить на эти вопросы позволяют *операторы сравнения* Python. Они дают `True` или `False`, поэтому часто используются для написания *логических выражений*.

Оператор `>` принимает два операнда и возвращает `True`, если первый больше, чем второй, и `False` в противном случае:

```
>>> 5 > 2
True
>>> 9 > 10
False
```

Аналогично работает оператор `<`:

```
>>> 4 < 1
False
>>> -2 < 0
True
```

Операторы `>=` (больше или равно) и `<=` (меньше или равно):

```
>>> 4 >= 2
True
>>> 4 >= 4
True
>>> 4 >= 5
False
>>> 8 <= 6
False
```

Для определения равенства используется оператор `==`. Это именно два знака равенства, а не один (помните, что в качестве оператора присваивания применяется один знак равенства (`=`), но равенство он не проверяет):

```
>>> 5 == 5
True
>>> 15 == 10
False
```

Для определения неравенства используется оператор `!=`. Он возвращает `True`, если операнды не равны, и `False`, если равны:

```
>>> 5 != 5
False
>>> 15 != 10
True
```

Реальные программы не будут оценивать выражения, значения которых мы уже знаем. Нам и без Python известно, что 15 не равно 10. Как правило, в выражениях такого типа используются переменные. Например, результат выражения `number != 10` зависит от того, что находится в переменной `number`.

Операторы сравнения работают и со строками. При проверке равенства учитывается регистр:

```
>>> 'hello' == 'hello'
True
>>> 'Hello' == 'hello'
False
```

Одна строка меньше другой, если она идет первой по алфавиту:

```
>>> 'brave' < 'cave'
True
>>> 'cave' < 'cavern'
True
>>> 'orange' < 'apple'
False
```

А вот если есть и строчные, и прописные символы, то все интереснее:

```
>>> 'apple' < 'Banana'
False
```

Странно, да? Это связано с тем, как символы хранятся внутри компьютера. Как правило, символы верхнего регистра идут по алфавиту раньше символов нижнего регистра. А как насчет этого:

```
>>> '10' < '4'
True
```

Если бы это были числа, то результат был бы `False`. Но строки сравниваются посимвольно слева направо. Python сравнивает 1 и 4, и поскольку 1 меньше, оператор `<` возвращает `True`. Всегда следите за тем, чтобы значения были именно тех типов, которых вы ожидаете!

Есть оператор сравнения, который работает со строками, но не с числами, — оператор `in`. Он возвращает `True`, если первая строка встречается хотя бы один раз во второй, и `False` в противном случае:

```
>>> 'ppl' in 'apple'
True
>>> 'ale' in 'apple'
False
```

ПРОВЕРИМ ЗНАНИЯ

Какой результат даст следующий код?

```
a = 3
b = (a != 3)
print(b)
```

А. `True`

Б. `False`

В. `3`

Г. Этот код вызывает синтаксическую ошибку.

Ответ: **Б.** Выражение `a != 3` дает `False`, и оно же выводится.

Оператор if

Теперь рассмотрим несколько вариантов оператора `if` в Python.

Одиночный оператор if

Предположим, у нас есть окончательные результаты в двух переменных, `apple_total` и `banana_total`, и мы хотим вывести `A`, если значение переменной `apple_total` больше, чем `banana_total`. Это делается вот так:

```
>>> apple_total = 20
>>> banana_total = 6
>>> if apple_total > banana_total:
...     print('A')
...
A
```

Python выводит `A`, как и следовало ожидать.

Оператор `if` начинается с ключевого слова `if`. *Ключевое слово* — это слово, которое имеет особое значение для Python и не может использоваться в качестве имени переменной. После слова `if` идет логическое выражение, за которым следует двоеточие, а затем один или несколько операторов с отступом. Операторы с отступом часто называют *блоком* оператора `if`. Блок выполняется, если логическое значение возвращает `True`, и пропускается, если логическое выражение дает `False`.

Обратите внимание на то, что приглашение изменится с `>>>` на `...`. Это напоминание о том, что мы находимся внутри блока оператора `if` и должны делать в коде отступы. Я выбрал отступ в четыре пробела. Некоторые программисты на Python делают отступы клавишей Tab, но мы в этой книге будем использовать исключительно пробелы.

Как только вы введете `print('A')` и нажмете клавишу Enter, появится еще одно приглашение `...`. Поскольку нам нечего добавить в данный оператор `if`, нажмите клавишу Enter еще раз, чтобы закрыть это приглашение и вернуться к приглашению `>>>`. Дополнительное нажатие клавиши Enter — особенность оболочки Python. При написании программы на Python в файл пустые строки не требуются.

Рассмотрим пример помещения двух операторов в блок `if`:

```
>>> apple_total = 20
>>> banana_total = 6
```

```
>>> if apple_total > banana_total:
...     print('A')
...     print('Apples win!')
...
A
Apples win!
```

Оба вызова `print` выполняются, что дает две строки вывода.

Давайте попробуем другой оператор `if`, на этот раз с логическим выражением, которое возвращает `False`:

```
>>> apple_total = 6
>>> banana_total = 20
>>> if apple_total > banana_total:
...     print('A')
...
```

На этот раз функция `print` *не* вызывается: выражение `apple_total > banana_total` дает `False`, поэтому блок оператора `if` пропускается.

Составной оператор с elif

Давайте с помощью последовательных операторов `if` выведем букву `A`, если выиграли «Яблоки», `B`, если выиграли «Бананы», и `T`, если ничья:

```
>>> apple_total = 6
>>> banana_total = 6
>>> if apple_total > banana_total:
...     print('A')
...
>>> if banana_total > apple_total:
...     print('B')
...
>>> if apple_total == banana_total:
...     print('T')
...
T
```

Блоки первых двух операторов `if` пропускаются, потому что их логические выражения ложны. А блок третьего оператора `if` выполняется, выводя `T`.

Когда вы ставите одно выражение `if` за другим, они независимы и каждое логическое выражение оценивается независимо от того, были ли предыдущие выражения равны `True` или `False`.

При любых заданных значениях `apple_total` и `banana_total` может выполняться только один из операторов `if`. Например, если выражение `apple_total > banana_total` имеет значение `True`, то первый оператор `if` сработает, а два других — нет. Можно написать код, чтобы подчеркнуть, что разрешен запуск только одного блока кода. Вот как это сделать:

```
❶ >>> if apple_total > banana_total:
  ...     print('A')
❷ ... elif banana_total > apple_total:
  ...     print('B')
❸ ... elif apple_total == banana_total:
  ...     print('T')
  ...
  T
```

Теперь у нас один оператор `if`, а не три отдельных. Чтобы сделать это, не нажимайте клавишу Enter в приглашении, а вместо этого введите строку `elif`.

Чтобы выполнить этот оператор `if`, Python вычисляет первое логическое выражение ❶. Если оно дает `True`, то выводится `A`, а остальные `elif` пропускаются. Если оно дает `False`, Python вычисляет второе логическое выражение ❷. Если оно дает `True`, то выводится `B` и последний `elif` пропускается. Если оно дает `False`, Python вычисляет третье логическое выражение ❸. Если это `True`, то выводится `T`.

Ключевое слово `elif` означает else-if. С помощью этой расшифровки напоминайте себе о том, что выражение `elif` проверяется только в том случае, если перед ним ни одна из частей не сработала.

Эта версия кода эквивалентна предыдущему коду, где применялись три отдельных оператора `if`. Если бы мы хотели разрешить существование возможности выполнения более чем одного блока, нам пришлось бы использовать три отдельных `if`.

Оператор if с оператором else

Мы можем использовать ключевое слово `else` для запуска альтернативного кода, если все логические выражения в операторе `if` вернули `False`. Вот пример:

```
>>> if apple_total > banana_total:
...     print('A')
... elif banana_total > apple_total:
...     print('B')
... else:
...     print('T')
...
T
```

Python вычисляет логические выражения сверху вниз. Если любое из них имеет значение `True`, он запускает связанный блок и пропускает остальную часть оператора `if`. Если все логические выражения имеют значение `False`, Python выполняет блок `else`.

Обратите внимание, что мы убрали сравнение `apple_total == banana_total`. Мы попадаем в блок `else`, только если `apple_total > banana_total` дает `False` и `banana_total > apple_total` дает `False`, то есть если значения равны.

Что лучше использовать — отдельные операторы `if`? Оператор `if` с несколькими `elif`? Оператор `if` с `else`? Часто это вопрос вкуса. Задействуйте цепочку `elif`, если хотите выполнить не более одного блока кода. `Else` может помочь сделать код более понятным и избавить от необходимости писать обобщающее логическое выражение. Соблюдение правильной логики гораздо важнее точного оформления оператора `if`!

ПРОВЕРИМ ЗНАНИЯ

Какое значение переменной `x` даст следующий код?

```
x = 5
if x > 2:
    x = -3
if x > 1:
    x = 1
else:
    x = 3
```

А. `-3`

Б. `1`

В. `2`

Г. `3`

Д. `5`

Ответ: **Г.** Поскольку `x > 2` дает `True`, выполняется блок первого оператора `if`. Выполняется присвоение `x = -3`. Теперь о втором операторе `if`. Здесь `x > 1` дает `False`, поэтому работает блок `else` и значение переменной `x` становится равным `3`. Попробуйте поменять `if x > 1` на `elif x > 1` и посмотреть, что изменится в поведении программы!

ПРОВЕРИМ ЗНАНИЯ

Одинаково ли работают приведенные фрагменты кода? Предположим, что в переменной `temperature` уже есть число.

Фрагмент 1:

```
if temperature > 0:
    print('warm')
elif temperature == 0:
    print('zero')
else:
    print('cold')
```

Фрагмент 2:

```
if temperature > 0:
    print('warm')
elif temperature == 0:
    print('zero')
print('cold')
```

А. Да.

Б. Нет.

Ответ: **Б.** Фрагмент 2 всегда выводит слово `cold`, поскольку команда `Print('cold')` написана без отступа и не находится внутри операторов условия!

Решение задачи

Вернемся к задаче «Команда-победитель». В книге я обычно показываю полный код, а затем привожу пояснения к нему. Но, поскольку здесь решение длиннее, чем в главе 1, я решил показать код тремя частями и только потом привести его целиком.

Во-первых, нужно прочитать ввод от пользователя. Для этого требуется шесть запросов ввода, потому что у нас есть две команды и три нужных числа для каждой из них. Потребуется преобразовать все введенные строки в целое число. А вот и код:

```
apple_three = int(input())
apple_two = int(input())
apple_one = int(input())
```

```
banana_three = int(input())
banana_two = int(input())
banana_one = int(input())
```

Во-вторых, нужно определить количество очков, набранных командами. Для каждой команды мы складываем очки за трех-, двух- и одноочковые броски. Это можно сделать следующим образом:

```
apple_total = apple_three * 3 + apple_two * 2 + apple_one
banana_total = banana_three * 3 + banana_two * 2 + banana_one
```

В-третьих, выводим результат. Если выигрывают «Яблоки», мы выводим `A`, если выигрывают «Бананы», выводим `B`, в противном случае у нас ничья, поэтому выводим `T`. Мы используем для этого оператор `if`, как показано далее:

```
if apple_total > banana_total:
    print('A')
elif banana_total > apple_total:
    print('B')
else:
    print('T')
```

Это весь код, который нам нужен. Полный код приведен в листинге 2.1.

Листинг 2.1. Решение задачи «Команда-победитель»

```
apple_three = int(input())
apple_two = int(input())
apple_one = int(input())

banana_three = int(input())
banana_two = int(input())
banana_one = int(input())

apple_total = apple_three * 3 + apple_two * 2 + apple_one
banana_total = banana_three * 3 + banana_two * 2 + banana_one

if apple_total > banana_total:
    print('A')
elif banana_total > apple_total:
    print('B')
else:
    print('T')
```

Если вы отправите код на сайт, он, скорее всего, пройдет все тесты.

ПРОВЕРИМ ЗНАНИЯ

Правильно ли решает задачу приведенный далее код?

```
apple_three = int(input())
apple_two = int(input())
apple_one = int(input())

banana_three = int(input())
banana_two = int(input())
banana_one = int(input())

apple_total = apple_three * 3 + apple_two * 2 + apple_one
banana_total = banana_three * 3 + banana_two * 2 + banana_one

if apple_total < banana_total:
    print('B')
elif apple_total > banana_total:
    print('A')
else:
    print('T')
```

А. Да.

Б. Нет.

Ответ: **А**. Порядок операторов в коде иной, но работает все правильно. Если проигрывают «Яблоки», выводим B, ведь тогда «Бананы» победили. Ну и наоборот.

Прежде чем продолжить, можете попробовать решить упражнение 1 к этой главе (приведено в конце).

Задача 4. Телемаркетологи

Иногда приходится кодировать более сложные логические выражения, чем те, которые вы видели до сих пор. В этой задаче изучим логические операторы, которые помогут в этом.

Это задача с сайта DMOJ, код `ccc18j1`.

Постановка задачи

Мы предполагаем, что телефонные номера состоят из четырех цифр. Номер принадлежит телемаркетологу, если его четыре цифры удовлетворяют всем трем следующим требованиям:

- первая цифра 8 или 9;
- четвертая цифра 8 или 9;
- вторая и третья цифры совпадают.

Например, номер телефона 8119 принадлежит телемаркетологу.

Нам нужно определить, принадлежит ли номер телефона телемаркетологу, и указать, следует отвечать на телефонный звонок или игнорировать его.

Входные данные

Вводятся четыре строки. В них указаны первая, вторая, третья и четвертая цифры номера телефона. Каждая цифра представляет собой целое число от 0 до 9.

Выходные данные

Если номер телефона принадлежит телемаркетологу, вывести слово `ignore`, в противном случае — `answer`.

Логические операторы

Итак, что отличает телефон, принадлежащий телемаркетологу? Его первая цифра должна быть 8 *или* 9. Четвертая цифра должна быть 8 *или* 9. Вторая и третья цифры должны быть одинаковыми. Мы можем закодировать логику «и» и «или» с помощью *логических операторов* Python.

Оператор or

Оператор `or` принимает в качестве операндов два логических выражения. Он возвращает `True`, если хотя бы один операнд равен `True`, иначе возвращает `False`:

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

Единственный способ получить значение `False` из оператора `or` — это если оба его операнда равны `False`.

Мы можем использовать оператор `or`, чтобы узнать, равна ли цифра 8 или 9:

```
>>> digit = 8
>>> digit == 8 or digit == 9
True
>>> digit = 3
>>> digit == 8 or digit == 9
False
```

В главе 1 мы говорили, что для определения порядка применения операторов Python использует правила приоритета. Приоритет оператора `or` ниже, чем приоритет операторов сравнения, это означает, что круглые скобки вокруг операндов не нужны. Например, в выражении `digit == 8 or digit == 9` оба операнда равны цифре 8 или 9. Это то же самое, как если бы мы написали `(digit == 8) or (digit == 9)`.

На естественном языке это звучит так: «цифра равна 8 или 9». А в Python выглядит так:

```
>>> digit = 3
>>> if digit == 8 or 9:
...     print('yes!')
...
yes!
```

Обратите внимание, что я записал во втором операнде `9` вместо `digit == 9`, — это неправильно! Python в ответ выводит `yes`, но мы хотели не совсем этого, учитывая, что переменной `digit` присвоено значение `3`. Причина в том, что Python считает ненулевые числа равными `True`. Поскольку 9 считается `True`, все выражение становится равно `True`. Проверяйте свои логические выражения, чтобы избежать ошибок при переводе с естественного языка на Python.

Оператор and

Оператор `and` возвращает `True`, если оба его операнда равны `True`, иначе возвращает `False`:

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

Получить значение `True` из оператора `and` можно лишь в случае, если оба его операнда имеют значение `True`.

Приоритет операции and выше, чем у or. Покажем, почему это важно:

```
>>> True or True and False
True
```

Python интерпретирует это выражение следующим образом, причем сначала выполняется оператор and:

```
>>> True or (True and False)
True
```

В результате получаем True, потому что первый операнд or дает True.

Мы можем заставить оператор or выполняться первым, добавив круглые скобки:

```
>>> (True or True) and False
False
```

В результате получается False, потому что второй операнд равен False.

Оператор not

Введем еще один логический оператор — not. В отличие от or и and он принимает один операнд, а не два. Если его операнд равен True, то оператор not возвращает False, и наоборот:

```
>>> not True
False
>>> not False
True
```

Приоритет операции not выше, чем у операторов or и and.

ПРОВЕРИМ ЗНАНИЯ

Далее приведены выражения. Какое из них равняется True?

А. not True and False

Б. (not True) and False

В. not (True and False)

Г. Ни одно из перечисленных.

Ответ: **В.** Выражение (True and False) равно False, а оператор not превращает его в True.

ПРОВЕРИМ ЗНАНИЯ

Рассмотрим выражение `not a and b`. При каких значениях атрибутов оно равно `False`?

А. `a = False, b = False`

Б. `a = False, b = True`

В. `a = True, b = False`

Г. `a = True, b = True`

Д. Более одного из перечисленных.

Ответ: **В**. Если `a = True`, то `not a = False`. Если `b = False`, то оба операнда равны `False`, как и все выражение.

Решение задачи

Вооружившись логическими операторами, мы можем решить задачу о телемаркетологах. Решение приведено в листинге 2.2.

Листинг 2.2. Решение проблем с телемаркетингом

```
num1 = int(input())
num2 = int(input())
num3 = int(input())
num4 = int(input())

❶ if ((num1 == 8 or num1 == 9) and
        (num4 == 8 or num4 == 9) and
        (num2 == num3)):
    print('ignore')
else:
    print('answer')
```

Как и в задаче «Команда-победитель», мы сперва должны считать входные данные и преобразовать их в целые числа.

Заголовок оператора `if` ❶ состоит из трех выражений, связанных операторами `and`. Каждый из них должен быть истинным, чтобы все выражение также было истинно. Нужно, чтобы первое число было равно 8 или 9, четвертое число — равно 8 или 9, а второе и третье числа были равны между собой. Если все три условия

выполняются, то мы будем знать, что номер телефона принадлежит телемаркетологу, и выводим сообщение `ignore`. В противном случае номер телефона принадлежит кому-то еще, выводим `answer`.

Я разбил логическое выражение на три строки. Для этого необходимо заключить все выражение в дополнительную пару круглых скобок (без них вы получите синтаксическую ошибку, потому что Python не может просто так переносить строки в середине выражения).

Руководство по стилям Python предполагает, что длина строки не должна превышать 79 символов. Строка с полным логическим выражением будет состоять из 76 символов. Однако мне кажется, что трехстрочная версия более понятна, так как каждое условие в ней находится в отдельной строке.

Мы применили хорошее решение. Но чтобы углубиться в задачу, обсудим несколько альтернативных подходов.

В нашем коде используется логическое выражение, позволяющее определить, принадлежит ли номер телефона телемаркетологу. Можно было бы, напротив, написать код, который говорил бы нам, что номер телефона не принадлежит телемаркетологу. Тогда мы бы выводили `answer`, а в противном случае — `ignore`.

Если первая цифра не 8 и не 9, то номер телефона не принадлежит телемаркетологу. То же самое, если четвертая цифра не 8 и не 9. Или если вторая и третья цифры не равны. Если хотя бы одно из этих выражений истинно, то телефонный номер не принадлежит телемаркетологу.

В листинге 2.3 обрабатывается именно такая логика.

Листинг 2.3. Альтернативное решение задачи

```
num1 = int(input())
num2 = int(input())
num3 = int(input())
num4 = int(input())

if ((num1 != 8 and num1 != 9) or
        (num4 != 8 and num4 != 9) or
        (num2 != num3)):
    print('answer')
else:
    print('ignore')
```

В этих `!=`, `and` и `or` легко запутаться! Обратите внимание, например, на то, что нам пришлось заменить все операторы `==` на `!=`, а все `and` — на `or`.

Альтернативный подход заключается в использовании оператора `not` и отрицании выражения «это телемаркетолог» (листинг 2.4).

Листинг 2.4. Решение проблем с телемаркетингом, а не с оператором

```
num1 = int(input())
num2 = int(input())
num3 = int(input())
num4 = int(input())

if not ((num1 == 8 or num1 == 9) and
        (num4 == 8 or num4 == 9) and
        (num2 == num3)):
    print('answer')
else:
    print('ignore')
```

Какое из этих решений кажется вам более удачным? Часто существует несколько способов структурировать логику оператора `if` и использовать следует тот из них, который проще для понимания. Для меня листинг 2.2 самый простой!

Выберите понравившуюся версию и отправьте ее на сайт. Убедитесь, что все тесты пройдены успешно.

Комментарии

Всегда нужно стремиться к тому, чтобы создаваемый код был как можно более понятным. Это помогает избежать появления ошибок при программировании и упрощает отладку кода, когда ошибки все же появляются. Значимые имена переменных, пробелы вокруг операторов, пустые строки для разделения программы на логические части, простая логика оператора `if` — все эти методы могут улучшить качество кода, который мы пишем. Еще одна хорошая привычка — добавлять в код комментарии.

Комментарий начинается с символа `#` и продолжается до конца строки. Python игнорирует комментарии, поэтому они не влияют на работу программы. Комментарии нужны, чтобы напомнить себе или другим, что и зачем использовано в коде. Предположим, что человек, читающий наш код, знает Python, поэтому не стоит писать в комментариях очевидные вещи наподобие таких:

```
>>> x = 5
>>> x = x + 1 # Увеличение x на 1
```

Этот комментарий не несет новой информации, так как мы и без того знаем об операторах присваивания.

В листинге 2.5 приведен код листинга 2.2 с комментариями.

Листинг 2.5. Решение проблем с телемаркетингом, добавлены комментарии

```
❶ # ccc18j1, Telemarketers

  num1 = int(input())
  num2 = int(input())
  num3 = int(input())
  num4 = int(input())

❷ # Номер телемаркетолога: первое число — 8 или 9,
  # четвертое — 8 или 9, второе и третье числа идентичны
  if ((num1 == 8 or num1 == 9) and
      (num4 == 8 or num4 == 9) and
      (num2 == num3)):
    print('ignore')
  else:
    print('answer')
```

Я добавил три строки комментариев: в первой приведены код и название задачи ❶, а в комментариях перед оператором `if` описаны правила определения телефонного номера телемаркетолога ❷.

Не переусердствуйте с комментариями. По возможности пишите код, который их не требует. Но если он неочевиден или нужно обозначить, почему вы решили сделать что-то определенным образом, хорошо продуманный комментарий поможет вам сэкономить время и избежать проблем с пониманием в будущем.

Перенаправление ввода и вывода

Когда вы отправляете свой код Python на сайт, его тестер запускает множество тестовых примеров, чтобы определить, правильно ли он работает. Неужели там сидит человек и ждет, когда наша программа запросит ввод с клавиатуры?

Не-а! Все автоматизировано. Никто не набирает тестовые примеры на клавиатуре. А как тогда платформа проверяет наш код, если в нем что-то вводится с клавиатуры?

Дело в том, что входные данные не обязательно читать именно с клавиатуры. На самом деле выполняется чтение из источника ввода, называемого *стандартным вводом*, который по умолчанию настроен на клавиатуру.

Можно настроить стандартный ввод так, чтобы он ссылался на файл, а не на клавиатуру. Этот метод называется *перенаправлением ввода*, и платформа пользуется им для ввода данных.

Вы тоже можете попробовать настроить перенаправление ввода. В программах с небольшим объемом входных данных оно не особенно полезно, но в программах, тестовые примеры которых могут содержать десятки или сотни строк, значительно упрощает тестирование работы. Вместо того чтобы вводить один и тот же тестовый пример снова и снова, мы можем сохранить его в файле, а затем запускать программу сколько угодно раз.

Попробуем настроить перенаправление ввода на нашей задаче. Перейдите в папку `programming` и создайте файл с именем `telemarketers_input.txt`. Введите в нем следующее:

```
8
1
1
9
```

В постановке задачи сказано, что мы должны вводить числа по одному в каждой строке, поэтому мы так и написали.

Сохраните файл. Теперь введите команду `python telemarketers.py < telemarketers_input.txt`, чтобы запустить свою программу с перенаправлением ввода. Программа возьмет данные из файла, как если бы вы набирали тестовый пример с клавиатуры.

Символ `<` говорит операционной системе, что для ввода нужно использовать файл, а не клавиатуру. После символа `<` указывается имя файла с входными данными.

Чтобы проверить программу на разных тестовых примерах, просто измените файл `telemarketers_input.txt` и снова запустите свою программу.

Мы можем перенаправить и выходной поток, хотя в данной книге нам это не понадобится. Функция `print` отправляет данные на стандартный вывод, которым по умолчанию является экран. Мы можем изменить стандартный вывод так, чтобы он ссылался на файл. Для этого, как и в случае с вводом, нужно ввести символ `>` и имя файла.

Введите команду `python telemarketers.py > telemarketers_output.txt`, чтобы запустить свою программу с перенаправлением вывода. Введите четыре целых числа, после чего должно появиться приглашение операционной системы. Но результатов от программы вы не получите! Так произойдет, потому что вывод перенаправлен в файл `telemarketers_output.txt`. Если вы откроете файл `telemarketers_output.txt` в текстовом редакторе, результат будет там.

Будьте осторожны с перенаправлением вывода. Если ввести в команду уже существующее имя файла, старый файл будет перезаписан! Всегда проверяйте, правильно ли ввели имя файла.

Резюме

В этой главе вы узнали, как использовать операторы `if` для управления работой программы. Ключевым ингредиентом оператора `if` является логическое выражение, которое получает значение `True` или `False`. Для построения логических выражений применяются операторы сравнения `==` и `>=`, а также логические операторы `and` и `or`.

Принятие решений на основании истинных или ложных высказываний делает программы более гибкими и позволяет им адаптироваться к текущей ситуации. Однако наши программы по-прежнему ограничены обработкой небольших объемов ввода и вывода, которые мы организуем с помощь операторов `input` и `print`. В следующей главе начнем изучать циклы, которые позволяют повторять код, чтобы можно было обрабатывать столько ввода и вывода, сколько захочется.

Хотите поработать со 100 значениями? Как насчет 1000? И с небольшим количеством кода Python? Да, я помню, что вам еще нужно выполнить следующие упражнения. Но когда будете готовы, читайте дальше!

Упражнения

Ниже приведено несколько упражнений, которые вы можете попробовать выполнить.

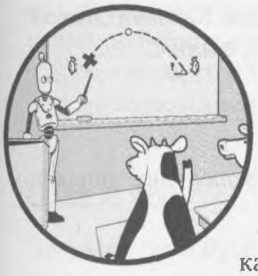
1. DMOJ, задача Canadian Calorie Counting с кодом `ccc06j1`.
2. DMOJ, задача Special Day с кодом `ccc15j1`.
3. DMOJ, задача Happy or Sad с кодом `ccc15j2`.
4. DMOJ, задача C.C. and Cheese-kun с кодом `dmopc16c1p0`.
5. DMOJ, задача Who is in the Middle с кодом `ccc07j1`.

Примечания

Задача «Команда-победитель» взята из канадского компьютерного конкурса 2019 года, младший уровень. Задача «Телемаркетологи» взята из того же конкурса, но 2018 года, младший уровень.

3

Повторяющийся код: определенные циклы



Компьютеры любят выполнять повторяющиеся действия. Они без устали выполняют поставленную задачу хоть десять, хоть сто, хоть миллиард раз. В этой главе мы поговорим о циклах — операторах, которые велят компьютеру выполнять какую-то часть кода многократно.

Мы будем использовать циклы для решения трех задач: отслеживания местоположения шарика под чашкой, подсчета количества занятых парковочных мест и получения информации о том, сколько трафика осталось на телефоне согласно тарифу.

Задача 5. Три чашки

Здесь мы будем отслеживать положение шарика под перемещающимися чашками. Делать это они могут много раз, поэтому писать код для каждого хода — не вариант. Вместо этого мы изучим и будем использовать цикл `for`, который позволяет упростить запуск кода каждого хода.

Задача с сайта DMOJ, код `coci06c5p1`.

Постановка задачи

У Борко есть три непрозрачные чашки: слева (позиция 1), посередине (позиция 2) и справа (позиция 3). Под левой чашкой спрятан шарик. Наша работа — отслеживать местонахождение шарика, когда Борко меняет чашки местами.

Он может выполнять три типа движений:

A — менять местами левую и среднюю чашки;

B — менять местами среднюю и правую чашки;

C — менять местами левую и правую чашки.

Например, если первым происходит перемещение A, то левая и средняя чашки меняются местами, а поскольку шарик находится слева, эта перестановка перемещает его в середину. Если же первое перемещение — B, то меняются местами средняя и правая чашки и шарик остается на месте.

Входные данные

Входные данные — одна строка, содержащая не более 50 символов. Каждый символ обозначает смену позиции чашек, которую выполняет Борко: `A`, `B` или `C`.

Выходные данные

Программа выводит окончательное местоположение шарика, обозначенное цифрой:

- `1`, если он слева;
- `2`, если он в середине;
- `3`, если он справа.

Зачем нужны циклы

Рассмотрим тестовый пример:

```
ACBA
```

Здесь выполняются четыре типа перемещений. Чтобы определить окончательное местоположение шарика, нужно выполнить их все.

Первое перемещение — это тип `A`, при котором меняются местами чашки слева и посередине. Поскольку шарик находится слева, это приводит к его смещению в середину. Второе перемещение — это тип `C`, меняются местами чашки слева и справа. Поскольку шарик в данный момент находится в центре, это не влияет на его расположение. Третье перемещение — `B`, меняются местами чашки посередине и справа. Это передвигает шарик с середины вправо. Четвертое перемещение — это тип `A`, меняются местами чашки слева и посередине. Это не влияет на шарик. Следовательно, ответ — `3`, потому что шарик оказался справа.

Обратите внимание на то, что во время каждого перемещения нужно определить, поменялась ли позиция шарика, и, если да, обозначить это соответствующим

образом. Из главы 2 мы уже знаем, как принимать такие решения. Например, если тип перемещения — A и шарик находится слева, то он переходит в середину. Это выглядит так:

```
if swap_type == 'A' and ball_location == 1:
ball_location = 2
```

Можно добавить `elif` для всех других случаев смены положения шарика: если выполняется перемещение A и шарик находится посередине, перемещение B и шарик находится посередине, перемещение B и шарик находится справа и т. д. Полученный большой `if` сможет правильно обработать одно перемещение. Но этого недостаточно для решения задачи, потому что в тестовом примере может быть до 50 случаев перемещения. Нужно будет повторять логику оператора `if` для каждого из них. И мы, конечно, не хотели бы копировать и вставлять один и тот же код 50 раз. Только представьте, что вы допустили опечатку и нужно исправить ее 50 раз. Или вам захотелось проверить программу на миллионе перемещений. Тогда имеющихся знаний нам уже не хватит. Требуется способ прокрутить все перемещения, выполняя одну и ту же логику многократно. Нужны циклы.

Цикл for

Оператор `for` в Python создает *циклы* `for`. Они позволяют обрабатывать все элементы последовательности. Единственный тип последовательности, который мы пока рассмотрели, — это строка. С другими типами последовательностей познакомимся по ходу дела, циклы `for` работают со всеми.

Вот первый пример цикла `for`:

```
>>> secret_word = 'olive'
>>> for char in secret_word:
...     print('Letter: ' + char)
...
Letter: o
Letter: l
Letter: i
Letter: v
Letter: e
```

После ключевого слова `for` пишется имя *переменной цикла*. Так называется переменная, которая в ходе выполнения цикла ссылается на разные значения. В цикле `for`, если мы передадим ему строку, переменная цикла перебирает символы строки.

Я выбрал имя переменной `char` (это означает «символ»), чтобы было понятнее, что в ней перебираются символы строки. Иногда код становится понятнее, если имена переменных каким-то образом характеризуют контекст. Например, в задаче «Три чашки» мы могли бы использовать имя `swap_type`, чтобы стало яснее, что в переменной хранится тип перемещения.

После имени переменной идет ключевое слово, а затем строка, которую мы хотим перебрать. Здесь мы перебираем строку `secret_word` со значением `olive`.

Как и строки `if`, `elif` и `else` в команде `if`, строка `for` заканчивается двоеточием (`:`). И как блок оператора `if`, блок оператора `for` пишется с отступом.

Выполнение блока операторов с отступом называется *итерацией цикла*. Рассмотрим по шагам, что делает цикл на каждой итерации.

- На первой итерации Python помещает в переменную `char` символ `'o'` — первый символ слова `'olive'`. Затем он запускает блок цикла, который состоит только из вывода буквы на экран. Поскольку `char` ссылается на значение `'o'`, результат будет таким: `Letter: o`.
- На второй итерации Python помещает в переменную `char` символ `'l'`, то есть вторую букву слова. Затем выводится `Letter: l`.
- Этот процесс повторяется еще три раза, перебирая все буквы в слове `'olive'`.
- Затем цикл завершается. Другого кода у нас нет, поэтому программа заканчивает работу. Если бы после цикла был еще код, он бы выполнился.

В блок цикла `for` можно помещать сразу несколько операторов. Пример:

```
>>> secret_word = 'olive'
>>> for char in secret_word:
...     print('Letter: ' + char)
...     print('*')
...
Letter: o
*
Letter: l
*
Letter: i
*
Letter: v
*
Letter: e
*
```

Теперь у нас есть два оператора, которые выполняются на каждой итерации цикла: один выводит текущую букву строки, а другой — символ *.

Цикл `for` перебирает элементы последовательности, и ее длина позволяет понять, сколько итераций будет выполнено. Функция `len` принимает строку и возвращает ее длину:

```
>>> len('olive')
5
```

Таким образом, наш цикл for на слове 'olive' будет состоять из пяти итераций:

```
>>> secret_word = 'olive'
>>> print(len(secret_word), 'iterations, coming right up!')
>>> for char in secret_word:
...     print('Letter: ' + char)
...
5 iterations, coming right up!
Letter: o
Letter: l
Letter: i
Letter: v
Letter: e
```

Я вызвал print с несколькими аргументами ❶ вместо использования конкатенации, чтобы избежать преобразования длины в строку.

Цикл for называется *определенным циклом*, так как количество итераций в нем предопределено. Существуют также *неопределенные циклы*, количество итераций в которых зависит от того, что происходит при запуске программы. Мы изучим их в следующей главе.

ПРОВЕРИМ ЗНАНИЯ

Какой результат даст следующий код?

```
s = 'garage'
total = 0

for char in s:
    total = total + s.count(char)

print(total)
```

А. 6

Б. 10

В. 12

Г. 36

Ответ: **Б.** Для каждого символа в слове 'garage' мы прибавляем количество его вхождений в переменную total. У нас по одной букве r и e, две буквы g и три буквы a.

Вложенные операторы

Блок цикла `for` состоит из одного или нескольких операторов. Это могут быть однострочные операторы, такие как вызовы функций и операторы присваивания. А могут быть и многострочные операторы, такие как `if` и циклы.

Начнем с примера оператора `if` внутри цикла `for`. Предположим, нужно вывести из строки только символы в верхнем регистре. У строк есть метод `isupper`, который позволяет определить, относится ли символ к верхнему регистру:

```
>>> 'q'.isupper()
False
>>> 'Q'.isupper()
True
```

Можно использовать метод `isupper()` в операторе `if`, чтобы задавать поведение в каждой итерации цикла `for`:

```
>>> title = 'The Escape'
>>> for char in title:
...     if char.isupper():
...         print(char)
...
T
E
```

Внимательно работайте с отступами. Нам нужен один уровень отступа для цикла `for` и еще один — для вложенного оператора `if`.

На первой итерации в переменной `char` хранится символ `'T'`. Поскольку `'T'` — это прописная буква, метод `isupper` возвращает `True` и выполняется блок оператора `if`. Выводится буква `T`. На второй итерации в `char` попадает буква `'h'`. На этот раз метод `isupper` возвращает `False`, поэтому блок оператора `if` не запускается. Цикл `for` перебирает строку целиком, но вложенный оператор `if` срабатывает только дважды: на букве `T` в начале строки и на `E` в начале слова `Escape`.

А можно ли вложить цикл `for` в цикл `for`? Можно! Вот пример:

```
>>> letters = 'ABC'
>>> digits = '123'
>>> for letter in letters:
...     for digit in digits:
...         print(letter + digit)
...
A1
A2
A3
B1
B2
```

```
B3
C1
C2
C3
```

Код производит все двухсимвольные строки, первый символ которых есть в `letters`, а второй — в `digits`

На первой итерации внешнего (буквенного) цикла в `char` попадает `A`. Здесь полностью выполняется внутренний (цифровой) цикл. На всем протяжении выполнения внутреннего цикла в переменной `char` находится `A`. На первой итерации внутреннего цикла `digit=1`, что объясняет вывод `A1`. На второй итерации внутреннего цикла `digit=2` и выводится `A2`. На третьей, последней итерации внутреннего цикла `digit=3` и выводится `A3`.

Но это еще не все! Мы прошли только одну итерацию внешнего цикла. На второй итерации внешнего цикла обрабатывается буква `B`. Снова выполняются три итерации внутреннего цикла, на этот раз с буквой `B`. Получаем `B1`, `B2` и `B3`. Наконец, на третьем этапе внешнего цикла `char='C'`, а внутренний цикл выводит `C1`, `C2` и `C3`.

ПРОВЕРИМ ЗНАНИЯ

Какой результат даст следующий код?

```
title = 'The Escape'
total = 0

for char1 in title:
    for char2 in title:
        total = total + 1

print(total)
```

А. `10`

Б. `20`

В. `100`

Г. Этот код вызывает синтаксическую ошибку, потому что два вложенных цикла не могут перебирать строку `title`.

Ответ: **В.** `Total` начинается с `0` и увеличивается на 1 на каждой итерации внутреннего цикла. Длина строки `'The Escape'` равна 10. Таким образом, внешний цикл отрабатывает 10 итераций. Для каждой из этих итераций внутренний цикл выполняет свои 10 итераций. Таким образом, внутренний цикл отрабатывает $10 \cdot 10 = 100$ итераций.

Решение задачи

Вернемся к трем чашкам. Организуем цикл `for` для перебора всех перемещений и вложенный оператор `if`, который следит, где находится шарик:

```
for swap_type in swaps:
    # Оператор if для отслеживания положения шарика
```

Существует три типа перемещения (`A`, `B` и `C`) и три возможных местоположения шарика, поэтому возникает соблазн сделать вывод, что мы должны написать оператор `if` с $3 \cdot 3 = 9$ логическими выражениями (одно после `if` и по одному после каждого из восьми `elif`). Но на самом деле нужно всего шесть логических выражений. Три из девяти возможных операций не передвигают шарик: перемещение `A`, когда шарик находится справа, перемещение `B`, когда шарик находится слева, и перемещение `C`, когда шарик посередине.

В листинге 3.1 приведено решение задачи.

Листинг 3.1. Решение задачи о трех чашках

```
swaps = input()

ball_location = 1

❶ for swap_type in swaps:
    ❷ if swap_type == 'A' and ball_location == 1:
        ❸ ball_location = 2
      elif swap_type == 'A' and ball_location == 2:
          ball_location = 1
      elif swap_type == 'B' and ball_location == 2:
          ball_location = 3
      elif swap_type == 'B' and ball_location == 3:
          ball_location = 2
      elif swap_type == 'C' and ball_location == 1:
          ball_location = 3
      elif swap_type == 'C' and ball_location == 3:
          ball_location = 1

print(ball_location)
```

С помощью функции `input` я присвоил строку перемещений переменной `swaps`. Цикл `for` ❶ проходит по всем перемещениям. Каждое из них обрабатывается вложенным оператором `if` ❷. Каждая ветвь `if` и `elif` определяет, что происходит с шариком при его текущем положении и данном типе перемещения, а затем передвигает шарик в нужную позицию. Например, если тип обмена — `A` и шарик находится в позиции `1` ❷, то он попадает в точку `2` ❸.

В этом коде имеет значение, используем ли мы несколько `elif` (один большой оператор `if`) или несколько `if` (несколько операторов `if`). Если мы перейдем

от `elif` к `if`, то код перестанет работать. В листинге 3.2 приведен неправильный код.

Листинг 3.2. Неправильное решение задачи

```
# Этот код неправильный

swaps = input()

ball_location = 1

for swap_type in swaps:
 ❶ if swap_type == 'A' and ball_location == 1:
       ball_location = 2
 ❷ if swap_type == 'A' and ball_location == 2:
       ball_location = 1
   if swap_type == 'B' and ball_location == 2:
       ball_location = 3
   if swap_type == 'B' and ball_location == 3:
       ball_location = 2
   if swap_type == 'C' and ball_location == 1:
       ball_location = 3
   if swap_type == 'C' and ball_location == 3:
       ball_location = 1

print(ball_location)
```

Если мы говорим, что код неверен, это значит, что он не прошел по крайней мере один тест. Можете ли вы найти тестовый пример, который даст неверный ответ?

Вот один из таких тестовых примеров:

```
A
```

Возможно и логично, что шарик может менять позицию не более одного раза за перемещение. Но Python автоматически запускает написанный вами код независимо от того, соответствует он вашим ожиданиям или нет. В данном случае выполняется лишь одно перемещение, поэтому шарик сдвинется один раз. На первой и единственной итерации цикла `for` Python проверяет выражение ❶. Оно истинно, поэтому он присваивает переменной `ball_location` значение `2`. Затем Python проверяет выражение ❷. Поскольку мы только что изменили значение `ball_location` на `2`, тут тоже получается истина! Поэтому Python устанавливает `ball_location` равным `1`. В результате получаем `1`, хотя должно быть `2`.

Это пример *логической ошибки* — такой, из-за которой программа следует неправильной логике и дает неверный ответ. Обычно это называют *просто* ошибкой. Работа программиста над своим кодом в попытке исправить ошибки называется *отладкой*.

Часто простого тестового примера достаточно, чтобы продемонстрировать: программа работает неверно. Когда вы пытаетесь сузить круг проблем, не начинайте с длинных тестовых примеров. Результаты таких тестов сложно проверить вручную, так как поток выполнения получается сложным и непонятным. Простой тестовый пример, напротив, не заставляет нашу программу перетруждаться, а если ошибка найдена, то искать виновника долго не придется. Придумать небольшие целевые тестовые примеры не всегда легко, но этот навык можно отточить на практике.

Отправьте правильный код на сайт, а затем мы двинемся дальше.

Прежде чем продолжить, можете попробовать выполнить упражнения 1 и 2, приведенные в конце главы.

Задача 6. Занятые места

Вы уже знаете, как перебирать символы строки. Но иногда нужно знать, на какой позиции мы находимся в строке, а не только какой символ в ней хранится. Эта задача — именно такая.

Задача с сайта DMOJ, код `ccc18j2`.

Постановка задачи

У вас есть парковка на n парковочных мест. Вчера вы записывали, какие места свободны, а какие заняты. Сегодня снова записали то же самое. Укажите количество парковочных мест, которые были заняты в обоих случаях.

Входные данные

Входные данные состоят из трех строк.

- В первой строке записано целое число n — количество парковочных мест (число от 1 до 100).
- Во второй строке из n символов записана информация о занятости мест вчера. Символ `C` обозначает занятое парковочное место, а символ `.` — свободное. Например, строка `CC.` означает, что два первых парковочных места были заняты, а третье свободно.
- В третьей строке из n символов показана занятость парковки на сегодня (формат тот же, что и у второй).

Выходные данные

Выведите количество парковочных мест, которые были заняты на протяжении обоих дней.

Новый вид циклов

У нас может быть до 100 парковочных мест, поэтому неудивительно, что в программе будет цикл. Цикл `for`, который мы рассмотрели в предыдущей задаче, позволит пройти через цепочку парковочных мест:

```
>>> yesterday = 'CC.'
>>> for parking_space in yesterday:
...     print('The space is ' + parking_space)
...
The space is C
The space is C
The space is .
```

Теперь мы знаем, было ли то или иное место занято вчера. Но нам также нужно знать, занято ли оно сегодня.

Рассмотрим тестовый пример:

```
3
CC.
.C.
```

Вчера первое парковочное место было занято. Но было ли оно занято в оба дня? Чтобы ответить на этот вопрос, нужно взглянуть на соответствующий символ в сегодняшней строке. Там символ `.` (пусто), значит, парковочное место сегодня свободно.

А как насчет второго парковочного места? Вчера было занято, сегодня тоже. И это единственное такое парковочное место, то есть правильный результат для этого тестового примера — 1.

Перебор символов одной строки не помогает нам найти соответствующие символы в другой строке. Но если бы мы могли отслеживать, где находимся в строке (по номеру парковочного места), то могли бы найти соответствующий символ в каждой строке. Цикл `for`, который мы знаем, не позволяет этого сделать. Но можно использовать индексирование и новый тип цикла `for`.

Индексирование

Каждый символ в строке имеет индекс, который указывает на его местоположение. Первый символ имеет индекс `0`, второй символ — индекс `1` и т. д. В естественном языке, в том числе английском, мы часто начинаем считать с 1. Но в большинстве языков программирования, включая Python, начинают считать с 0.

Чтобы использовать индекс, его нужно указать после имени строки в квадратных скобках. Рассмотрим пару примеров индексации:

```
>>> word = 'splore'
>>> word[0]
's'
>>> word[3]
'o'
>>> word[5]
'e'
```

В индексе можно применять также переменные:

```
>>> where = 2
>>> word[where]
'l'
>>> word[where + 2]
'r'
```

Максимальный индекс, который мы можем использовать для непустой строки, — это ее длина минус 1 (у пустой строки нет действительного индекса). Например, длина строки `'splore'` равна 6, поэтому индекс `5` соответствует последней букве. Если взять большее значение, получим ошибку:

```
>>> word[len(word)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> word[len(word) - 1]
'e'
```

Как получить доступ ко второму справа символу в строке? См. код:

```
>>> word[len(word) - 2]
'r'
```

Есть способ попроще. Python поддерживает отрицательные индексы как еще один вариант доступа к символам. Индекс `-1` указывает на крайний правый символ, индекс `-2` — на второй справа символ и т. д.:

```
>>> word[-2]
'r'
>>> word[-1]
'e'
>>> word[-5]
'p'
>>> word[-6]
's'
>>> word[-7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Мы будем использовать индексацию для доступа к определенным позициям в данных о вчерашней и сегодняшней занятости парковки. Например, индекс `0` каждой строки позволяет получить информацию о первом парковочном месте, индекс 1 — о втором и т. д.

Но прежде, чем мы сможем реализовать этот план, нужно изучить новый вид цикла `for`.

ПРОВЕРИМ ЗНАНИЯ

Какой результат дает следующий код?

```
s = 'abcde'
t = s[0] + s[-5] + s[len(s) - 5]

print(t)
```

А. `aaa`

Б. `aae`

В. `aee`

Г. Этот код выдает ошибку.

Ответ: **А**. Каждый из трех индексов относится к первому символу в строке `'abcde'`. Первый, `s[0]`, указывает на `'a'`, потому что `'a'` находится в начале строки. Второй, `s[-5]`, относится к `'a'`, потому что это пятый символ справа. Наконец, `s[len(s) – 5]` тоже указывает на эту букву, так как 5 – 5 = 0.

Функция range и циклы

Функция `range` в Python генерирует диапазоны целых чисел, которые можно использовать для управления циклами `for`. Вместо того чтобы перебирать символы строки, с помощью функции `range` мы будем перебирать целые числа. Передав аргумент функции `range`, получаем диапазон от 0 до числа, на 1 меньшего, чем этот аргумент:

```
>>> for num in range(5):
...     print(num)
...
0
1
2
3
4
```

Обратите внимание, что `5` не выводится.

Если мы передадим функции два аргумента, то получим последовательность от первого до второго, не включая второй:

```
>>> for num in range(3, 7):
...     print(num)
...
3
4
5
6
```

Можно изменить *размер шага*, добавив еще и третий аргумент.

По умолчанию размер шага равен 1. Попробуем удвоить:

```
>>> for num in range(0, 10, 2):
...     print(num)
...
0
2
4
6
8
>>> for num in range(0, 10, 3):
...     print(num)
...
0
3
6
9
```

Можно также считать в обратном направлении, но *не так*:

```
>>> for num in range(6, 2):
...     print(num)
...
```

Так нельзя, потому что по умолчанию счет идет вверх. Но значение шага `-1` позволяет считать назад:

```
>>> for num in range(6, 2, -1):
...     print(num)
...
6
5
4
3
```

Чтобы выполнить отсчет от 6 до 0, включая 0, нужно, чтобы значение второго аргумента было `-1`:

```
>>> for num in range(6, -1, -1):
...     print(num)
...
6
5
4
3
2
1
0
```

Иногда бывает полезно быстро просмотреть числа в некотором диапазоне, не используя циклы. К сожалению, функция `range` не позволяет вывести числа напрямую:

```
>>> range(3, 7)
range(3, 7)
```

Но зато мы можем передать этот результат в функцию `list`, чтобы получить то, что нужно:

```
>>> list(range(3, 7))
[3, 4, 5, 6]
```

При вызове совместно с `range` функция `list` создает список целых чисел из заданного диапазона. Мы узнаем о списках позже, а пока помните, что ими можно пользоваться для работы с диапазонами.

ПРОВЕРИМ ЗНАНИЯ

Сколько итераций выполнит следующий цикл?

```
for i in range(10, 20):
    # Некий код
```

А. 9

Б. 10

В. 11

Г. 20

Ответ: **Б.** Функция `range` переберет числа от 10 до 19, а это 10 чисел.

Использование функции range для перебора индексов

Предположим, у нас есть строки, содержащие информацию о вчерашней и сегодняшней занятости парковки:

```
>>> yesterday = 'CC.'
>>> today = '.C.'
```

Имея индекс, мы можем посмотреть для него вчерашнюю и сегодняшнюю информацию:

```
>>> yesterday[0]
'C'
>>> today[0]
'.'
```

Можно применять цикл `for` и диапазоны для проверки каждой пары соответствующих символов. Мы знаем, что строки `yesterday` и `today` одинаковы по длине. Но она может быть от 1 до 100, поэтому нельзя сразу написать `range(3)`. Вместо этого нужно перебрать индексы от 0 до длины строки минус 1. Мы можем использовать длину одной из строк в качестве аргумента для функции `range`:

```
>>> for index in range(len(yesterday)):
...     print(yesterday[index], today[index])
...
C .
C C
. .
```

Я назвал переменную цикла `index`. Часто используется имя `i` (первая буква индекса) и `ind`. В дальнейшем буду применять имя `i`.

Не называйте переменную цикла `status` или `information`. В ходе работы с целыми числами они подразумевают нечто иное.

Решение задачи

Вооружившись диапазонами и циклами, мы готовы решить задачу. Переберем каждый индекс от начала до конца строки. Затем проверим, что находится по данному индексу во вчерашней и в сегодняшней информации. Используя вложенный оператор `if`, определим, было ли парковочное место занято оба дня.

Решение приведено в листинге 3.3.

Листинг 3.3. Определение занятости парковки

```
n = int(input())
yesterday = input()
today = input()

❶ occupied = 0

❷ for i in range(len(yesterday)):
    ❸ if yesterday[i] == 'C' and today[i] == 'C':
        ❹ occupied = occupied + 1

print(occupied)
```

В начале программы мы считываем три строки ввода: `n` означает количество парковочных мест, а `yesterday` и `today` — это вчерашняя и сегодняшняя информация о парковочных местах соответственно.

Обратите внимание на то, что количество парковочных мест (`n`) мы больше не задействуем. С помощью этого параметра мы могли бы узнать длину строк, но я решил им не пользоваться, потому что в реальных сценариях подобные переменные даются редко.

Мы применим переменную `occupied`, чтобы подсчитать количество парковочных мест, которые были заняты вчера и сегодня. Вначале она равна `0` ❶.

Теперь заводим цикл `for`, который перебирает `yesterday` и `today` ❷ по индексам. Для каждого индекса мы проверяем, было ли парковочное место занято вчера и сегодня ❸. Если так и было, добавляем это парковочное место в общую сумму, увеличив `occupied` на 1 ❹.

Когда цикл `for` закончится, все парковочные места будут проанализированы. Общее количество парковочных мест, которые были заняты вчера и сегодня, хранится в переменной `occupied`. Осталось лишь вывести ее.

Задача решена, можно отправлять код на сайт.

Задача 7. Тарифный план

Мы узнали, что циклы `for` полезны для обработки данных после их чтения из ввода. Помимо этого, они полезны для чтения самих данных. В этой задаче рассмотрим данные, распределенные по нескольким строкам, а цикл `for` поможет нам их считать.

Задача с сайта DMOJ, код `coci16c1p1`.

Постановка задачи

По тарифному плану мобильной связи Пьеро предоставляется x мегабайт трафика в месяц. Кроме того, трафик, который он не тратит в данном месяце, переносится на следующий месяц. Например, если значение переменной x равно 10, а Пьеро в данном месяце использовал всего 4 Мбайт, оставшиеся 6 Мбайт переносятся на следующий месяц, в котором у него будет 10 + 6 = 16 Мбайт.

Нам дано число мегабайтов данных, которые Пьеро тратил в каждый из первых n месяцев. Наша задача — определить, сколько мегабайтов у него будет в следующем месяце.

Входные данные

Входные данные состоят:

- из строки, содержащей целое число x — количество мегабайтов, потраченных Пьеро в этом месяце. Число x может иметь значение от 1 до 100;
- строки, содержащей целое число n — количество месяцев, в течение которых Пьеро использует данный тариф. Это число от 1 до 100;
- n строк, по одной на каждый месяц, в которых содержится число мегабайтов, потраченных в данном месяце. Каждое число равно как минимум 0 и никогда не превышает количество доступных мегабайтов (например, если x равно 10 и Пьеро в данный момент доступно 30 Мбайт, следующее число будет не более 30).

Выходные данные

Количество мегабайтов, доступных в следующем месяце.

Чтение ввода в цикле

Прежде во всех задачах мы всегда точно знали, сколько строк входных данных нужно будет считать. Например, в задаче о трех чашках мы считывали одну строку, а в задаче о парковке — три строки. Здесь же не знаем заранее, сколько строк нужно будет считать, потому что их количество задается числом, которое считываем во второй строке.

Считаем первую строку входных данных:

```
monthly_mb = int(input())
```

(Я дал переменной имя `monthly_mb` вместо `x`, чтобы было понятнее.)

Считаем вторую строку входных данных:

```
n = int(input())
```

А вот теперь уже не обойтись без цикла. Функция `range` здесь будет как нельзя кстати, потому что благодаря ей мы можем запустить цикл ровно `n` раз:

```
for i in range(n):
    # Обработка данных за месяц
```

Решение задачи

В предлагаемом мной решении будем отслеживать количество мегабайтов, оставшихся после предыдущего месяца. Будем называть его `excess`.

Рассмотрим тестовый пример:

```
10
3
4
12
1
```

Каждый месяц Пьеро получает новые 10 Мбайт данных, плюс к этому нужно обработать количество трафика, которое он использовал за три месяца. В первый месяц ему дается 10 Мбайт, из которых потрачено 4 Мбайт, поэтому излишек

составляет 6 Мбайт. Во второй месяц Пьеро получает еще 10 Мбайт, поэтому теперь у него 16 Мбайт. Он использует 12 Мбайт, поэтому переносимый излишек составляет 16 – 12 = 4 Мбайт. На третий месяц Пьеро получает еще 10 Мбайт, поэтому теперь у него 14 Мбайт. В этом месяце он потратил 1 Мбайт, поэтому переносимый излишек составляет 14 – 1 = 13 Мбайт.

Нам необходимо узнать количество мегабайтов, доступных Пьеро в следующем (то есть четвертом) месяце. У него есть 13 Мбайт, которые остались после первых трех месяцев, и он получил еще 10 Мбайт на четвертый месяц, так что теперь у него 13 + 10 = 23 Мбайт.

Приступив к написанию кода, в котором все это реализовано, я не добавил последние 10 Мбайт, поэтому получил в результате 13 Мбайт вместо 23 Мбайт. Я думал только об излишке и забыл, что в последнем месяце требуется найти не накопленный излишек, а общее количество доступных мегабайтов. То есть на 10 Мбайт больше.

В листинге 3.4 приведен исправленный код.

Листинг 3.4. Решение задачи

```
monthly_mb = int(input())
n = int(input())

excess = 0

❶ for i in range(n):
    used = int(input())
  ❷ excess = excess + monthly_mb - used

❸ print(excess + monthly_mb)
```

Переменная `excess` сначала имеет значение `0`. На каждой итерации цикла `for` мы присваиваем ей значение излишка, которое учитывает количество мегабайтов, предоставленных в месяц, и количество мегабайтов, использованных в этом месяце.

Цикл выполняется `n` раз, `n` равно количеству месяцев, в течение которого был активен тарифный план ❶. Значения переменной `i` — `0`, `1` и т. д. — нас не интересуют, потому что нет разницы, какой месяц обрабатывать. По этой причине значение `i` в программе не применяется.

Можно заменить имя переменной `i` на `_` (нижнее подчеркивание), чтобы явно указать, что эта переменная нам безразлична, но я оставлю имя `i` для согласованности с другими примерами.

В цикле мы читаем количество мегабайтов, использованных в этом месяце. Затем обновляем количество излишних мегабайтов ❷: суммируем прежний остаток

и количество мегабайтов, которое Пьеро получает в месяц, и вычитаем количество мегабайтов, потраченных в течение месяца.

Вычислив избыточное количество мегабайтов через n месяцев, мы выводим количество мегабайтов, доступных в следующем месяце ❺.

Решить задачу всегда можно несколькими способами. Программирование — это творческий процесс, и я люблю смотреть, какие стратегии и решения придумывают люди. Даже если вам удалось решить задачу, можете погуглить ее решение, чтобы узнать, как поступили другие. Кроме того, некоторые онлайн-судьи, такие как DMOJ, позволяют вам просматривать решения других пользователей, когда написанное вами проходит проверку. Если решение прошло все тесты, это позволяет вам увидеть другие подходы. Если нет — позволяет понять свою ошибку. Чтение чужого кода — отличный способ улучшить свои навыки программирования!

Можете ли вы придумать другой способ решить задачу о тарифном плане?

Вот подсказка: можно сперва вычислить общее количество мегабайтов, предоставленных Пьеро, а затем вычесть количество мегабайтов, которые он потратил. Попробуйте понять, как это сделать, прежде чем продолжить!

Общее количество мегабайтов, предоставленных Пьеро, включая трафик за будущий месяц, равно $x(n + 1)$, где x — количество мегабайтов, выделяемых в месяц. Чтобы определить количество мегабайтов, доступных в следующем месяце, мы можем взять эту сумму и вычесть трафик, который Пьеро использует каждый месяц. Это решение реализовано в листинге 3.5.

Листинг 3.5. Решение задачи, альтернативный подход

```
monthly_mb = int(input())
n = int(input())

total_mb = monthly_mb * (n + 1)

for i in range(n):
    used = int(input())
    total_mb = total_mb - used

print(total_mb)
```

Выберите решение, которое вам больше по душе, и отправьте его на сайт.

То, что интуитивно понятно одному человеку, может оказаться непонятным другому. Возможно, вы прочитаете описание кода, но не сможете разобраться в нем. Это означает лишь то, что вам нужен другой способ объяснения, который больше соответствует вашему образу мышления. Вы также можете отметить сложные объяснения и примеры, чтобы просмотреть их позже. Они могут оказаться на удивление полезными.

Резюме

В этой главе вы узнали о циклах `for`. Обычный цикл `for` может перебирать символы в последовательности, а с помощью `range` — перебирать целые числа в заданном диапазоне. В задачах, которые мы решали, обрабатывалось большое количество входных данных, и справиться с ними без цикла не получилось бы.

Цикл `for` — это цикл, позволяющий повторить код определенное количество раз. В Python есть еще один тип цикла, и с ним мы познакомимся в следующей главе. А зачем? Что нельзя делать с помощью цикла `for`? Хорошие вопросы! Но пока лишь отмечу, что попрактиковаться с циклами `for` — это прекрасный способ подготовиться к тому, что будет дальше.

Упражнения

Далее приведены задачи, которые вы можете попробовать решить.

1. DMOJ, задача Uncrackable с кодом `wc17c3j3`.
2. DMOJ, задача Magnus с кодом `coci18c3p1`.
3. DMOJ, задача English or French с кодом `ccc11s1`.
4. DMOJ, задача Multiple Choice с кодом `ccc11s2`.
5. DMOJ, задача Ljesvica с кодом `Jcoci12c5p1`.
6. DMOJ, задача Rijeci с кодом `coci13c3p1`.
7. DMOJ, задача Elder с кодом `coci18c4p1`.

Примечания

Задача «Три чашки» взята из открытого хорватского конкурса по информатике 2006/2007 годов, конкурс 5. Задача «Занятые места» — из Канадского компьютерного конкурса 2018 года, младший уровень. «Тарифный план» — из открытого хорватского конкурса по информатике 2016/2017 годов, конкурс 1.

4

Повторяющийся код: неопределенные циклы

Циклы `for` и их вариант с диапазоном, которые вы изучили в главе 3, удобны для просмотра строк или диапазона индексов. Но что нам делать, если строки у нас нет, а индексы нельзя представить в виде диапазона? Мы используем цикл `while`, о котором поговорим в этой главе. Его применение шире, и он позволяет обрабатывать ситуации, с которыми `for` не справляется.

Мы решим три задачи, в которых цикл `for` не работает: определение количества возможных запусков игрового автомата, составление списка воспроизведения песен до тех пор, пока пользователь не захочет остановиться, и декодирование закодированного сообщения.

Задача 8. Игровые автоматы

Сколько раз ты сможешь сыграть на игровом автомате, прежде чем у тебя закончатся деньги? Это тонкий вопрос, который зависит не только от наличия денег, но и от модели выигрышей. Мы увидим, что в этой ситуации нужен цикл `while`, а не цикл `for`.

Задача с сайта DMOJ, код `ccc00s1`.

Постановка задачи

Марта идет в казино, у нее в кармане n монет. В казино есть три игровых автомата, и она играет на них по порядку, пока не кончатся деньги. То есть она играет на первом игровом автомате, затем на втором и третьем, а дальше снова начинает с первого и т. д. Каждая игра стоит монету.

Игровые автоматы работают по следующим правилам.

- Первый игровой автомат выдает 30 монет за каждую 35-ю игру.
- Второй игровой автомат выдает 60 монет за каждую 100-ю игру.
- Третий игровой автомат выдает 9 монет за каждую 10-ю игру.
- В прочих случаях выигрыша нет.

Определите, сколько раз Марта сможет сыграть, пока у нее не кончатся деньги.

Входные данные

Входные данные состоят из четырех строк.

- В первой строке записано целое число n — количество монет, которое есть у Марты, в диапазоне от 1 до 1000.
- Во второй строке записано целое число, указывающее, сколько раз с момента последнего выигрыша играли на первом игровом автомате (до прихода Марты). Например, предположим, что на первом автомате сыграли 34 раза с момента последней выплаты. Тогда Марта выиграет 30 монет с первой же попытки.
- В третьей строке записано целое число, указывающее, сколько раз с момента последнего выигрыша играли на втором игровом автомате.
- В четвертой строке записано целое число, указывающее, сколько раз с момента последнего выигрыша играли на третьем игровом автомате.

Выходные данные

Строка, где `x` — количество игр, которые сможет сыграть Марта, пока не кончатся деньги:

```
Martha plays x times before going broke.
```

Пример тестового случая

Рассмотрим пример, чтобы убедиться, что мы понимаем постановку задачи:

```
7
28
0
8
```

Чтобы внимательно отследить игры Марты, нужно будет помнить шесть значений. Для этого удобно использовать таблицу, где в каждой строке будет описано состояние задачи после каждой игры. Графы:

- **«Игры»** — количество игр Марты;
- **«Монеты»** — количество монет, которое есть у Марты;
- **«Следующая игра»** — игровой автомат, на котором Марта будет играть в следующий раз;
- **«Игр на первом»** — количество игр на первом автомате с момента последнего выигрыша;
- **«Игр на втором»** — количество игр на втором автомате с момента последнего выигрыша;
- **«Игр на третьем»** — количество игр на третьем автомате с момента последнего выигрыша.

В начале задачи Марта еще не играла, у нее семь монет, и играть она будет на первом игровом автомате. На первом игровом автомате играли 28 раз с момента последнего выигрыша, на втором — 0 раз, а на третьем — 8 раз. Текущее состояние выглядит так:

Игры	Монеты	Следующая игра	Игр на первом	Игр на втором	Игр на третьем
0	7	Первый	28	0	8

Марта начинает с игры на первом игровом автомате. Это стоит ей монету. Поскольку с момента последней выплаты на нем сыграли 29 раз, а не 35, он ничего не выдаст. Марта перейдет ко второму автомату. Новое состояние таково:

Игры	Монеты	Следующая игра	Игр на первом	Игр на втором	Игр на третьем
1	6	Второй	29	0	8

Игра на втором игровом автомате стоит монету. Поскольку это первая игра с момента последней выплаты, автомат ничего не выдаст. Марта будет играть на третьем игровом автомате. Новое состояние:

Игры	Монеты	Следующая игра	Игр на первом	Игр на втором	Игр на третьем
2	5	Третий	29	1	8

Игра на третьем игровом автомате стоит монету. Поскольку с момента последней выплаты на этом автомате сыграли 9 раз, а не 10, он ничего не выдаст. Затем Марта вернется к первому автомату. Новое состояние:

Игры	Монеты	Следующая игра	Игр на первом	Игр на втором	Игр на третьем
3	4	Первый	29	1	9

Марта играет на первом игровом автомате:

Игры	Монеты	Следующая игра	Игр на первом	Игр на втором	Игр на третьем
4	3	Второй	30	1	9

Затем на втором:

Игры	Монеты	Следующая игра	Игр на первом	Игр на втором	Игр на третьем
5	2	Третий	30	2	9

Монетки почти кончились, но впереди хорошие новости, потому что теперь она сыграет на третьем игровой автомат. С момента последней выплаты на нем сыграли девять раз. Таким образом, следующая игра будет десятой и Марта получит девять монет. У нее было две монеты, одну она потратит на игру, а затем получит девять, в итоге у нее будет 2 – 1 + 9 = 10 монет.

Игры	Монеты	Следующая игра	Игр на первом	Игр на втором	Игр на третьем
6	10	Первый	30	2	0

Обратите внимание на то, что теперь на третьем игровом автомате сыграли 0 раз после выигрыша, так как он только что был.

Это только шесть игр. Но можно и продолжить. Если вы попробуете, то увидите, что Марта больше ничего не выиграет и после десяти игр (в общей сложности 16) ей придется пойти домой.

Ограничения цикла for

В главе 3 мы изучили цикл `for`. Стандартный цикл `for` перебирает последовательность, например строку. В этой задаче их нет.

Цикл `for` с функцией `range` перебирает диапазон целых чисел и может использоваться для повторения кода заданное количество раз. Но сколько раз нужно будет выполнить цикл на игровых автоматах: 10, 50? Неизвестно. Это зависит от того, сколько раз Марта сможет сыграть с имеющимся запасом монет.

Итак, ни строк, ни диапазона нет. Если бы у нас был только цикл `for`, мы бы застряли.

Но у нас есть *цикл* `while` — более общий по своей сути цикл в Python. Мы можем писать циклы `while`, которые не имеют ничего общего со строками или последовательностями целых чисел. В обмен на эту дополнительную гибкость нам придется быть немного более осторожными и взять на себя немного бóльшую ответственность при написании цикла. Давайте разбираться!

Цикл while

Для написания цикла `while` применяется оператор `while`. Цикл `while` управляется логическим выражением. Если логическое выражение истинно, Python выполняет итерацию цикла `while`. Если выражение по-прежнему истинно, Python продолжает выполнять его до тех пор, пока оно не станет ложным. Если логическое выражение изначально имело значение `False`, цикл не выполняется.

Цикл `while` называется *неопределенным*, так как количество итераций не может быть известно заранее.

Использование циклов while

Начнем со следующего примера цикла `while`:

```
❶ >>> num = 0
❷ >>> while num < 5:
  ...     print(num)
❸ ...     num = num + 1
  ...
  0
  1
  2
  3
  4
```

В цикле `for` нужно создавать переменную цикла, но для этого не требуется использовать оператор присваивания явно. А вот в цикле `while` само ничего не делается. Если нужна переменная для перебора значений в цикле `while`, мы должны создать ее самостоятельно. Что мы и делаем, присваивая `num` значение `0` ❶.

Сам цикл `while` управляется логическим выражением `num < 5` ❷. Если оно истинно, то выполнится код в теле цикла. Сначала переменная `num` равна `0`, поэтому логическое выражение истинно. Так что мы запускаем блок цикла, который выводит `0`, а затем увеличивает `num` на единицу ❸.

Мы возвращаемся к началу цикла и снова вычисляем логическое выражение `num < 5`. Поскольку `num` равно `1`, выражение истинно. Поэтому мы снова запускаем блок цикла, который выводит `1`, а затем увеличиваем `num` до `2`.

Возвращаемся к началу цикла и проверяем выражение. Оно еще истинно, так как значение переменной `num` равно `2`. Это запускает еще одну итерацию цикла, которая выводит `2` и увеличивает значение переменной `num` до `3`.

Те же действия выполняются еще две итерации цикла: при `num = 3` и `num = 4`. Когда `num = 5`, логическое выражение `num < 5` наконец принимает значение `False`, что завершает цикл.

Важно не забывать увеличивать значение `num` ❸. Цикл `for` автоматически меняет значение переменной. Но в цикле `while` за нас никто ничего не сделает — мы сами должны обновлять переменные, чтобы цикл завершился. Если забудем увеличить `num`, получится вот что:

```
>>> num = 0
>>> while num < 5:
...     print(num)
...
0
0
0
0
0
0
0
0
... forever
```

Если вы запустите этот код на своем компьютере, экран заполнится нулями и вам придется прервать программу. Для этого можно нажать клавиши Ctrl+C или закрыть окно Python.

Здесь все дело в том, что выражение `num < 5` остается истинным всегда и внутри цикла ничего не предпринимается, чтобы это изменить. Ситуация, когда цикл никогда не завершается, называется *бесконечным циклом*. Писать бесконечные циклы `while` на удивление легко. Если вы видите, что выводятся одни и те же значения или программа вообще ничего не делает, вероятно, вы застряли в бесконечном цикле. Внимательно проверьте логическое выражение цикла `while` и убедитесь, что у него есть завершение.

Мы можем делать с переменной `num` что угодно. Далее приведен цикл `while`, считающий до трех:

```
>>> num = 0
>>> while num < 10:
...     print(num)
...     num = num + 3
...
0
3
6
9
```

А вот цикл while, отсчитывающий от 4 до 0:

```
>>> num = 4
❶ >>> while num >= 0:
...     print(num)
...     num = num - 1
...
4
3
2
1
0
```

Обратите внимание на то, что здесь я использовал оператор >=, а не > ❶. Таким образом цикл while запускается при num = 0, но это по желанию.

ПРОВЕРИМ ЗНАНИЯ

Какой результат дает следующий код?

```
n = 3
while n > 0:
    if n == 5:
        n = -100
    print(n)
    n = n + 1
```

А.
```
3
4
```

Б.
```
3
4
5
```

В.
```
3
4
-100
```

Г.
```
3
4
5
-100
```

Ответ: **В**. Выражение цикла проверяется только в начале каждой итерации. Даже если в какой-то момент во время выполнения оно станет ложным, текущая итерация будет доведена до конца.

Поскольку `3` больше `0`, выполняется итерация цикла. Блок `if` пропускается, так как его логическое выражение ложно, поэтому данная итерация выводит `3` и задает `n = 4`. `4` больше, чем `0`, выполняется еще одна итерация цикла, которая выводит `4` и задает `n = 5`. `5` больше, чем `0`, и выполняется еще одна итерация цикла. На этот раз блок `if` устанавливает `n = -100`. Выводится значение `-100`, и `n` становится равной `-99`. Здесь мы останавливаемся, потому что `n > 0` больше не истинно.

ПРОВЕРИМ ЗНАНИЯ

Какой результат дает следующий код?

```
x = 6
while x > 4:
    x = x - 1
    print(x)
```

А.
```
6
5
```

Б.
```
6
5
4
```

В.
```
5
4
```

Г.
```
5
4
3
```

Д.
```
6
5
4
3
```

Ответ: **В**. Многие циклы `while` делают что-то, а затем обновляют переменную цикла, но в данном случае это не так. Цикл сначала уменьшает переменную, а потом выводит ее. `6` больше, чем `4`, выполняется итерация цикла, которая присваивает переменной `x` значение `5`, а затем выводит `5`. `5` больше `4`, так что выполняется еще одна итерация, которая присваивает `x` значение `4` и выводит `4`. На этом все — `4` не больше `4`, поэтому цикл завершается.

Вложенные циклы

Мы можем вкладывать внутрь циклов `while` другие циклы, как уже делали с циклом `for`. В главе 3 я говорил, что внутренний цикл `for` завершает все свои итерации до того, как начнется следующая итерация внешнего цикла. То же самое и с циклами `while`. Вот пример:

```
>>> i = 0
>>> while i < 3:
...     j = 8
...     while j < 11:
...         print(i, j)
...         j = j + 1
...     i = i + 1
...
0 8
0 9
0 10
1 8
1 9
1 10
2 8
2 9
2 10
```

Каждое значение `i` участвует в трех строках вывода, по одной для каждой итерации внутреннего цикла по `j`.

ПРОВЕРИМ ЗНАНИЯ

Какой результат дает следующий код?

```
x = 0
y = 1
while x < 3:
    while y < 3:
        print(x, y)
        y = y + 1
    x = x + 1
```

А. 2

Б. 3

В. 6

Г. 8

Д. 9

Ответ: **A**. Логическое выражение внешнего цикла `x < 3` истинно, поэтому мы выполняем итерацию внешнего цикла. При `y = 1` и `y = 2` выполняются две итерации внутреннего цикла, каждая из которых выводит строку вывода. Получаем две строки вывода.

Но значение `y` в коде не сбрасывается! Следовательно, `y < 3` не истинно и дальнейших итераций внутреннего цикла не будет.

Забыть сбросить переменную цикла — распространенная ошибка при работе с вложенными циклами `while`.

Использование логических операторов

В этой задаче цикл должен работать, пока у Марты есть хотя бы одна монета. Код выглядит так:

```
while quarters >= 1:
```

Этого простого логического выражения будет достаточно для решения задачи. Но, как и в случае с операторами `if`, логическое выражение после слова `while` может быть усложнено логическими операторами или сравнениями. Пример:

```
>>> x = 4
>>> y = 10
>>> while x <= 10 and y <= 13:
...     print(x, y)
...     x = x + 1
...     y = y + 1
...
4 10
5 11
6 12
7 13
```

Цикл `while` управляется логическим выражением `x <= 10 and y <= 13`. Как и в случае с любым оператором `and`, оба его операнда должны иметь значение `True`, чтобы все выражение было истинным. Когда `x = 8`, а `y = 14`, цикл завершается, потому что выражение `y <= 13` становится равным `False`.

Решение задачи

Для решения задачи, как вы уже знаете, понадобится цикл `while`, а не цикл `for`, потому что мы не знаем заранее, сколько потребуется итераций. Каждая итерация цикла будет имитировать текущую игру на автомате. Когда цикл завершится, у Марты не останется монеток, и мы выведем количество сыгранных ею игр.

Вот что мы делаем на каждой итерации.

- Уменьшаем число монеток Марты на одну, так как игра стоит одну монетку.
- Если Марта в данный момент находится на первом игровом автомате, запускаем игру на нем. Для этого увеличиваем количество сыгранных на нем игр. Если это 35-я игра, то выдаем Марте выигрыш и сбрасываем количество игр на автомате до 0.
- Если Марта в данный момент находится на втором игровом автомате, играем на нем так же, как делали это на первом.
- Если Марта в данный момент находится на третьем игровом автомате, играем на нем аналогично тому, как делали на первом.
- Увеличиваем количество игр Марты, так как она только что сыграла.
- Переходим к следующему автомату. Если Марта только что сыграла на первом автомате, переходим ко второму, если играла на втором, переходим к третьему, а если только что сыграла на третьем, возвращаемся к первому.

Поскольку наши программы становятся длиннее, написать план наподобие приведенного — полезный метод, позволяющий держать сложность под контролем и двигаться в нужную сторону. Благодаря ему мы можем проверять ход задачи и ничего не забывать.

Код приведен в листинге 4.1.

Листинг 4.1. Решение задачи

```
quarters = int(input())
first = int(input())
second = int(input())
third = int(input())
plays = 0
❶ machine = 0

❷ while quarters >= 1:
    ❸ quarters = quarters - 1

    ❹ if machine == 0:
          first = first + 1
        ❺ if first == 35:
              first = 0
              quarters = quarters + 30
          elif machine == 1:
              second = second + 1
              if second == 100:
                  second = 0
                  quarters = quarters + 60
```

```
        elif machine == 2:
            third = third + 1
            if third == 10:
                third = 0
                quarters = quarters + 9
    ❻ plays = plays + 1
    ❼ machine = machine + 1
    ❽ if machine == 3:
          machine = 0

print('Martha plays', plays, 'times before going broke.')
```

В переменной `quarters` хранится количество монет, которые есть у Марты. Переменные `first`, `second` и `third` отслеживают количество игр с момента выигрыша на первом, втором и третьем автоматах соответственно.

В переменной `machine` хранится игровой автомат, на котором Марта сыграет в следующий раз. Первый автомат обозначается цифрой 0, второй — цифрой 1, а третий — 2. Таким образом, значение 0 означает, что следующая игра будет на первом автомате ❶.

Можно было бы использовать номера 1, 2 и 3 вместо 0, 1 и 2. Или строковые значения `'first'`, `'second'` и `'third'`. Но нумерация элементов с нуля — это обычное дело, так что не удивляйтесь.

Последняя переменная в этой программе — `plays`, которая отслеживает количество игр Марты. Мы выведем ее, как только у нее кончатся монеты.

Основная часть программы состоит из цикла `while`, который повторяется до тех пор, пока у Марты есть монеты ❷.

На каждой итерации цикла имитируется одна игра на автомате. Таким образом, сперва мы уменьшаем богатство Марты на единицу ❸. Затем играем на текущем игровом автомате.

А на каком мы автомате, кстати, — первом, втором или третьем? Нам нужен оператор `if`, чтобы ответить на этот вопрос.

Сначала проверяем, находимся ли мы на игровом автомате под номером 0 ❹. Если да, то увеличиваем количество игр этого автомата на единицу. Чтобы определить, получает ли Марта деньги, мы проверяем, сыграли ли на этом автомате ровно 35 раз с момента последнего выигрыша ❺. Если да, то сбрасываем его количество игр до 0 и увеличиваем количество монет Марты на 30.

В программе есть несколько уровней вложенности, поэтому надо проверить, что логика кода верна. Например, каждый раз, играя на первом автомате, мы увеличиваем его количество игр на единицу. Поскольку мы платим Марте только после каждых 35 игр, нужно сбросить счетчик до 0 ❻!

Второй и третий игровые автоматы мы обрабатываем так же, как и первый. Разница лишь в том, что у каждого автомата свои выигрыш и частота их выдачи.

Поиграв на игровом автомате, мы увеличиваем количество игр Марты на единицу ❻. Теперь остается лишь перейти к следующему автомату и выполнить следующую итерацию, если надо.

Чтобы поменять автомат, мы увеличиваем значение переменной `machine` на 1 ❼. Если бы мы были на машине 0, это переместило бы нас на автомат 1. Если бы мы были на автомате 1, это переместило бы нас на автомат 2. Если бы мы были на автомате 2, это переместило бы нас на автомат 3.

...Автомат 3? Стоп, у нас нет такого! После автомата 2 нужно перемещаться на автомат 0. Для этого добавляем проверку: если мы только что перешли на автомат с номером 3 ❽, то сбрасываем значение `machine` до 0.

Когда цикл завершается, у Марты не остается монет. В качестве последнего шага мы выводим нужное предложение с числом игр.

В этом коде мы делаем кучу всего: останавливаем цикл, когда у Марты не осталось монет, отслеживаем номер текущего автомата, выдаем выигрыш, когда это необходимо, и считаем число игр. Вы можете сразу же отправить код на сайт, а можете вместо этого подумать, нельзя ли написать его по-другому. Что произойдет, если вы будете менять количество игр на 1 в начале цикла, а не в конце? Имеет ли значение, в какой момент менять число монет? Можно ли использовать новые переменные, чтобы отслеживать, сколько раз Марта играла на каждом игровом автомате, а не менять номер текущего автомата? Я настоятельно рекомендую вам поэкспериментировать с вариантами реализации этой задачи. Если вы внесете изменения и код перестанет проходить тесты — отлично! Это даст возможность поучиться исправлять неверный код и поможет понять, почему изменения кода иногда меняют его поведение.

В следующих двух разделах мы продолжим углубляться в код. Применим оператор `%`, чтобы уменьшать количество необходимых переменных, и задействуем форматированные строки для упрощения вывода.

Оператор деления по модулю

В главе 1 мы говорили об операторе `%`, используемом для вычисления остатка от целочисленного деления. Например, при делении 16 на 5 остаток равен 1:

```
>>> 16 % 5
1
```

При делении 15 на 5 получаем остаток 0, потому что 5 — делитель 15:

```
>>> 15 % 5
0
```

Второй операнд определяет диапазон значений, которые может вернуть оператор `%`. Возможные возвращаемые значения: от 0 до второго операнда, но не включая его. Например, если второй операнд равен 3, то оператор `%` может вернуть значения 0, 1 и 2. По мере увеличения первого операнда мы циклически перебираем все возможные возвращаемые значения. Вот пример:

```
>>> 0 % 3
0
>>> 1 % 3
1
>>> 2 % 3
2
>>> 3 % 3
0
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

Обратите внимание на шаблон: 0, 1, 2, 0, 1, 2 и т. д.

При таком поведении выполняется подсчет до указанного числа, а затем возврат к нулю. Именно это нам и нужно для перебора игровых автоматов: мы играем на автомате 0, затем 1, затем 2, потом вновь 0, 1, 2, и опять 0, 1 и т. д. (Это еще одна причина, по которой я использовал для обозначения игровых автоматов значения 0, 1 и 2, а не какие-то другие.)

Предположим, что переменная `play` обозначает количество игр, сыгранных Мартой. Чтобы определить следующий автомат (0, 1 или 2), мы можем применить оператор `%`. Предположим, что Марта играла на некотором автомате, и мы хотим знать, на каком будет играть дальше. Она будет играть на игровом автомате 1, а оператор `%` сообщает нам, что:

```
>>> plays = 1
>>> plays % 3
1
```

Если Марта сыграла шесть раз, значит, она успела посетить игровые автоматы 0, 1, 2, 0, 1, 2. Следующий игровой автомат, на котором она сыграет, будет автоматом 0. И поскольку она сыграла на всех трех машинах дважды, оператор `%` даст 0:

```
>>> plays = 6
>>> plays % 3
0
```

В качестве последнего примера предположим, что Марта сыграла 11 раз. Она прошла три полных цикла: 0, 1, 2, 0, 1, 2, 0, 1, 2. Это девять. После еще двух игр Марта дойдет до автомата 2:

```
>>> plays = 11
>>> plays % 3
2
```

То есть мы можем определить, на каком автомате играть, не используя переменную `machine`.

Можно также использовать оператор `%`, чтобы упростить определение того, выиграет ли Марта следующую игру на текущем игровом автомате. Рассмотрим первый автомат. В листинге 4.1 мы считали количество игр с момента предыдущего выигрыша. Если это число 35, то мы платим Марте и сбрасываем счетчик до 0. Но если используем оператор `%`, сбрасывать счетчик нет необходимости. Мы можем просто проверить, кратно ли 35 число игр, и выдать выигрыш, если это так. Проверить, кратно ли число 35, можно с помощью оператора `%`. Число кратно 35, если его деление на 35 не дает остатка:

```
>>> first = 35
>>> first % 35
0
>>> first = 48
>>> first % 35
13
>>> first = 70
>>> first % 35
0
>>> first = 175
>>> first % 35
0
```

Мы можем проверить выражение `first % 35 == 0`, чтобы определить, платить ли Марте выигрыш. Я переписал листинг 4.1 с применением оператора `%`. Новый код приведен в листинге 4.2.

Листинг 4.2. Решение задачи с использованием оператора %

```
quarters = int(input())
first = int(input())
second = int(input())
third = int(input())

plays = 0

while quarters >= 1:
 ❶ machine = plays % 3
   quarters = quarters - 1
```

```
    if machine == 0:
        first = first + 1
    ❷ if first % 35 == 0:
            quarters = quarters + 30
        elif machine == 1:
            second = second + 1
            if second % 100 == 0:
              quarters = quarters + 60
        elif machine == 2:
            third = third + 1
            if third % 10 == 0:
                quarters = quarters + 9

    plays = plays + 1

print('Martha plays', plays, 'times before going broke.')
```

Я использовал оператор % двумя способами, описанными в этом разделе: чтобы определить текущий автомат по количеству игр ❶ и чтобы определить, выиграет ли Марта в данной игре (например, ❷).

Применение % только для вычисления остатка от деления ограничивает возможности оператора. Всякий раз, когда вам нужно считать в цикле (0, 1, 2, 0, 1, 2), подумайте, можно ли упростить код с помощью оператора %.

Форматированные строки

Последнее, что мы делаем в решении задачи, — выводим на экран нужное предложение, например:

```
print('Martha plays', plays, 'times before going broke.')
```

Приходится закончить первую строку, чтобы вывести количество игр, а затем начать новую строку для вывода второй половины предложения. Кроме того, мы используем несколько аргументов функции `print`, чтобы правильно преобразовать вывод в строку. Если бы мы сохраняли полученную строку, а не выводили ее, нам понадобилось бы преобразование `str`:

```
>>> plays = 6
>>> result = 'Martha plays ' + str(plays) + ' times before going broke.'
>>> result
'Martha plays 6 times before going broke.'
```

Склеивание строк и целых чисел вместе естественно для данного простого предложения, но такое решение не масштабируется. Вот что получится, когда мы попытаемся вставить в предложение три числа:

```
>>> num1 = 7
>>> num2 = 82
>>> num3 = 11
>>> 'We have ' + str(num1) + ', ' + str(num2) + ', and ' + str(num3) + '.'
'We have 7, 82, and 11.'
```

За этими кавычками и пробелами трудно уследить.

Самый гибкий способ построить строку, состоящую из строк и чисел, — использовать *форматированную строку*. Вот как выглядит предыдущий пример с такой строкой:

```
>>> num1 = 7
>>> num2 = 82
>>> num3 = 11
>>> f'We have {num1}, {num2}, and {num3}.'
'We have 7, 82, and 11.'
```

Обратите внимание на букву `f` перед открывающей кавычкой строки. `f` означает `format`, потому что `f`-строки позволяют форматировать содержимое строки. Внутри `f`-строки можно помещать выражения в фигурные скобки. По мере сборки строки каждое выражение заменяется своим значением и вставляется в нее. В результате получается обычная строка:

```
>>> type(f'hello')
<class 'str'>
>>> type(f'{num1} days')
<class 'str'>
```

Выражения в фигурных скобках могут быть более сложными, чем просто имена переменных:

```
>>> f'The sum is {num1 + num2 + num3}'
'The sum is 100'
```

Мы можем применять `f`-строки в задаче об игровых автоматах. Вот как это будет выглядеть:

```
print(f'Martha plays {plays} times before going broke.')
```

Я думаю, что даже в этом простейшем примере использование форматированных строк добавляет коду ясности. Помните о них, когда нужно будет собрать строку из более мелких кусочков.

Но есть важный момент: форматированные строки были добавлены в версии Python 3.6, которая на момент написания книги является последней версией языка. В более старых версиях Python форматированные строки вызывают синтаксические ошибки.

Если вы используете f-строки, то на сайте, куда отправляете решения, должен быть выбран Python 3.6 или более новая версия.

Прежде чем продолжить, можете попробовать решить упражнение 1 к этой главе.

Задача 9. Список воспроизведения

Иногда мы не знаем заранее, сколько входных данных у нас будет. В этой задаче мы увидим, что в таких случаях нужен цикл `while`.

Задача с сайта DMOJ, код `ccc08j2`.

Постановка задачи

У нас есть пять любимых песен с названиями A, B, C, D и E. Мы создали из них плейлист и используем приложение для управления им. Песни воспроизводятся в порядке A, B, C, D, E. В приложении есть четыре кнопки:

- кнопка 1 перемещает первую песню плейлиста в конец. Например, если текущий плейлист — A, B, C, D, E, он изменится на B, C, D, E, A;
- кнопка 2 перемещает последнюю песню плейлиста в начало. Например, плейлист A, B, C, D, E превратится в E, A, B, C, D;
- кнопка 3 меняет местами первые две песни в плейлисте. Например, плейлист A, B, C, D, E превратится в B, A, C, D, E;
- кнопка 4 воспроизводит плейлист!

Работа программы зависит от нажатий кнопок пользователем. Когда он нажимает кнопку 4, программа должна вывести порядок песен в плейлисте.

Входные данные

Входные данные состоят из пар строк, в которых первая строка содержит номер кнопки (1, 2, 3 или 4), а вторая — количество нажатий на нее (от 1 до 10). То есть первая строка — это номер кнопки, вторая — количество нажатий, третья — номер кнопки, четвертая — количество нажатий и т. д. Вход заканчивается двумя строками:

```
4
1
```

Эти данные означают, что пользователь нажал кнопку 4 один раз.

Выходные данные

Вывести порядок песен в плейлисте после нажатия всех кнопок. Выходные данные выводятся одной строкой с пробелами между парами песен.

Срезы строк

В основе решения задачи лежит цикл `while`, который будет выполняться до тех пор, пока не будет нажата кнопка 4. В каждой итерации мы станем читать две строки и обрабатывать их. Получается такая структура:

```
❶ button = 0

  while button != 4:
      # Чтение кнопки
      # Чтение количества нажатий
      # Обработка нажатий кнопки
```

Перед циклом `while` мы создаем переменную `button` и присваиваем ей значение `0` ❶. Без этого переменной `button` не существовало бы и мы получили бы ошибку `NameError` в логическом выражении цикла `while`. Любое число, кроме 4, будет запускать первую итерацию цикла.

В цикле `while` мы будем использовать цикл `for` для обработки нажатий кнопок. Для каждого нажатия с помощью оператора `if` станем проверять, какая именно кнопка была нажата. Нам понадобятся четыре блока операторов `if` — по количеству кнопок.

Поговорим о том, как обработать каждую кнопку. Кнопка 1 перемещает первую песню плейлиста в конец. Поскольку мы работаем с небольшим и известным количеством песен, для объединения символов достаточно будет строковой индексации. Помните, что первый символ строки имеет индекс `0`, а не `1`. Можно поместить этот символ в конец строки следующим образом:

```
>>> songs = 'ABCDE'
>>> songs = songs[1] + songs[2] + songs[3] + songs[4] + songs[0]
>>> songs
'BCDEA'
```

Получается довольно громоздко даже для пяти песен. Для написания более общего и менее подверженного ошибкам кода можно использовать срезы строк.

Срез — это функция Python, которая позволяет выделять подстроку из строки (но строки работают с любой последовательностью, как мы увидим позже).

Он принимает два индекса: индекс, с которого мы хотим начать, и индекс, расположенный справа от места, где хотим закончить. Если мы используем, например, индексы 4 и 8, то получаем символы с индексами 4, 5, 6 и 7. В срезах применяются квадратные скобки с двоеточием между двумя индексами:

```
>>> s = 'abcdefghijk'
>>> s[4:8]
'efgh'
```

Срез `s` не меняет объект. Но мы можем сослаться на срез `s` с помощью оператора присваивания:

```
>>> s
'abcdefghijk'
>>> s = s[4:8]
>>> s
'efgh'
```

Здесь легко допустить ошибку на единицу и подумать, что в срез `s[4:8]` входит также символ с индексом `8`. Но это не так, как и `range(4, 8)` не включает `8`. Такое поведение может показаться несколько нелогичным, но зато оно одинаково и в функции `range`, и в срезах.

Двоеточие при взятии среза строки обязательно, а вот индексы начала и конца необязательны. Если мы опустим начальный индекс, Python возьмет срез с индекса `0`:

```
>>> s = 'abcdefghijk'
>>> s[:4]
'abcd'
```

Если не укажем конечный индекс, Python будет брать срез до конца строки:

```
>>> s[4:]
'efghijk'
```

А если не указать никаких индексов? Получим срез, равный строке:

```
>>> s[:]
'abcdefghijk'
```

Мы даже можем использовать в срезе отрицательные индексы. Вот пример:

```
>>> s[-4:]
'hijk'
```

Начальный индекс указывает на четвертый символ справа, то есть `'h'`, а конечный индекс опускается. Таким образом мы получаем срез от буквы `h` до конца строки.

В отличие от индексирования, срезы никогда не приводят к ошибкам индексации. Если использовать индексы, которые находятся за пределами строки, Python срезает соответствующий конец строки:

```
>>> s[8:20]
'ijk'
>>> s[-50:2]
'ab'
```

Срезы строк помогут нам реализовать поведение кнопок 1, 2 и 3.

Вот как выглядит код кнопки 1:

```
>>> songs = 'ABCDE'
>>> songs = songs[1:] + songs[0]
>>> songs
'BCDEA'
```

Срез дает нам всю строку, за исключением символа с индексом `0` (при этом тут нет конкретных указаний на строку длиной 5, и код будет работать с непустой строкой любой длины). Добавление этого отсутствующего символа приводит к тому, что первая песня перемещается в конец плейлиста. Срезы остальных строк делаются аналогично, позже вы увидите это в коде.

ПРОВЕРИМ ЗНАНИЯ

Какой результат даст следующий код?

```
game = 'Lost Vikings'

print(game[2:-6])
```

А. `st V`

Б. `ost V`

В. `iking`

Г. `st Vi`

Д. `Viking`

Ответ: **А**. Символ с индексом `2` это `'s'` в слове `Lost`. Символ по индексу — это первая `i` в `Vikings`. Поскольку мы берем срез с `2` по `6`, не включая `6`, получаем `'st V'`.

ПРОВЕРИМ ЗНАНИЯ

Какой пароль получится после выполнения цикла?

```
valid = False

while not valid:
    s = input()
    valid = len(s) == 5 and s[:2] == 'xy'
```

А. `xyz`

Б. `xyabc`

В. `abcxy`

Г. Более чем один из перечисленных паролей выводит нас из цикла.

Д. Никакой — цикл никогда не выполняется и пароли не образуются.

Ответ: **Б.** Цикл `while` завершается, когда `valid = True` (потому что `not valid` в этом случае `False`). Единственный из данных паролей длиной 5 и с `'xy'` в начале — это `xyabc`. Таким образом, это единственный пароль, который удовлетворяет условию и заканчивает цикл.

Решение задачи

Вы попрактиковались в использовании циклов `while` для перебора нажатий кнопки и в применении срезов для манипуляций со строками и теперь готовы решить задачу. В листинге 4.3 приведен код.

Листинг 4.3. Составление списка воспроизведения песни

```
songs = 'ABCDE'

button = 0

❶ while button != 4:
      button = int(input())
      presses = int(input())
   ❷ for i in range(presses):
         if button == 1:
```

```
            ❸ songs = songs[1:] + songs[0]
          elif button == 2:
            ❹ songs = songs[-1] + songs[:-1]
          elif button == 3:
            ❺ songs = songs[1] + songs[0] + songs[2:]

❻ output = ''
  for song in songs:
      output = output + song + ' '

❼ print(output[:-1])
```

Цикл `while` продолжается до тех пор, пока не будет нажата кнопка 4 ❶. На каждой итерации цикла `while` мы читаем номер кнопки, а затем — количество нажатий на нее.

Теперь, уже внутри цикла `while`, нужно выполнить цикл по одному разу для каждого нажатия кнопки. Вспомните известные вам типы циклов, чтобы решить, какой из них использовать. Здесь лучше всего подойдет цикл `for` с функцией `range` ❷, поскольку это самый простой способ выполнить цикл ровно столько раз, сколько мы укажем.

Поведение внутри цикла `for` зависит от того, какая кнопка нажата. Поэтому мы задействуем оператор `if` для проверки номера кнопки и соответствующего изменения плейлиста. Если нажата кнопка 1, используем срезы, чтобы переместить первую песню в конец плейлиста ❸. Если нажата кнопка 2, применяем срезы, чтобы переместить последнюю песню в его начало ❹. Для этого мы начинаем с символа, находящегося в правом конце строки, а затем задействуем срезы для прибавления остальных символов. Для кнопки 3 нужно изменить список воспроизведения так, чтобы первые две песни поменялись местами. Мы строим новую строку из символов с индексами 1 и 0, а затем добавляем все символы, начиная с имеющего индекс 2 ❺.

Как только мы выйдем из цикла `while`, нужно будет вывести песни с пробелом между парами. Выводить песни просто так не получится, потому что между ними нет пробелов. Вместо этого мы собираем выходную строку с пробелами в нужных местах. Для этого начинаем с пустой строки ❻, а затем используем цикл `for` для добавления в нее песен и пробелов. Маленькая неприятность заключается в том, что мы добавили пробел в конец строки после последней песни, а это уже лишнее. Поэтому с помощью среза удалим последний пробел ❼.

Теперь можно отправлять код.

Прежде чем продолжить, можете попробовать решить упражнение 3, размещенное в конце главы.

Задача 10. Секретное предложение

Даже если у нас есть строка и мы знаем, сколько будет входных данных, цикл `while` по-прежнему может быть наилучшим вариантом. В этой задаче я покажу, почему так происходит.

Задача с сайта DMOJ, код `coci08c3p2`.

Постановка задачи

Лука на уроке пишет секретное предложение. Он хочет, чтобы учитель не смог его прочитать, поэтому вместо самого предложения записывает закодированную версию. После каждой гласной в предложении (*a*, *e*, *i*, *o* или *u*) он добавляет букву *p* и еще раз эту гласную. Например, вместо `I like you` получится `ipi lipikepe yopyupu`.

Учитель отбирает у Луки закодированное послание. Помогите ему понять, что там написано.

Входные данные

На вход приходит одна строка текста — закодированное Лукой предложение. Оно состоит из строчных букв и пробелов. Между словами стоит ровно один пробел. Максимальная длина строки — 100 символов.

Выходные данные

Исходное предложение, которое закодировал Лука.

Еще одно ограничение циклов for

В главе 3 мы узнали, как использовать циклы `for` для обработки строк. Цикл `for` проходит по строке от начала до конца, перебирая по одному символу за раз. Часто этого оказывается достаточно. Например, в задаче «Три чашки» нам нужно было просмотреть каждую смену позиции слева направо, поэтому мы применили цикл `for`.

В других случаях такой подход неудобен, и на выручку может прийти функция `range` , так как она дает доступ к индексам, а не к символам. А еще она позволяет пропускать последовательность с любым размером шага, который мы выберем. Например, мы можем использовать функцию `range` для просмотра каждого третьего символа строки:

```
>>> s = 'zephyr'
>>> for i in range(0, len(s), 3):
...     print(s[i])
...
z
h
```

Мы можем использовать этот цикл и для обработки строки справа налево:

```
>>> for i in range(len(s) - 1, -1, -1):
...     print(s[i])
...
r
y
h
p
e
z
```

Во всех этих примерах предполагается, что во время каждой итерации шаг фиксирован.

А что, если мы хотим перемещаться то на один, то на три символа? И так действительно бывает. Фактически, если бы мы могли реализовать это, задача была бы решена.

Чтобы понять, почему было бы так, рассмотрим тестовый пример:

```
ipi lipikepe yopoupu
```

Представьте, что мы реконструируем исходное предложение Луки, копируя в него символы. Первый символ в закодированном предложении — гласная `i`. Это также первый символ исходного предложения. Основываясь на том, как Лука кодирует предложение, мы знаем, что следующими двумя символами будут `p` и `i`. Нам не хочется включать их в исходное предложение, поэтому нужно их пропустить. То есть после обработки индекса `0` требуется индекс `3`.

Индекс `3` — это пробел. Поскольку это не гласная, мы копируем этот символ как есть, а затем переходим к индексу `4`. Индекс `4` — это `l`, согласная, поэтому копируем также `tt` и переходим к индексу `5`. Здесь уже гласная. После ее копирования мы хотим перейти к индексу `8`.

И как здесь задать размер шага? Иногда прыгаем на три символа, иногда на один. Циклы `for` для такого не годятся.

С помощью цикла `while` мы можем перемещаться по строке так, как нам угодно, не ограничиваясь значениями шага.

Перебор индексов циклом while

Написание цикла `while`, который перебирает строковые индексы, ничем не отличается от написания любого другого цикла `while`. Просто нужно указать длину строки. Мы можем перебирать каждый символ строки слева направо:

```
>>> s = 'zephyr'
>>> i = 0
❶ >>> while i < len(s):
...     print('We have ' + s[i])
...     i = i + 1
...
We have z
We have e
We have p
We have h
We have y
We have r
```

Переменная `i` позволяет обратиться к каждому символу строки. Значение `i` начинается с `0` и увеличивается на единицу за каждый проход цикла.

Я использовал в логическом выражении ❶ оператор `<`, чтобы цикл работал до тех пор, пока мы не достигнем конца строки. Если бы я поставил `<=` вместо `<`, мы бы получили `IndexError`:

```
>>> i = 0
>>> while i <= len(s):
...     print('We have ' + s[i])
...     i = i + 1
...
We have z
We have e
We have p
We have h
We have y
We have r
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: string index out of range
```

Длина строки равна 6. Мы получаем эту ошибку, потому что цикл пытается обратиться к символу `s[6]`, которого в строке нет.

А что, если мы хотим прыгать через три символа, а не через один? Без проблем — просто увеличьте `i` на 3 вместо 1:

```
>>> i = 0
>>> while i < len(s):
...     print('We have ' + s[i])
...     i = i + 3
...
We have z
We have h
```

Можно перебирать строку справа налево, а не слева направо. Мы должны начать с `len(s) - 1` вместо 0 и уменьшать `i` на каждой итерации. Да еще придется изменить логическое выражение цикла, чтобы поймать момент, когда мы окажемся в начале строки, а не в конце. Вот как перебрать строку посимвольно справа налево:

```
>>> i = len(s) - 1
>>> while i >= 0:
...     print('We have ' + s[i])
...     i = i - 1
...
We have r
We have y
We have h
We have p
We have e
We have z
```

Последний вариант использования цикла `while` — это остановка на некотором индексе, соответствующем некоторому критерию.

Стратегия заключается в применении логического оператора, согласно которому мы продолжаем проверки, если заданный критерий еще не выполнен. Например, можно найти индекс первого символа `y` в строке:

```
>>> i = 0
>>> while i < len(s) and s[i] != 'y':
...     i = i + 1
...
>>> print(i)
4
```

Если в строке нет символа `y`, цикл останавливается, когда переменная `i` дойдет до конца строки:

```
>>> s = 'breeze'
>>> i = 0
>>> while i < len(s) and s[i] != 'y':
...     i = i + 1
...
>>> print(i)
6
```

Когда переменной `i` присвоено значение `6`, первый операнд логического выражения становится равен `False`, поэтому цикл завершается. Вы можете спросить: почему второй операнд не вызывает ошибки, ведь индекса `6` в строке нет? Причина в том, что логические операторы вычисляются *слева направо с замыканием*, то есть операнды перестают вычисляться, если общий результат оператора уже известен. Если первый операнд равен `False`, то второй операнд уже не влияет на результат. Поэтому Python не вычисляет второй операнд. Точно так же происходит с оператором `or`: если первый операнд равен `True`, то `or` гарантированно вернет `True`, поэтому Python не вычисляет второй операнд.

Решение задачи

Теперь мы знаем, как использовать цикл `while` для перебора строки.

В нашей задаче нужно выполнять разные действия в зависимости от того, какая перед нами буква — гласная или согласная. Если гласная, то нужно скопировать символ и прыгнуть на три символа вперед (чтобы пропустить букву `p` и копию этой гласной). Если согласная или пробел, следует скопировать символ и перейти к следующему символу. То есть мы всегда копируем текущий символ, но затем перемещаемся на три или один символ вперед в зависимости от того, что собой представляет текущая буква. Мы можем добавить оператор `if` внутрь цикла `while`, чтобы принять это решение для каждого символа, который видим.

Решение задачи показано в листинге 4.4.

Листинг 4.4. Решение задачи

```
sentence = input()

❶ result = ''
i = 0

❷ while i < len(sentence):
      result = result + sentence[i]
    ❸ if sentence[i] in 'aeiou':
          i = i + 3
      else:
          i = i + 1

print(result)
```

Переменная `result` ❶ используется для посимвольного воссоздания исходного предложения.

Такое логическое выражение цикла `while` довольно стандартно для случаев, когда надо перебрать строку до конца ❷. В этом цикле мы сначала присоединяем текущий символ к концу строки `result`. Затем проверяем, является ли текущий символ гласной буквой ❸. В главе 2 мы говорили, что с помощью оператора `in` можно проверить, встречается ли первая строка во второй. Если текущий символ находится в строке гласных, мы прыгаем вперед на три символа, в противном случае переходим к следующему символу.

Цикл завершается, когда мы прошли по всему закодированному предложению и скопировали нужные символы в `result`. Осталось лишь вывести эту переменную.

Теперь можно отправить код на сайт. Отличная работа!

Операторы break и continue

В этом разделе я покажу вам два важных оператора Python, используемых внутри циклов, — `break` и `continue`. Знаю по собственному опыту: познакомившись с этими операторами, учащиеся часто злоупотребляют ими в ущерб ясности работы циклов, поэтому дальше по ходу книги их не будет. Тем не менее иногда они полезны, и вы наверняка увидите их в коде какого-нибудь проекта на Python, поэтому давайте узнаем, что тут к чему.

Оператор break

Ключевое слово `break` мгновенно и безусловно завершает цикл.

В решении задачи «Список воспроизведения» мы использовали цикл `while`, который работает, пока не будет нажата кнопка 4. Мы могли бы решить эту задачу и с помощью оператора `break` — код приведен в листинге 4.5.

Листинг 4.5. Решение задачи «Список воспроизведения» с помощью оператора break

```
songs = 'ABCDE'

❶ while True:
    button = int(input())
  ❷ if button == 4:
    ❸ break
    presses = int(input())
    for i in range(presses):
        if button == 1:
            songs = songs[1:] + songs[0]
```

```
        elif button == 2:
            songs = songs[-1] + songs[:-1]
        elif button == 3:
            songs = songs[1] + songs[0] + songs[2:]

output = ''
for song in songs:
    output = output + song + ' '

print(output[:-1])
```

Логическое выражение ❶ цикла выглядит подозрительно, так как `True` всегда равно `True`, поэтому кажется, что цикл вообще никогда не завершится (и это обратная сторона использования `break`, так как теперь нельзя, посмотрев на логическое выражение, узнать, в какой момент цикл должен завершиться). А завершится он в момент применения оператора `break`. Если нажата кнопка 4 ❷, мы вызываем этот оператор ❸, прерывая цикл.

Рассмотрим еще один пример использования `break`. Ранее в этой главе мы написали код, который определял индекс первой буквы `'y'`. Вот как он выглядит при использовании `break`:

```
>>> s = 'zephyr'
>>> i = 0
>>> while i < len(s):
...     if s[i] == 'y':
...         break
...     i = i + 1
...
>>> print(i)
4
```

И снова обратите внимание на то, что логическое выражение цикла несколько путает нас, так как предполагается, что цикл всегда выполняется до конца строки. Но при более внимательном анализе мы увидим, что оператор `break` в какой-то момент завершит цикл.

Оператор `break` завершает только свой собственный цикл, а внешние циклы не затрагивает. Пример:

```
>>> i = 0
>>> while i < 3:
...     j = 10
...     while j <= 50:
...         print(j)
...         if j == 30:
❶ ...             break
...         j = j + 10
```

```
...     i = i + 1
...
10
20
30
10
20
30
10
20
30
```

Обратите внимание на то, что оператор `break` ❶ прервал только цикл по `j`, не трогая цикл `i`. Его итерации закончились успешно ❶.

Оператор continue

Ключевое слово `continue` завершает текущую итерацию, не выполняя код в ней до конца. В отличие от `break` этот оператор не завершает цикл полностью. Если условие цикла истинно, то дальнейшие итерации цикла продолжают выполняться.

Далее приведен пример использования `continue` для вывода гласных и их индексов:

```
>>> s = 'zephyr'
>>> i = 0
>>> while i < len(s):
❶ ...     if not s[i] in 'aeiou':
...         i = i + 1
❷ ...         continue
❸ ...     print(s[i], i)
...     i = i + 1
...
e 1
```

Если текущий символ не является гласной буквой ❶, то выводить его не надо. Итак, мы увеличиваем `i` на 1, игнорируем ненужный символ, а затем используем `continue` ❷ для завершения текущей итерации. Если же попадаем в код после `if` ❸, это означает, что перед нами гласная (иначе сработало бы `continue`). Поэтому мы выводим этот символ и увеличиваем `i` на 1.

Ключевое слово `continue` выглядит заманчиво, потому что позволяет завершить отдельно взятую итерацию, которая нам не нужна, говоря: «Это не гласная, сматываем удочки!» Но того же поведения можно добиться и с помощью оператора `if`, а выглядит он яснее:

```
>>> s = 'zephyr'
>>> i = 0
```

```
>>> while i < len(s):
...     if s[i] in 'aeiou':
...         print(s[i], i)
...     i = i + 1
...
e 1
```

Вместо того чтобы пропускать итерацию, если текущий символ не гласная буква, оператор `if` делает то, что нужно, встретив гласную.

Резюме

Общая особенность задач, приведенных в этой главе, состоит в том, что мы не знаем заранее, сколько итераций цикла потребуется.

- **«Игровые автоматы»** — количество итераций зависит от введенного числа монет и выигрышей на игровых автоматах.
- **«Список воспроизведения»** — количество итераций зависит от нажатий на кнопки.
- **«Секретное предложение»** — количество итераций и действия на каждой из них зависят от того, где в строке расположены гласные.

Когда количество итераций неизвестно, используется цикл `while`, который будет работать столько, сколько необходимо. Применение `while` порождает больше ошибок, чем код с циклом `for`. Но в то же время он более гибкий, поскольку мы освобождаемся от ограничения цикла `for`, который вынуждает перебирать заданные заранее последовательности.

В следующей главе узнаем о списках, которые позволяют хранить большие объемы числовых или строковых данных. А как мы будем обрабатывать подобные данные? С помощью циклов, конечно же! Выполните приведенные далее упражнения, чтобы отточить навыки создания циклов. Мы будем часто пользоваться ими, решая задачи со списками.

Упражнения

Теперь в вашем распоряжении три типа циклов: `for`, `for-range` и `while`. Решение задач с циклами во многом заключается в определении того, какой цикл использовать! В следующих задачах поэкспериментируйте с различными типами циклов и найдите решение, которое вам больше нравится.

1. DMOJ, задача Epidemiology с кодом `ccc20j2`.
2. DMOJ, задача Ptice с кодом `coci08c1p2`.

3. DMOJ, задача AmeriCanadian с кодом `ccc02j2`.
4. DMOJ, задача Take a Number с кодом `ecoo13r1p1`.
5. DMOJ, задача When You Eat Your Smarties с кодом `ecoo15r1p1`.
6. DMOJ, задача Cold Compress с кодом `ccc19j3`.

Примечания

Задача «Игровые автоматы» взята с канадского компьютерного конкурса 2000 года, младший/старший уровень. Задача «Список воспроизведения» — с канадского компьютерного конкурса 2008 года, младший уровень. «Секретное предложение» — с хорватского конкурса по информатике 2008/2009 годов, часть 3.

5

Упорядоченные значения и списки

Вы видели, что для работы с последовательностями символов можно использовать строки. В этой главе поговорим о списках, которые помогают работать с последовательностями значений других типов, например целых или дробных чисел. Вы также узнаете, что можно вкладывать списки в списки, создавая двумерные наборы данных.

С помощью списков мы решим три задачи: найдем наименьшие окрестности поселений из заданного набора, определим, достаточно ли собрано денег на школьную поездку, и подсчитаем количество бонусных баллов в пекарне.

Задача 11. Деревни у дороги

В этой задаче нам нужно определить размер самых маленьких окрестностей деревень из заданного набора. Для этого нужно будет где-то хранить данные о размере всех окрестностей. Учитывая, что у нас может быть до 100 деревень, использование отдельной переменной для каждой из них выглядит нелепо. А вот списки позволяют собрать в одну коллекцию множество данных, которые не хочется представлять отдельными переменными. Мы также поговорим об операциях Python, которые позволяют выполнять изменение, поиск и сортировку списков.

Задача с сайта DMOJ с кодом `ccc18s1`.

Постановка задачи

Пусть есть n деревень, расположенных вдоль прямой дороги. Каждая деревня представлена целым числом, которое обозначает ее положение на дороге.

Слева от каждой деревни находится деревня с ближайшим меньшим номером, а справа — с ближайшим бо́льшим. *Окрестности* деревни определяются половинами расстояний между ней и ее левой и правой соседками. Например, если есть деревня на позиции 10 и у нее есть соседка слева на позиции 6 и соседка справа на позиции 15, то относящаяся к ней территория начинается с точки 8 (посередине между 6 и 10) и тянется до точки 12,5 (посередине между 10 и 15).

Крайняя левая и самая правая деревни имеют только одного соседа, поэтому определение окрестностей для них смысла не имеет и в этой задаче мы не будем принимать их во внимание.

Размер территории рассчитывается как разница между ее крайними точками. Например, территория, лежащая между точками 8 и 12,5, имеет размер 12,5 – 8 = 4,5.

Требуется определить размер самой маленькой территории.

Входные данные

Входные данные состоят:

- из строки, содержащей целое число n — количество деревень в диапазоне от 3 до 100;
- n строк, в которых указана позиция деревни у дороги. Каждая позиция представляет собой целое число от –1 000 000 000 до 1 000 000 000. Эти позиции не обязательно отсортированы, то есть соседка любой деревни может быть где угодно в списке.

Выходные данные

Размер наименьшей территории с точностью до одного знака после запятой.

Так зачем нам списки?

Для чтения входных данных нужно будет считать n целых чисел, которые обозначают позиции деревень. Подобное мы уже делали в задаче «Тарифный план» в главе 3. Тогда мы задействовали цикл `for-range`, чтобы выполнить цикл ровно n раз. Здесь поступим так же.

Но между задачами «Тарифный план» и «Деревни у дороги» есть огромная разница. В «Тарифном плане» мы считывали целое число, что-то с ним делали и больше никогда не использовали. Хранить его не требовалось. А в задаче «Деревни у дороги» недостаточно будет просмотреть числа лишь раз, ведь величина окрестностей

деревни зависит от ее левой и правой соседок, без которых мы не сможем рассчитать размер территории. Поэтому нужно хранить позиции всех деревень для дальнейшего применения.

Рассмотрим пример, показывающий, почему нам нужно хранить все позиции деревень. Исходные данные:

```
6
20
50
4
19
15
1
```

Тут у нас шесть деревень. Чтобы узнать размер окрестностей деревни, нам нужны соседки слева и справа от нее.

Первая деревня находится на позиции 20. И каков размер относящейся к ней территории? Чтобы ответить на этот вопрос, надо получить всю информацию о деревнях, иначе мы не определим ее левых и правых соседок. Зная позиции всех деревень, можно определить, что левая соседка находится на позиции 19, а правая — на позиции 50. Таким образом, размер окрестностей этой деревни равен (20 – 19) / 2 + + (50 – 20) / 2 = 15,5.

Вторая деревня находится на позиции 50. Каков размер ее окрестностей? Опять же нужно просмотреть весь список. Эта деревня оказывается самой правой, поэтому для нее мы ничего не считаем.

Третья деревня из входных данных располагается на позиции 4. Ее левая соседка — на позиции 1, а правая — на позиции 15, поэтому размер ее окрестностей равен (4 – 1) / 2 + (15 – 4) / 2 = 7.

Четвертая деревня на входе находится на позиции 19. Ее левая соседка расположена на позиции 15, а правая — на позиции 20, поэтому размер ее окрестностей равен (19 – 15) / 2 + (20 – 19) / 2 = 2,5.

Осталось рассмотреть деревню на позиции 15. Рассчитывая размер ее окрестностей, вы получите 7,5.

Сравнив все найденные размеры окрестностей, мы поймем, что самый маленький из них — 2,5.

Получается, нужно сохранить все позиции деревень, чтобы определить соседок каждой из них. Строки нас не устроят, ведь в них хранятся символы, а не числа. Тут нужны списки Python!

Списки

Список — это тип Python, в котором хранится некоторый набор значений (они называются *элементами* списка). Для определения списка используются открывающие и закрывающие квадратные скобки.

Если в строках хранятся только символы, то в списках можно хранить значения любого типа. В приведенном далее списке чисел хранятся позиции деревень из предыдущего примера.

```
>>> [20, 50, 4, 19, 15, 1]
[20, 50, 4, 19, 15, 1]
```

А вот список строк:

```
>>> ['one', 'two', 'hello']
['one', 'two', 'hello']
```

Мы даже можем поместить в список значения разных типов:

```
>>> ['hello', 50, 365.25]
['hello', 50, 365.25]
```

Многое из того, что вы делали со строками, работает и со списками. Например, списки поддерживают оператор конкатенации + и оператор репликации *:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1, 2, 3] * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Также есть оператор `in`, который сообщает, есть ли в списке некоторое значение:

```
>>> 'one' in ['one', 'two', 'hello']
True
>>> 'n' in ['one', 'two', 'three']
False
```

А еще есть функция `len`, которая возвращает длину списка:

```
>>> len(['one', 'two', 'hello'])
3
```

Поскольку список — это последовательность, то для просмотра его значений можем использовать цикл `for`:

```
>>> for value in [20, 50, 4, 19, 15, 1]:
...     print(value)
...
```

```
20
50
4
19
15
1
```

Списки, как и строки, целые числа и числа с плавающей точкой, можно хранить в переменных. Создадим две переменные со списками, а затем объединим их в новый список:

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [4, 5, 6]
>>> lst1 + lst2
[1, 2, 3, 4, 5, 6]
```

Мы вывели объединенный список, но никуда не сохранили его, а исходные списки не изменились:

```
>>> lst1
[1, 2, 3]
>>> lst2
[4, 5, 6]
```

Чтобы переменная ссылалась на объединенный список, используется присваивание:

```
>>> lst3 = lst1 + lst2
>>> lst3
[1, 2, 3, 4, 5, 6]
```

Имена наподобие `lst`, `lst1` и `lst2` можно применять, только если нет необходимости явно указывать на то, какие значения содержит список.

А вот само слово `list` в качестве имени переменной использовать нельзя. Это имя уже носит функция преобразования других коллекций в список:

```
>>> list('abcde')
['a', 'b', 'c', 'd', 'e']
```

Если вы создадите переменную с именем `list`, эта функция будет утрачена, а читающие код будут сбиты с толку.

Наконец, списки поддерживают индексацию и срезы. Обращение по индексу возвращает одно значение, а срез — список значений:

```
>>> lst = [50, 30, 81, 40]
>>> lst[1]
30
>>> lst[-2]
81
>>> lst[1:3]
[30, 81]
```

Если у вас есть список строк, то обратиться к одному из символов какой-то из них можно с помощью двух индексов — первый выбирает строку, а второй — символ в ней:

```
>>> lst = ['one', 'two', 'hello']
>>> lst[2]
'hello'
>>> lst[2][1]
'e'
```

ПРОВЕРИМ ЗНАНИЯ

Что окажется в переменной `total` после выполнения кода?

```
lst = [список чисел]
total = 0
i = 1

while i <= len(lst):
    total = total + i
    i = i + 1
```

А. Сумма элементов списка.

Б. Сумма элементов списка без его первого значения.

В. Сумма элементов списка, не включающая его первое и последнее значения.

Г. Этот код вызовет ошибку, потому что обращается к недопустимому индексу списка.

Д. Все ответы неверны.

Ответ: **Д**. Этот код складывает числа 1, 2, 3 и так далее до значения, равного длине списка. Числа из списка и индексы не используются вовсе!

Изменяемость списков

Строки *неизменяемы*, то есть их значения нельзя менять. Когда нам кажется, что мы изменяем строку (например, с помощью конкатенации строк), на самом деле мы создаем новую строку, а не изменяем уже существующую.

А вот списки *изменяемы*. Эта разница хорошо иллюстрируется индексацией. Если мы попытаемся изменить символ строки, получим ошибку:

```
>>> s = 'hello'
>>> s[0] = 'j'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

В сообщении об ошибке говорится, что строки не поддерживают присваивание элементов, то есть мы не можем изменять их символы.

Но поскольку списки изменяемы, их значения можно менять:

```
>>> lst = ['h', 'e', 'l', 'l', 'o']
>>> lst
['h', 'e', 'l', 'l', 'o']
>>> lst[0] = 'j'
>>> lst
['j', 'e', 'l', 'l', 'o']
>>> lst[2] = 'x'
>>> lst
['j', 'e', 'x', 'l', 'o']
```

Требуется хорошо понимать принципы работы оператора присваивания, иначе вас ждут сюрпризы. Пример:

```
  >>> x = [1, 2, 3, 4, 5]
❶ >>> y = x
  >>> x[0] = 99
  >>> x
  [99, 2, 3, 4, 5]
```

Пока все понятно. А теперь:

```
>>> y
[99, 2, 3, 4, 5]
```

Откуда в списке y оказалось число 99?

Когда выполняем присваивание `y = x` ❶, y начинает ссылаться на тот же список, что и x. Оператор присваивания не копирует список. Существует только один список, на который ссылаются два имени (или *псевдонима*). Поэтому, если мы внесем в список изменение, оно проявится при обращении по любому из этих имен.

Свойство изменяемости полезно, поскольку позволяет напрямую моделировать то, что можно делать со значениями в списке. Если мы хотим изменить значение, то просто меняем его. У неизменяемых типов изменить одно значение невозможно. Вместо этого пришлось бы создать новый список, который был бы таким же,

как и старый, но с одним измененным значением. Это сработает, но это окольный и менее прозрачный путь.

Если же вам действительно нужна копия списка, а не просто ссылка, используйте срезы. Опустите начальный и конечный индексы — и получите копию списка:

```
>>> x = [1, 2, 3, 4, 5]
>>> y = x[:]
>>> x[0] = 99
>>> x
[99, 2, 3, 4, 5]
>>> y
[1, 2, 3, 4, 5]
```

Обратите внимание на то, что теперь список y не изменился при изменении списка x. Получились отдельные списки.

ПРОВЕРИМ ЗНАНИЯ

Каким будет результат выполнения кода?

```
lst = ['abc', 'def', 'ghi']
lst[1] = 'wxyz'

print(len(lst))
```

А. 3

Б. 9

В. 10

Г. 4

Д. Код выдаст ошибку.

Ответ: **А.** Изменение значений списка разрешено, поскольку списки изменяемы. И изменение значения по индексу 1 на более длинную строку не меняет того факта, что в списке все так же три значения!

Введение в методы

Как и у строк, у списков много полезных методов. Некоторые из них мы увидим в следующем разделе, но сначала посмотрим, что вы можете сами узнать о том или ином методе.

Можете воспользоваться функцией `dir`, чтобы получить список методов для определенного типа. Просто вызовите функцию `dir` с некоторым значением — и получите все методы для этого типа.

Вот что дает функция `dir`, если передать ей строку:

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__',
<и другие операторы с подчеркиванием>
'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Обратите внимание на то, что мы вызвали `dir` с пустой строкой. Можно было бы использовать любое значение, но пустую строку ввести проще всего.

Имена с подчеркиванием нам неинтересны, так как они предназначены для внутренних операций Python и обычно не нужны. А вот остальные имена — это строковые методы, которые вы можете вызвать. В этом списке есть и уже известные вам строковые методы, такие как `isupper` и `count`, и неизвестные.

Узнать, как использовать определенный метод, можно с помощью имени этого метода и функции `help`. Вот справка по методу `count`:

```
>>> help(''.count)
Help on built-in function count:

count(...) method of builtins.str instance
 ❶ S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of
    substring sub in string S[start:end]. Optional
    arguments start and end are interpreted as in
    slice notation.
```

В справке говорится, как работать с методом ❶.

В квадратных скобках приведены необязательные аргументы. Вы можете использовать методы `start` и `end`, если хотите подсчитать вхождения `sub` в некоторую часть строки.

Иногда полезно просмотреть список методов, чтобы проверить, нет ли среди них такого, который помог бы решить вашу задачу. Даже если вы использовали

какой-либо метод раньше, в справке можете найти о нем такие подробности, о которых и не подозревали!

Чтобы посмотреть, какие методы имеются у списков, вызовите функцию `dir([])`. А чтобы почитать о них подробнее, вызовите `help([].`*xxx*`)`, где *xxx* — имя метода списка.

ПРОВЕРИМ ЗНАНИЯ

Справка метода строки `center`:

```
>>> help(''.center)
Help on built-in function center:

center(width, fillchar=' ', /) method of builtins.str instance
    Return a centered string of length width.

    Padding is done using the specified fill character
    (default is a space).
```

Что вы увидите после выполнения кода, показанного далее?

```
'cave'.center(8, 'x')
```

А. `'xxcavexx'`

Б. `'  cave  '`

В. `'xxxxcavexxxx'`

Г. `'    cave    '`

Ответ: **А**. Мы вызвали метод `center` с параметрами `width=8` и `fillchar='x'` (если бы указали только один аргумент, то вместо `fillchar` был бы подставлен пробел). Результирующая строка будет иметь длину 8. В строке `'cave'` четыре символа, поэтому нужны еще четыре символа, чтобы получить длину 8. Python добавляет два пробела в начале и два в конце, чтобы центрировать строку.

Методы списков

Вернемся к задаче «Деревни у дороги». Есть две операции над списками, которые помогут нам решить ее.

Во-первых, операция добавления в список. Вначале данных о позициях деревень у нас нет, и мы будем считывать их по одной из исходных данных. Поэтому нужен

способ добавлять вновь полученные значения в список: сначала в нем ничего не будет, затем появится одна деревня, потом две и т. д.

Во-вторых, сортировка списка. Когда все позиции деревень будут прочитаны, нам нужно будет определить самую маленькую территорию окрестностей. Для этого требуются позиция каждой деревни и расстояния до обеих ее соседок. Позиции в исходных данных могут приходить в любом порядке, поэтому найти соседку с ходу может оказаться не так-то просто. Вспомните работу, которую мы проделали ранее в этой главе. Чтобы найти соседей каждой деревни, нужно было всякий раз просмотреть весь список. Было бы намного проще, если бы деревни были отсортированы по позициям. Тогда бы мы точно знали, где находятся соседи: они будут слева и справа от деревни в списке.

Далее приведены позиции деревень в том порядке, в котором мы их читаем:

```
20 50 4 19 15 1
```

Полный бардак! Ведь на самом деле они должны стоять по порядку позиций, вот так:

```
1 4 15 19 20 50
```

И кто тогда соседи деревни на позиции 4? Просто смотрим налево и направо — это 1 и 15. А у деревни 15? Да вот же они — 4 и 19. И больше никаких долгих поисков. Поэтому нужно отсортировать список позиций, чтобы упростить код.

Элементы добавляются в список с помощью метода `append`, а сортируются — с помощью метода `sort`. Рассмотрим эти и несколько других методов, которые вы, вероятно, сочтете полезными для работы со списками, а затем вернемся к решению задачи.

Добавление в список

Метод `append` *добавляет* значение в список, а если говорить точнее, то в конец списка, после всех уже имеющихся значений. В приведенном далее коде мы добавляем три позиции деревень в изначально пустой список:

```
>>> positions = []
>>> positions.append(20)
>>> positions
[20]
>>> positions.append(50)
>>> positions
[20, 50]
>>> positions.append(4)
>>> positions
[20, 50, 4]
```

Обратите внимание, что мы используем метод `append` без оператора присваивания.

Метод append не возвращает список, а изменяет существующий.

Применение оператора присваивания с методами, которые изменяют список, — это распространенная ошибка, которая приводит к утрате списка, например:

```
>>> positions
[20, 50, 4]
>>> positions = positions.append(19)
>>> positions
```

Пусто! Переменная positions теперь ссылается на None, что и выводится на экран:

```
>>> print(positions)
None
```

Значение None означает отсутствие информации. Мы этого совершенно не ожидали, а хотели получить четыре деревни. Но сделали все неправильно и все испортили.

Если ваш список исчезает или вы получаете сообщения об ошибках, связанные со значением None, проверьте, не используется ли оператор присваивания с методом, который просто изменяет список.

Метод extend похож на метод append, но применяется в случаях, когда нужно добавить не одно значение, а целый список в конец существующего списка. Пример:

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [4, 5, 6]
>>> lst1.extend(lst2)
>>> lst1
[1, 2, 3, 4, 5, 6]
>>> lst2
[4, 5, 6]
```

Если вы хотите вставить значение в произвольное место списка, используйте метод insert. Этот метод принимает индекс и значение и вставляет его по заданному индексу:

```
>>> lst = [10, 20, 30, 40]
>>> lst.insert(1, 99)
>>> lst
[10, 99, 20, 30, 40]
```

Сортировка списков

Метод sort *сортирует* список, то есть упорядочивает его значения. Если мы вызовем его без аргументов, список будет отсортирован по возрастанию:

```
>>> positions = [20, 50, 4, 19, 15, 1]
>>> positions.sort()
>>> positions
[1, 4, 15, 19, 20, 50]
```

Если вызовем метод с аргументом `reverse=True`, список будет отсортирован по убыванию:

```
>>> positions.sort(reverse=True)
>>> positions
[50, 20, 19, 15, 4, 1]
```

Синтаксис передачи атрибутов `reverse=True` для нас в новинку. Ранее, вызывая методы и функции, мы обходились простым указанием значения, например, `True`. Однако метод `sort` несколько более требователен по причинам, которые я объясню в главе 6.

Удаление значений из списка

Метод `pop` удаляет значение по заданному индексу. Если аргумент не указан, метод `pop` удаляет и возвращает самое правое значение.

```
>>> lst = [50, 30, 81, 40]
>>> lst.pop()
40
```

Можно в качестве аргумента метода `pop` передать индекс. Например, в этом примере мы удаляем и возвращаем значение с индексом `0`:

```
>>> lst.pop(0)
50
```

Поскольку метод `pop` что-то возвращает, в отличие методов наподобие `append` и `sort`, имеет смысл присвоить его возвращаемое значение переменной:

```
>>> lst
[30, 81]
>>> value = lst.pop()
>>> value
81
>>> lst
[30]
```

Метод `remove` выполняет удаление по значению, а не по индексу. Передайте методу значение, которое нужно удалить, и он уберет из списка самое левое его вхождение. Если значение отсутствует, команда `remove` выдает ошибку. В примере далее два вхождения значения 50, поэтому метод `remove(50)` срабатывает дважды, а затем выдает:

```
>>> lst = [50, 30, 81, 40, 50]
>>> lst.remove(50)
```

```
>>> lst
[30, 81, 40, 50]
>>> lst.remove(50)
>>> lst
[30, 81, 40]
>>> lst.remove(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

ПРОВЕРИМ ЗНАНИЯ

Какое значение окажется в переменной `lst` после выполнения кода?

```
lst = [2, 4, 6, 8]
lst.remove(4)
lst.pop(2)
```

А. `[2, 4]`

Б. `[6, 8]`

В. `[2, 6]`

Г. `[2, 8]`

Д. Код выдаст ошибку.

Ответ: **В**. Метод `remove` удаляет значение 4, оставляя `[2, 6, 8]`. Затем метод `pop` удаляет значение по индексу 2, то есть 8. Остается `[2, 6]`.

Решение задачи

Предположим, мы успешно прочитали и отсортировали позиции деревень. Список в этот момент будет выглядеть так:

```
>>> positions = [1, 4, 15, 19, 20, 50]
>>> positions
[1, 4, 15, 19, 20, 50]
```

Чтобы найти размер наименьших окрестностей, сперва определим их размер для деревни с индексом 1 (обратите внимание на то, что мы не начинаем с индекса 0,

так как деревня с индексом 0 — самая левая и, согласно условиям задачи, можно ее игнорировать). Размер ее окрестностей определяется следующим образом:

```
>>> left = (positions[1] - positions[0]) / 2
>>> right = (positions[2] - positions[1]) / 2
>>> min_size = left + right
>>> min_size
7.0
```

В переменной `left` хранится размер левой части окрестностей, а в переменной `right` — размер правой части. Затем мы складываем их, чтобы получить общий размер. Получаем значение 7,0.

От этого значения будем отталкиваться. Как мы узнаем, есть ли у какой-либо другой деревни окрестности поменьше? С помощью цикла можем обработать другие деревни. Если найдем окрестности меньше, чем текущие наименьшие, то заменим их значение на вновь найденное.

Код решения находится в листинге 5.1.

Листинг 5.1. Решение задачи

```
n = int(input())

❶ positions = []

❷ for i in range(n):
    ❸ positions.append(int(input()))

❹ positions.sort()

❺ left = (positions[1] - positions[0]) / 2
  right = (positions[2] - positions[1]) / 2
  min_size = left + right

❻ for i in range(2, n - 1):
      left = (positions[i] - positions[i - 1]) / 2
      right = (positions[i + 1] - positions[i]) / 2
      size = left + right
    ❼ if size < min_size:
          min_size = size

print(min_size)
```

Сначала мы считываем из входных данных значение `n` — количество деревень. В переменную `positions` помещаем пустой список ❶.

На каждой итерации первого цикла `for` ❷ мы считываем одну позицию деревни и добавляем ее в список `positions`. Для этого с помощью функции `input` считываем позицию деревни, с помощью `int` преобразуем ее в целое число и добавляем его

в список с помощью метода append ❸. Приведенная строка ❸ будет эквивалентна вот этим трем строкам:

```
position = input()
position = int(position)
positions.append(position)
```

Считав позиции деревень, мы должны отсортировать их в порядке возрастания ❹. Затем определяем размер окрестностей деревни по индексу 1, сохраняя меньшее значение в переменной min_size ❺.

Затем во втором цикле мы перебираем все другие деревни, размеры окрестностей которых нужно вычислить ❻. Эти деревни начинаются с индекса 2 и заканчиваются индексом n - 2 (мы не хотим рассматривать деревню с индексом n - 1, потому что она крайняя правая). Поэтому используем функцию range с первым аргументом 2 (чтобы начать с 2) и вторым аргументом n - 1 (таким образом, заканчивая на n - 2).

Внутри цикла мы вычисляем размер окрестностей текущей деревни точно так же, как делали это для первой. Размер самой маленькой территории всегда находится в переменной min_size. А как узнать, будут ли вновь вычисленные окрестности меньше, чем минимальные из найденных ранее? Тут поможет оператор if ❼. Если размер окрестностей текущей деревни меньше min_size, мы помещаем его в min_size. Если окрестности этой деревни не меньше min_size, ничего не делаем, так как эта деревня нам неинтересна.

Перебрав все деревни, программа оставит в min_size размер наименьших окрестностей. Осталось вывести значение min_size.

В разделе «Выходные данные» этой задачи указано: «...с точностью до одного знака после запятой». А что если мы получим 6,25 или 8,33333? Разве мы не должны что-то с этим делать?

Нет. Все сработает и так, поскольку мы можем получить только размеры вроде 3,0 (с 0 после десятичной запятой) и 3,5 (с цифрой 5 после запятой). Вычисляя левую часть окрестностей, мы вычитаем одно целое число из другого и делим полученное целое число на 2. Если на 2 делится четное целое число, то результат получается с 0 после запятой (деление без остатка). А если делим нечетное, то получаем 5 после запятой. То же самое справедливо и для правой части окрестностей. Таким образом, складывая эти два значения, мы получим число, оканчивающееся только на 0 или 5.

Как избежать повторов кода: еще два решения

Не хотелось бы использовать код для вычисления размера окрестностей и перед циклом, и внутри него. Обычно наличие повторяющегося кода говорит о том, что код следует улучшить. Нам хочется избежать повторяющегося кода, потому что из-за этого его объем растет, а исправление проблем усложняется, если выясняется, что допущена ошибка именно в повторяющемся фрагменте. И хотя в данном случае

объем повторов выглядит приемлемым (всего три строчки), рассмотрим два способа уйти от этого. Мы разберем общие подходы, которые можно будет применять и к другим подобным проблемам.

Нужен размерчик побольше

Вычисление размера окрестностей деревни до цикла нужно лишь для того, чтобы у цикла было что сравнивать с соседями. Если мы войдем в цикл без значения `min_size`, то получим ошибку, когда код попытается сравнить его с размером окрестностей текущей деревни.

Если же мы перед циклом присвоим переменной `min_size` значение `0.0`, то цикл никогда не найдет меньшего размера и ответом задачи всегда будет 0,0. Значит, так тоже нельзя!

Зато можно задать большое значение, равное максимально возможному размеру окрестностей. Его нужно сделать таким огромным, чтобы на первой же итерации цикла гарантированно нашелся размер меньше начального. Тогда исходное большое значение не будет выведено никогда.

Из раздела «Входные данные» мы знаем, что все позиции находятся в диапазоне от –1 000 000 000 до 1 000 000 000. Таким образом, самая большая из возможных территория получается при наличии двух деревень в позициях –1 000 000 000 и 1 000 000 000 и третьей — между ними. У этой третьей деревни будут окрестности размером 1 000 000 000. Поэтому мы можем инициализировать переменную `min_size` значением `1000000000.0` или больше. Этот подход показан в листинге 5.2.

Листинг 5.2. Решение задачи с использованием большого начального значения

```
n = int(input())

positions = []

for i in range(n):
    positions.append(int(input()))

positions.sort()

min_size = 1000000000.0

❶ for i in range(1, n - 1):
    left = (positions[i] - positions[i - 1]) / 2
    right = (positions[i + 1] - positions[i]) / 2
    size = left + right
    if size < min_size:
        min_size = size

print(min_size)
```

Внимание! Теперь начинать вычисление размеров нужно будет с индекса `1` ❶, а не `2`, иначе мы забудем учесть окрестности деревни по индексу `1`.

Составление списка размеров

Еще один способ избежать дублирования кода — сохранить размеры всех окрестностей в отдельном списке. В Python есть встроенная функция `min`, которая принимает последовательность и возвращает ее минимальное значение:

```
>>> min('qwerty')
'e'
>>> min([15.5, 7.0, 2.5, 7.5])
2.5
```

(А еще есть функция `max`, которая возвращает максимум последовательности.)

В листинге 5.3 приведено решение, основанное на этом подходе.

Листинг 5.3. Решение задачи с использованием функции `min`

```
n = int(input())

positions = []

for i in range(n):
    positions.append(int(input()))

positions.sort()

sizes = []

for i in range(1, n - 1):
    left = (positions[i] - positions[i - 1]) / 2
    right = (positions[i + 1] - positions[i]) / 2
    size = left + right
    sizes.append(size)

min_size = min(sizes)
print(min_size)
```

Любое из этих решений можно отправить на сайт, выбирайте по вкусу!

Прежде чем продолжить, можете попробовать решить упражнение 1, приведенное в конце этой главы.

Задача 12. Студенческая поездка

Во многих задачах, с которыми вам придется столкнуться, в строках будут вводиться несколько целых или дробных чисел. До сих пор мы избегали этих проблем, но встречаются они постоянно! Пора узнать, как использовать списки для обработки подобного ввода.

Задача с сайта DMOJ, код `ecoo17r1p1`.

Постановка задачи

Студенты хотят в конце года отправиться в поездку, но им нужны деньги, чтобы ее оплатить. Для этого они организовали второй завтрак. Чтобы посетить его, студент первого курса платит 12 долларов, второго — 10 долларов, третьего — 7 долларов, а четверокурсник — 5 долларов.

Из денег, собранных на этих завтраках, 50 % можно использовать для оплаты поездки, а остальные 50 % идут на оплату самого завтрака.

Нам известны стоимость поездки, доля студентов на каждом курсе и общее количество студентов. Определите, нужно ли студентам собрать еще средства.

Входные данные

Входные данные состоят из десяти тестовых примеров, по три строки на пример (всего 30 строк). Формат строк каждого тестового примера:

- первая строка содержит стоимость поездки в долларах — целое число от 50 до 50 000;
- вторая строка содержит четыре числа, обозначающих доли учащихся первого, второго, третьего и четвертого курсов соответственно. Между числами есть пробелы. Каждое число лежит в диапазоне от 0 до 1, а их сумма равна 1 (чтобы получить 100 %);
- в третьей строке записано целое число n — общее количество студентов на завтраке — в диапазоне от 4 до 2000.

Выходные данные

Если учащимся нужно собрать больше денег на поездку, выведите `YES`, в противном случае — `NO`.

Маленькая хитрость

Предположим, что у нас 50 студентов и 10 % из них (доля равна 0,1) учатся на четвертом курсе. Получается, что количество четверокурсников составляет $50 \cdot 0{,}1 = 5$.

А если студентов 50, но на четвертом курсе 15 % из них (доля 0,15), то при умножении получим $50 \cdot 0{,}15 = 7{,}5$ студента.

Получается полтора землекопа, и мы не знаем, что делать в этом случае. В полном описании задачи указано, что мы должны округлить такую дробь в меньшую сторону, поэтому округлим ее до 7. Но тогда сумма студентов первого, второго,

третьего и четвертого курсов окажется не равна общему количеству студентов. Этих неучтенных студентов нужно будет добавить к самому многочисленному курсу. Гарантируется, что такой курс будет всего один.

Сперва решим задачу, игнорируя это обстоятельство. А затем обработаем и его, получив полное решение.

Разделение строк и объединение списков

Вторая строка каждого тестового примера состоит из четырех чисел, например:

```
0.2 0.08 0.4 0.32
```

Нам нужен способ извлечь эти четыре числа из строки для дальнейшей обработки. Изучим метод разделения строки на список ее составных частей. А заодно поговорим о методе соединения строк, который позволяет выполнить обратную задачу — свернуть список в строку.

Преобразование строки в список

Напомню, что функция `input` возвращает строку независимо от того, как выглядит ввод. Если ввод интерпретируется как целое число, нужно преобразовать строку в целое число. Если ввод следует интерпретировать как число с плавающей точкой, требуется преобразовать строку в число с плавающей точкой. А что, если ввод нужно интерпретировать как четыре числа? Тогда придется разделить его на отдельные числа, а затем уже выполнять преобразования!

Метод `split` разбивает строку на список, состоящий из ее частей. По умолчанию метод `split` выполняет разбиение по пробелам, что и нужно в данном случае:

```
>>> s = '0.2 0.08 0.4 0.32'
>>> s.split()
['0.2', '0.08', '0.4', '0.32']
```

Метод `split` возвращает список строк, после чего мы сможем обращаться к каждой из них уже независимо. Я сохраняю список, который вернул метод `split`, а затем обрабатываю два его значения:

```
>>> proportions = s.split()
>>> proportions
['0.2', '0.08', '0.4', '0.32']
>>> proportions[1]
'0.08'
>>> proportions[2]
'0.4'
```

Реальные данные часто разделяются запятыми, а не пробелами. Все проще простого: можно вызвать `split` с аргументом, в котором указать, каким будет разделитель:

```
>>> info = 'Toronto,Ontario,Canada'
>>> info.split(',')
['Toronto', 'Ontario', 'Canada']
```

Объединение списка в строку

Чтобы выполнить обратную задачу, а именно получить из списка строку, можно применить метод `join`. Строка, на которой вызван метод, служит разделителем между значениями списка. Два примера:

```
>>> lst = ['Toronto', 'Ontario', 'Canada']
>>> ','.join(lst)
'Toronto,Ontario,Canada'
>>> '**'.join(lst)
'Toronto**Ontario**Canada'
```

Технически `join` можно применять для объединения значений в любой последовательности, а не только в списках. Вот пример объединения символов из строки:

```
>>> '*'.join('abcd')
'a*b*c*d'
```

Изменение значений списка

Используя метод `split` на строке из четырех частей, мы получаем список строк:

```
>>> s = '0.2 0.08 0.4 0.32'
>>> proportions = s.split()
>>> proportions
['0.2', '0.08', '0.4', '0.32']
```

В главе 1 вы узнали, что строки, даже если в них содержатся числа, нельзя задействовать в числовых вычислениях. Значит, нужно преобразовать этот список строк в список дробных чисел.

Преобразовать строку в число с плавающей точкой можно с помощью функции `float`:

```
>>> float('45.6')
45.6
```

Но это всего одно число. А как преобразовать весь список строк в список чисел с плавающей точкой? Интуитивно кажется, что можно использовать такой цикл:

```
>>> for value in proportions:
...     value = float(value)
```

Логика заключается в том, что цикл берет каждое значение в списке и превращает его в дробное число.

К сожалению, так это не работает. Список по-прежнему состоит из строк:

```
>>> proportions
['0.2', '0.08', '0.4', '0.32']
```

А почему? Функция `float` не работает? Да нет, с ней все в порядке:

```
>>> for value in proportions:
...     value = float(value)
...     type(value)
...
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
```

Четыре числа — все верно! Но в списке все равно находятся строки.

Все дело в том, что мы не меняем значения, которые хранятся в списке. Мы меняем то, на что ссылается переменная `value`, но на значения в списке это не влияет. Чтобы действительно изменить значения в списке, нужно назначить новые значения по индексам. Вот как это сделать:

```
>>> proportions
['0.2', '0.08', '0.4', '0.32']
>>> for i in range(len(proportions)):
...     proportions[i] = float(proportions[i])
...
>>> proportions
[0.2, 0.08, 0.4, 0.32]
```

Цикл теперь проходит по индексам, выполняет присваивание, и значения меняются.

Решение задачи (большей ее части)

Теперь мы способны решить задачу без учета той самой ловушки.

Начнем с примера, чтобы пояснить, что должен делать код. Затем перейдем к самому коду.

Тестовый пример

Входные данные для этой задачи состоят из десяти тестовых примеров, но здесь рассмотрим только один. Если вы наберете этот пример с клавиатуры, то увидите ответ. Но программа на этом не завершится, потому что ждет следующего тестового примера. Использовав перенаправление ввода, вы снова увидите ответ, но получите ошибку `EOFError`. `EOF` означает «конец файла», а ошибка вызвана тем, что программа пытается прочитать больше входных данных, чем их находится в файле. После того как ваш код заработает для одного тестового примера, можете попробовать добавить к нему другие примеры, чтобы убедиться, что они тоже работают. Набрав десять примеров, вы доведете программу до полноценной работы.

Рассмотрим вместе тестовый пример:

```
504
0.2 0.08 0.4 0.32
125
```

Студенческая поездка стоит 504 доллара, на завтрак пришли 125 студентов.

Чтобы определить, сколько денег собрано, надо подсчитать, сколько денег получено от студентов каждого курса. У нас $125 \cdot 0{,}2 = 25$ студентов первого курса, и каждый из них платит по 12 долларов. То есть первокурсники собрали $25 \cdot 12 = 300$ долларов. Аналогичным образом можем рассчитать собранные остальными курсами суммы. Результат сведен в табл. 5.1.

Таблица 5.1. Пример задачи

Курс	Студентов на курсе	Цена для одного студента	Денег собрано
Первый	25	12	300
Второй	10	10	100
Третий	50	7	350
Четвертый	40	5	200

Деньги, собранные студентами каждого курса, рассчитываются умножением количества студентов этого курса на стоимость завтрака для одного человека. Общая сумма, собранная всеми студентами, составляет $300 + 100 + 350 + 200 = 950$ долларов. Но лишь 50 % этой суммы можно использовать на поездку. Итак, у нас осталось $950 / 2 = 475$ долларов, а этого недостаточно, чтобы заплатить за поездку стоимостью 504 доллара. Таким образом, правильный ответ — `YES`, потому что студентам нужно продолжать сбор денег.

Код

Приведенное далее частичное решение будет правильно обрабатывать любой ввод, где умножение доли на количество студентов дает целое число студентов, как в приведенном ранее примере. См. листинг 5.4.

Листинг 5.4. Большая часть решения задачи «Студенческая поездка»

```
❶ YEAR_COSTS = [12, 10, 7, 5]

❷ for dataset in range(10):
      trip_cost = int(input())
    ❸ proportions = input().split()
      num_students = int(input())

    ❹ for i in range(len(proportions)):
          proportions[i] = float(proportions[i])

    ❺ students_per_year = []

      for proportion in proportions:
       ❻ students = int(num_students * proportion)
          students_per_year.append(students)

      total_raised = 0

    ❼ for i in range(len(students_per_year)):
          total_raised = total_raised + students_per_year[i] * YEAR_COSTS[i]

    ❽ if total_raised / 2 < trip_cost:
          print('YES')
      else:
          print('NO')
```

Сначала используем переменную `YEAR_COSTS`, в которой в виде списка хранятся цены для студентов первого, второго, третьего и четвертого курсов ❶. Определив количества студентов на курсах, мы умножим их на эти значения, чтобы найти собранную сумму. Стоимость никогда не меняется, поэтому этот список можно считать константой, а имена констант в Python принято писать прописными буквами, как я и сделал.

Входные данные содержат десять тестовых примеров, поэтому мы выполняем внешний цикл десять раз ❷. по одному разу для каждого примера. Остальная часть программы находится внутри этого цикла.

В каждом тестовом примере читаем три строки ввода. Вторая строка содержит четыре доли, поэтому мы используем `split`, чтобы разбить ее на список из четырех

строк ❸. Затем применяем цикл `range for` для преобразования каждой из них в число с плавающей точкой ❹.

Следующая задача — определить количество студентов на каждом курсе. Мы начинаем с пустого списка ❺. Затем берем каждую долю, умножаем ее на общее количество студентов ❻ и добавляем результат в список. Обратите внимание, что я взял функцию `int`, чтобы гарантированно добавлять только целые числа. При использовании чисел с плавающей точкой `int` отбрасывает дробную часть, округляя их в меньшую сторону.

Теперь у нас есть два списка, которые нужны, чтобы подсчитать, сколько денег было собрано. В списке `student_per_year` находятся количества студентов на курсах, выглядит он примерно так:

```
[25, 10, 50, 40]
```

А в `YEAR_COSTS` содержатся цены для студентов, опять же по курсам:

```
[12, 10, 7, 5]
```

Значения по индексу `0` в этих списках относятся к студентам первого курса, с индексом `1` — второго и т. д. Такие списки называются *параллельными*, потому что они работают параллельно, вместе давая больше информации, чем каждый в отдельности.

Эти два списка используются для расчета общей суммы собранных денег. Для этого умножаем количество студентов на соответствующую стоимость на одного человека и складываем полученные произведения ❼.

Достаточно ли денег на поездку? Чтобы выяснить это, используем оператор `if` ❽. На поездку можно истратить половину денег, собранных во время завтрака. Если эта сумма меньше стоимости поездки, то нужно собрать больше денег (`YES`), в противном случае — нет (`NO`).

Код, который мы написали, очень общий. Лишь указание количества курсов ❶ несет конкретную информацию. Если бы мы хотели решить аналогичную задачу для другого количества курсов, достаточно было бы изменить эту строку и передать нужные входные данные. В этом сила списков: они помогают писать гибкий код и легко вносить изменения в решаемые задачи.

Теперь про хитрость

Давайте посмотрим, почему наша программа иногда работает неправильно и как это исправить.

Рассмотрим пример

Далее приведен тестовый пример, на котором этот код не сработает:

```
50
0.7 0.1 0.1 0.1
9
```

На этот раз поездка стоит 50 долларов и на завтрак приходят девять студентов. Количество студентов на первом курсе равно 9 · 0,7 = 6,3, что округляется до 6. Именно округление в меньшую сторону является причиной, по которой надо быть повнимательнее. Результаты работы для всех курсов приведены в табл. 5.2.

Таблица 5.2. Пример, на котором программа не сработает

Курс	Студентов на курсе	Цена для 1 студента	Денег собрано
Первый	6	12	72
Второй	0	10	0
Третий	0	7	0
Четвертый	0	5	0

На всех курсах, кроме первого, обучается 0 студентов, потому что 9 · 0,1 = 0,9, что округляется до 0. Собрано всего 72 доллара. Половина от 72 долларов — это 36 долларов, и их недостаточно для оплаты поездки за 50 долларов. То есть программа должна вывести `YES`, и нам нужно собрать больше денег...

...или нет. Стоп, ведь должно быть девять студентов, а не шесть! Мы потеряли трех человек из-за округления. В описании задачи говорится, что мы должны добавить недостающих студентов к курсу с наибольшим их числом. Если мы сделаем это, то собранная сумма станет 9 · 12 = 108 долларов. Половина от 108 долларов — это 54 доллара, поэтому для поездки за 50 долларов денег достаточно! Правильный вывод — `NO`.

Другие операции над списками

Чтобы исправить программу, нам нужно сделать две вещи: выяснить, сколько студентов потерялось из-за округления, и добавить их к курсу с наибольшим числом человек.

Суммирование списка

Чтобы определить количество потеряшек, мы можем сложить числа в списке `student_per_year`, а затем вычесть эту сумму из общего числа студентов. Функция `sum` принимает список и возвращает сумму его значений:

```
>>> students_per_year = [6, 0, 0, 0]
>>> sum(students_per_year)
6
>>> students_per_year = [25, 10, 50, 40]
>>> sum(students_per_year)
125
```

Нахождение индекса максимума

Функция `max` принимает последовательность и возвращает ее максимальное значение:

```
>>> students_per_year = [6, 0, 0, 0]
>>> max(students_per_year)
6
>>> students_per_year = [25, 10, 50, 40]
>>> max(students_per_year)
50
```

Но нам нужен не сам максимум, а индекс максимума, чтобы по нему можно было увеличить количество студентов. Зная максимальное значение, мы можем найти его индекс с помощью метода `index`. Он возвращает крайний левый индекс, по которому найдено указанное значение, или выводит ошибку, если значения нет в списке:

```
>>> students_per_year = [6, 0, 0, 0]
>>> students_per_year.index(6)
0
>>> students_per_year.index(0)
1
>>> students_per_year.index(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 50 is not in list
```

Но мы ищем значение, которое точно есть в списке, так что эта ошибка не возникнет.

Решение задачи

Готово! Осталось дополнить наше частичное решение, чтобы обработать все возможные случаи. Новая программа приведена в листинге 5.5.

Листинг 5.5. Решение задачи «Студенческая поездка»

```
YEAR_COSTS = [12, 10, 7, 5]

for dataset in range(10):
    trip_cost = int(input())
    proportions = input().split()
    num_students = int(input())

    for i in range(len(proportions)):
        proportions[i] = float(proportions[i])

    students_per_year = []

    for proportion in proportions:
        students = int(num_students * proportion)
        students_per_year.append(students)

❶ counted = sum(students_per_year)
    uncounted = num_students - counted
    most = max(students_per_year)
    where = students_per_year.index(most)
❷ students_per_year[where] = students_per_year[where] + uncounted

    total_raised = 0

    for i in range(len(students_per_year)):
        total_raised = total_raised + students_per_year[i] * YEAR_COSTS[i]

    if total_raised / 2 < trip_cost:
        print('YES')
    else:
        print('NO')
```

Появилось пять новых строк, начиная с ❶. Воспользуемся функцией `sum`, чтобы подсчитать, сколько студентов у нас вышло, а затем вычитаем сумму из общего количества студентов, чтобы получить число потеряшек. Затем применяем функции `max` и `index`, чтобы определить индекс курса, в который нужно добавить неучтенных студентов. Наконец, добавляем неучтенных студентов в этот индекс ❷ (добавление 0 к числу не меняет это число, поэтому код безопасен для случая, если потеряшек не будет).

Вот и все. Бегом на сайт! А потом вернитесь, так как пора изучить еще более общие структуры списков.

Прежде чем продолжить, можете попробовать решить упражнение 5, приведенное в конце главы.

Задача 13. Бонус «Бейкера»

В этой задаче мы увидим, как с помощью списков можно работать с двумерными данными. Они часто используются в реальных программах. Например, данные в электронной таблице состоят из строк и столбцов, именно для этих случаев нужны методы, которые мы сейчас рассмотрим.

Задача с сайта DMOJ, код `ecoo17r3p1`.

Постановка задачи

У кондитерской Baker Brie есть несколько франшиз, каждая из которых продает потребителям выпечку. Чтобы отметить 13 лет работы, руководитель выдает франшизам награду, начисляя бонусы в зависимости от объема продаж. Бонусы зависят от продаж в день и продаж на франшизу. Система работы бонусов такова.

- За каждый день, когда общий объем продаж всех франшиз кратен 13, кратный множитель будет выдан в качестве бонуса. Например, за день, когда франшизы продали в сумме 26 хлебобулочных изделий, они получат 26 / 13 = 2 бонуса.
- Каждой франшизе, чей общий объем продаж за все дни кратен 13, кратный множитель будет выдан в качестве бонуса. Например, продав в сумме 39 хлебобулочных изделий, вы получите 39 / 13 = 3 бонуса.

Определите общее количество бонусов.

Входные данные

Входные данные состоят из десяти тестовых примеров. Каждый пример состоит:

- из строки, содержащей количество франшиз f и целое число дней d, разделенные пробелом. f лежит в диапазоне от 4 до 130, а d — от 2 до 4745;
- d строк, по одной на день, в которых содержится f целых чисел, разделенных пробелами. Каждое целое число определяет количество продаж. В первой строке приведены продажи для каждой франшизы в первый день, во второй — продажи франшизы во второй день и т. д. Все целые числа должны быть от 1 до 13 000.

Выходные данные

Для каждого теста выведите общее количество начисленных бонусов.

Табличные данные

Данные для этой задачи можно представить в виде таблицы. Начнем с примера, а затем посмотрим, как представить таблицу в виде списка.

Тестовый пример

Если у нас есть d дней и f франшиз, мы можем представить данные в виде таблицы с d строк и f столбцов.

Вот пример тестового случая:

```
6 4
1 13 2 1 1 8
2 12 10 5 11 4
39 6 13 52 3 3
15 8 6 2 7 14
```

Этому примеру соответствует табл. 5.3.

Таблица 5.3. Бонусный стол «Бейкера»

	0	1	2	3	4	5
0	1	13	2	1	1	8
1	2	12	10	5	11	4
2	39	6	13	52	3	3
3	15	8	6	2	7	14

Я пронумеровал строки и столбцы, начиная с 0, чтобы они совпадали с тем, как данные будут храниться в списке.

Сколько бонусов будет начислено в этом примере? Сначала посмотрим на строки таблицы, которые соответствуют дням. Сумма продаж для строки 0 равна 1 + 13 + + 2 + 1 + 1 + 8 = 26. Поскольку 26 делится на 13, эта строка дает нам 26 / 13 = 2 бонуса. Сумма в строке 1 равна 44. Она не кратна 13, поэтому бонусов здесь нет. Сумма строки 2 равна 116 — опять без бонусов. Сумма строки 3 равна 52, что дает 52 / 13 = 4 бонуса.

Теперь посмотрим на столбцы, которые соответствуют франшизам. Сумма в столбце 0 равна 1 + 2 + 39 + 15 = 57. Не кратно 13, поэтому никаких бонусов. Единственный столбец, который дает бонусы, — это столбец 1. Сумма в нем составляет 39, что дает 39 / 13 = 3 бонуса.

Общее количество выданных бонусов составляет 2 + 4 + 3 = 9. Это и есть ответ для данного тестового примера.

Вложенные списки

Мы уже видели списки целых чисел, чисел с плавающей точкой и строк. А также можем создавать списки списков, то есть *вложенные списки*. Каждое значение такого списка само по себе является списком. Обычно для ссылки на вложенный список используется имя переменной, например «сетка» или «таблица». Далее приведен список Python, соответствующий табл. 5.3:

```
>>> grid = [[ 1, 13,  2,  1,  1,  8],
...         [ 2, 12, 10,  5, 11,  4],
...         [39,  6, 13, 52,  3,  3],
...         [15,  8,  6,  2,  7, 14]]
```

Каждое значение списка соответствует одной строке. Использовав индекс, мы получим строку, которая сама по себе является списком:

```
>>> grid[0]
[1, 13, 2, 1, 1, 8]
>>> grid[2]
[39, 6, 13, 52, 3, 3]
```

Использовав два индекса, получим одно значение. Например, значение в строке 1, столбце 2:

```
>>> grid[1][2]
10
```

Работа со столбцами устроена немного сложнее, чем со строками, потому что каждый из них разнесен на несколько списков. Чтобы получить доступ к столбцу, нужно собрать по одному значению из каждой строки. Мы можем сделать это с помощью цикла, который постепенно создает новый список, соответствующий столбцу. Например, составим столбец 1:

```
>>> column = []
>>> for i in range(len(grid)):
❶ ...     column.append(grid[i][1])
...
>>> column
[13, 12, 6, 8]
```

Обратите внимание на то, что первый индекс (номер строки) меняется, а второй (столбец) не меняется ❶. То есть берутся значения из одного столбца.

А как насчет суммирования строк и столбцов? Суммировать строку мы можем с помощью функции `sum`. Например, сумма строки 0:

```
>>> sum(grid[0])
26
```

Мы также можем использовать цикл, например:

```
>>> total = 0
>>> for value in grid[0]:
...     total = total + value
...
>>> total
26
```

Использование функции `sum` — более простой способ, поэтому на нем и остановимся.

Чтобы просуммировать столбец, мы можем построить список столбцов и применить к нему функцию `sum`, а можем вычислить сумму напрямую, не создавая новый список. Второй вариант для столбца 1:

```
>>> total = 0
>>> for i in range(len(grid)):
...     total = total + grid[i][1]
...
>>> total
39
```

ПРОВЕРИМ ЗНАНИЯ

Каким будет результат выполнения кода:

```
lst = [[1, 1],
       [2, 3, 4]]
x = 0

for i in range(len(lst)):
    for j in range(len(lst[0])):
        x = x + lst[i][j]

print(x)
```

А. 2

Б. 7

В. 11

Г. Код выдаст ошибку (неверный индекс).

Ответ: **Б**. Переменная `i` перебирает значения `0` и `1`, поскольку список `lst` имеет длину 2, переменная `j` — тоже, поскольку и `lst[0]` имеет длину 2. Значения списка складываются, если находятся на индексах 0 и 1, то есть `lst[1][2]` в сумму не входит.

ПРОВЕРИМ ЗНАНИЯ

В этом коде дважды вызывается функция `print`:

```
lst = [[5, 10], [15, 20]]
x = lst[0]
x[0] = 99
print(lst)

lst = [[5, 10], [15, 20]]
y = lst[0]
y = y + [99]
print(lst)
```

Что будет выведено?

А.
```
[[99, 10], [15, 20]]
[[5, 10], [15, 20]]
```

Б.
```
[[99, 10], [15, 20]]
[[5, 10, 99], [15, 20]]
```

В.
```
[[5, 10], [15, 20]]
[[5, 10], [15, 20]]
```

Г.
```
[[5, 10], [15, 20]]
[[5, 10, 99], [15, 20]]
```

Ответ: **А.** Переменная `x` ссылается на первую строку списка `lst`, то есть `lst[0]`, поэтому `x[0] = 99` влияет на первую строку и ее первый элемент.

Переменная `y` тоже ссылается на первую строку, но присваивание выполняется в саму переменную `y`, не затрагивая список.

Решение задачи

Код для решения этой задачи приведен в листинге 5.6.

Листинг 5.6. Решение задачи «Бонус "Бейкера"»

```
for dataset in range(10):
❶   lst = input().split()
    franchisees = int(lst[0])
    days = int(lst[1])
    grid = []
```

```
❷ for i in range(days):
      row = input().split()
    ❸ for j in range(franchisees):
          row[j] = int(row[j])
    ❹ grid.append(row)

  bonuses = 0

❺ for row in grid:
    ❻ total = sum(row)
      if total % 13 == 0:
          bonuses = bonuses + total // 13

❼ for col_index in range(franchisees):
      total = 0
    ❽ for row_index in range(days):
          total = total + grid[row_index][col_index]
      if total % 13 == 0:
          bonuses = bonuses + total // 13

  print(bonuses)
```

Как и в случае с задачей «Студенческая поездка», входные данные содержат десять тестовых примеров, поэтому мы помещаем весь код внутрь цикла, который повторяется десять раз.

В каждом тестовом примере мы читаем первую строку ввода и вызываем метод `split`, чтобы превратить ее в список ❶. Он будет содержать два значения — количество франшиз и количество дней, поэтому преобразуем их в целые числа и присваиваем значения нужным переменным.

Переменная `grid` начинается с пустого списка. Позже в ней появится список строк, где каждая строка представляет собой список продаж за данный день.

С помощью цикла `for-range` выполним перебор по дням ❷. Считав строку из ввода, вызываем метод `split`, чтобы превратить ее в список значений продаж. Эти значения в данный момент являются строками, поэтому мы используем вложенный цикл для преобразования их в целые числа ❸. Затем добавляем строку в сетку ❹.

Входные данные прочитаны, таблица готова. Пора считать бонусы. Мы делаем это в два этапа: сначала по строкам, затем по столбцам.

Чтобы посчитать бонусы по строкам, задействуем цикл `for` на переменной `grid` ❺. Как и любой цикл `for`, он перебирает значения в списке по одному за раз. Здесь каждое значение представляет собой список, поэтому в переменную `row` попадают списки. Функция `sum` работает с любым списком чисел, поэтому с ее помощью мы складываем числа в строке ❻. Если сумма делится на 13, то складываем количество бонусов.

Перебирать столбцы списка так же, как строки, не получится, поэтому нам придется прибегнуть к циклическому просмотру индексов. Для этого снова возьмем цикл `for` и переберем индексы столбцов ❼. Для суммирования текущего столбца функция `sum` не подойдет, поэтому понадобится вложенный цикл. Он проходит по строкам ❽, складывая значения в нужном столбце. Затем мы проверяем, делится ли эта сумма на 13, и добавляем бонусы, если да.

В конце выводим общее количество бонусов.

Пора на сайт! Отправив код, вы увидите, что все тестовые примеры будут засчитаны.

Резюме

В этой главе мы узнали о списках, которые помогают работать с коллекциями значений любого типа. Списки чисел, списки строк, списки списков — Python поддерживает все, что нам нужно. Мы также узнали о методах списков и о том, почему сортировка упрощает обработку значений.

В отличие от строк списки изменяемы, то есть мы можем изменять их содержимое. Это помогает нам легче манипулировать списками, но следует быть внимательными и понимать, когда изменение проходит, а когда — нет.

Настал тот момент обучения, когда мы можем писать программы с большим количеством строк кода. Мы можем управлять потоком выполнения программы, используя операторы условия и циклы. Мы можем хранить информацию и управлять ею с помощью строк и списков. Мы можем писать программы для решения сложных задач. Такие программы может быть трудно разрабатывать и читать. К счастью, есть инструмент, который позволяет организовать программу, упрощая ее код, его мы изучим в следующей главе. Выполнение некоторых из следующих упражнений поможет вам попрактиковаться в написании большого объема кода. Решив их, вы будете готовы продолжать!

Упражнения

Вот несколько упражнений, которые вам стоит выполнить.

1. DMOJ, задача Deal or No Real Calculator с кодом `ccc07j3`.
2. DMOJ, задача Cezar с кодом `coci17c1p1`.
3. DMOJ, задача Preokret с кодом `coci18c2p1`.
4. DMOJ, задача Babbling Brooks с кодом `ccc00s2` (вам пригодится функция `round`).
5. DMOJ, задача Willow's Wide Ride с кодом `ecoo18r1p1`.

6. DMOJ, задача Free Shirts с кодом `ecoo19r1p1`.
7. DMOJ, задача Tides с кодом `dmopc14c7p2`.
8. DMOJ, задача Wesley Plays DDR с кодом `wac3p3`.
9. DMOJ, задача Rue's Rings с кодом `ecoo18r1p2`. (Если будете использовать форматированные строки, вам понадобится способ включить сами символы { и }. Для этого применяются двойные скобки `{{}}`.)
10. DMOJ, задача Emacs с кодом `coci19c5p1`.
11. DMOJ, задача Crtanje с кодом `coci20c2p1`. (Вам нужны будут строки от –100 до 100. А как хранить строки с отрицательным индексом, когда списки Python начинаются с индекса `0`? Увеличьте все индексы на 100 и заведите список от `0` до `200`. Тут есть еще одно небольшое неудобство со строками: символ `'\'` особенный и, чтобы его вывести, придется писать `'\\'`.)
12. DMOJ, задача Charlie's Crazy Conquest с кодом `dmopc19c5p2`. (Здесь повнимательнее с индексами и правилами игры!)

Примечания

Задача «Деревни у дороги» взята с канадского компьютерного конкурса 2018 года, старший уровень. «Студенческая поездка» взята с конкурса по программированию Образовательной вычислительной организации Онтарио 2017 года, этап 1. «Бонус "Бейкера"» — с конкурса по программированию Образовательной вычислительной организации Онтарио 2017 года, этап 3.

6

Пишем собственные функции

При написании больших программ важно разбивать код на малые логические фрагменты, каждый из которых выполняет свою задачу в рамках общей сверхзадачи. Это позволяет нам заниматься каждой частью функционала отдельно, не думая о том, что делают другие части. А затем эти части объединяются. И вот эти-то самые логические части называются *функциями*.

В этой главе мы будем использовать функции для разделения кода на фрагменты и для решения двух задач: подсчитаем очки в карточной игре для двух игроков и попробуем определить, можно ли красиво украсить коробки с фигурками.

Задача 14. Карточная игра

В этой задаче реализуем карточную игру для двух игроков. Решая задачу, мы обнаружим, что одна и та же логика возникает несколько раз. Рассмотрим, как объединить этот код в функцию Python, чтобы избежать его дублирования и сделать более ясным.

Задача с сайта DMOJ, код `ccc99s1`.

Постановка задачи

Два игрока, A и B, играют в карточную игру (отмечу, что опыт игры в карты нам для решения этой задачи не потребуется).

В игре используется колода из 52 карт. Игрок A берет из нее карту, потом игрок B берет карту, затем снова игрок A, вновь игрок B, и так продолжается, пока в колоде не останется карт.

В колоде 13 типов карт, а именно: двойка, тройка, четверка, пятерка, шестерка, семерка, восьмерка, девятка, десятка, валет, дама, король, туз. Имеются по четыре карты каждого из этих типов, то есть четыре двойки, четыре тройки и так далее, вплоть до четырех тузов (поэтому в колоде 52 карты: 13 типов по 4 карты каждого типа).

Старшими картами будут называться валет, дама, король и туз.

Когда игрок берет старшую карту, он может заработать очки.

Правила начисления очков таковы.

- Если игрок берет валета, при этом в колоде остается хотя бы одна карта и следующая карта не является старшей, игрок получает 1 очко.
- Если игрок берет даму, при этом в колоде остается как минимум две карты и ни одна из следующих двух карт в колоде не является старшей, игрок получает 2 очка.
- Если игрок берет короля, при этом в колоде остается не менее трех карт и ни одна из следующих трех карт в колоде не является старшей, игрок получает 3 очка.
- Если игрок берет туза, при этом в колоде остается не менее четырех карт и ни одна из следующих четырех карт в колоде не является старшей, игрок получает 4 очка.

Нам требуется выводить информацию каждый раз, когда игрок получает очки, а в конце игры вывести общий счет каждого игрока.

Входные данные

Входные данные состоят из 52 строк. В каждой из них указан тип карты. Строки расположены в том порядке, в котором карты берут из колоды, то есть первая строка — это первая карта, взятая из колоды, вторая строка — вторая карта и т. д.

Выходные данные

Всякий раз, когда игрок зарабатывает очки, выводите следующую строку:

```
Player p scores q point(s).
```

Здесь вместо *p* следует подставить букву `A` для игрока A или букву `B` для игрока B, а вместо *q* — количество очков, заработанное данным ходом.

Когда игра закончится, выведите следующие две строки:

```
Player A: m point(s).
Player B: n point(s).
```

где *m* — общий счет игрока A, *n* — общий счет игрока B.

Тестовый пример

Если вы подумаете над решением этой задачи, у вас может возникнуть вопрос: «Можем ли мы решить ее прямо сейчас, не изучая ничего нового?» Да, можем! Мы вполне готовы. Можно взять список и с его помощью смоделировать колоду карт. Мы знаем, как использовать метод `append`, который позволяет добавить карту в колоду. Можем просмотреть значения в списке, чтобы понять, если ли в нужном месте старшие карты. У нас даже есть форматированные строки, которые позволяют выводить информацию об игроках и их очках.

Но хватит подробностей, лучше рассмотрим небольшой пример. Из него станет ясно, что нам не хватает одной важной функции Python, которая позволила бы упростить организацию решения задачи.

Если в примере будет 52 карты, то разбирать его мы будем до Нового года, поэтому возьмем колоду всего из десяти карт. Это неполный тестовый пример, поэтому код, который мы напишем позже, на нем не сработает, но и этого примера будет достаточно, чтобы понять особенности игры и суть ее решения. Тестовый пример:

```
queen
three
seven
king
nine
jack
eight
king
jack
four
```

Игрок A берет первую карту — даму. Это старшая карта, и игрок A может получить 2 очка. Но сначала надо проверить, что в колоде после нее осталось не менее двух карт. Далее мы должны проверить следующие две карты в надежде, что среди них нет старшей. Следующие две карты не являются старшими — это тройка и семерка, поэтому игрок A получает 2 очка.

Игрок B берет вторую карту — тройку. Она не старшая, поэтому игрок B не получает очков.

Игрок A берет семерку. Ноль очков.

Теперь игрок B берет короля, поэтому у него есть шанс получить 3 очка. После этого короля в колоде осталось не менее трех карт. Мы должны проверить следующие три карты на предмет, нет ли среди них старшей. К сожалению, среди них есть старшая карта, валет. Игрок B ничего не получает.

Игрок A берет девятку. Нет очков.

Игрок B берет первого валета. После него в колоде осталась хотя бы одна карта. Мы должны проверить следующую карту, надеясь, что она не старшая. Да, это не старшая карта, а восьмерка, поэтому игрок B получает 1 очко.

В колоде осталось лишь одно очко, и его получает игрок A, когда берет из колоды предпоследнюю карту — валета.

Таким образом, результат тестового примера выходит вот таким:

```
Player A scores 2 point(s).
Player B scores 1 point(s).
Player A scores 1 point(s).
Player A: 3 point(s).
Player B: 1 point(s).
```

Обратите внимание на то, что каждый раз, когда игрок берет старшую карту, нужно проверить две вещи: что в колоде осталось определенное количество карт и среди ближайших карт нет старшей. Первое можно проверить с помощью переменной, которая сообщает, сколько карт к этому моменту взято. А вот со вторым сложнее. Нужно будет проверить, нет ли среди заданного количества карт старшей. Хуже того, если мы не проявим осторожность, то получится, что некоторый код у нас повторится четыре раза: один раз, чтобы проверить карту после валета, один раз, чтобы проверить две карты после дамы, один раз, чтобы проверить три карты после короля, и один раз, чтобы проверить четыре карты после туза. Если же мы в какой-то момент обнаружим ошибку в логике, исправлять ее придется тоже в четырех разных местах.

Есть в Python что-нибудь, что позволило бы нам один раз упаковать логику «здесь нет старших карт», а затем вызвать ее четыре раза? Есть. Это называется *функцией*. А функция — это просто именованный блок кода, выполняющий некую небольшую задачу. Функции необходимы для организации кода и придания ему ясности. Ими пользуются все программисты. Без них создание крупных программных систем вроде игр или текстовых процессоров было бы невозможно. Давайте узнаем, как работать с функциями.

Определение и вызов функций

В принципе, мы уже умеем вызывать функции, встроенные в Python. Например, мы использовали функцию `input` для чтения входных данных. Вот так можно вызвать ее без аргументов:

```
>>> s = input()
hello
>>> s
'hello'
```

Еще мы применяли функцию `print` для вывода текста. Пример вызова функции `print` с одним аргументом:

```
>>> print('well, well')
well, well
```

Встроенные функции Python универсальны и предназначены для использования в самых разных условиях. Но если нам нужна функция, которая решала бы какую-то конкретную задачу, придется определить ее самостоятельно.

Функции без аргументов

Чтобы *определить* или создать функцию, применяется ключевое слово Python `def`. Далее приведена функция, которая выводит три строки:

```
>>> def intro():
...     print('*********')
...     print('*WELCOME*')
...     print('*********')
...
```

Структура определения функции похожа на структуру оператора `if` или цикла. После ключевого слова `def` идет имя определяемой нами функции, в данном случае `intro`. После имени функции стоит пара пустых скобок `()`. Позже мы увидим, что в них заключается информация для будущей передачи аргументов функциям.

Но эта функция пока что не принимает никаких аргументов, поэтому скобки пусты. После скобок стоит двоеточие, и, как и в случае с операторами `if` или циклами, его отсутствие — это синтаксическая ошибка. В последующих строках находится блок операторов, записанных с отступом, который будет запускаться при каждом вызове функции.

Определив функцию `intro`, вы, скорее всего, ожидали увидеть такой вывод:

```
*********
*WELCOME*
*********
```

А вот и нет, ведь пока мы лишь определили функцию, но еще не вызвали ее. Определение функции само по себе не дает результата, так как это просто сохранение функции в памяти компьютера, чтобы можно было вызвать ее позже. Вызывать свои функции можно точно так же, как и любые встроенные функции Python. Поскольку нашей функции никакие аргументы не нужны, при вызове пишем пустой набор круглых скобок:

```
>>> intro()
*********
*WELCOME*
*********
```

Вызывать эту функцию можно сколько угодно раз. Хоть в каждой строке, если надо.

Функции с аргументами

Наша функция `intro` не слишком гибкая, поскольку при каждом вызове она делает одно и то же. Мы можем немного переписать эту функцию, чтобы можно было передавать ей аргументы, и аргументы эти будут влиять на то, что и как она делает. Далее приведена новая версия функции `intro`, которая позволяет нам передавать ей один аргумент:

```
>>> def intro2(message):
...     line_length = len(message) + 2
...     print('*' * line_length)
...     print(f'*{message}*')
...     print('*' * line_length)
...
```

Чтобы вызвать эту функцию, передайте ей строковый аргумент:

```
>>> intro2('HELLO')
*******
*HELLO*
*******
>>> intro2('WIN')
*****
*WIN*
*****
```

Мы не можем вызвать функцию `intro2` без аргумента, а если все же попробуем, то получим ошибку:

```
>>> intro2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: intro2() missing 1 required positional argument: 'message'
```

Текст ошибки говорит нам, что мы не передали функции аргумент `message`. Эта самая переменная `message` называется *параметром* функции. Когда мы вызываем функцию `intro2`, Python сначала присваивает переменной `message` то, что мы передали в аргументе, и дальше к значению аргумента можно обратиться по имени `message`.

Можно также создавать функции с несколькими параметрами. Далее приведена функция, которая принимает два параметра — выводимое сообщение и количество его повторений:

```
>>> def intro3(message, num_times):
...     for i in range(num_times):
...         print(message)
...
```

Чтобы вызвать эту функцию, нужно передать ей два аргумента. Python выполняет код слева направо, присваивая первый аргумент первому параметру, а второй аргумент — второму параметру. В следующем вызове слово `'high'` попадает в параметр `message`, а число `5` — в параметр `num_times`:

```
>>> intro3('high', 5)
high
high
high
high
high
```

Важно передать функции правильное количество аргументов. Функции `intro3` нужно уже два аргумента, а иначе получится ошибка:

```
>>> intro3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: intro3() missing 2 required positional arguments: 'message'
and 'num_times'
>>> intro3('high')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: intro3() missing 1 required positional argument: 'num_times'
```

Кроме того, передаваемые аргументы должны быть правильных типов. Передача данных неправильного типа не помешает нам вызвать функцию, но внутри нее возникнет ошибка:

```
>>> intro3('high', 'low')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in intro3
TypeError: 'str' object cannot be interpreted as an integer
```

Исключение `TypeError` возникает оттого, что в функции `intro3` используется цикл `for-range` по переменной `num_times`. Если аргумент, который мы передадим в `num_times`, не является целым числом, цикл не сработает.

Именованные аргументы

Чтобы не приходилось помнить и соблюдать порядок аргументов слева направо при вызове функции, можно явно указывать имена параметров и передавать их в любом порядке. Аргумент, передаваемый по имени параметра, называется *именованным аргументом*. Вот как это работает:

```
>>> def intro3(message, num_times):
...     for i in range(num_times):
...         print(message)
...
>>> intro3(message='high', num_times=3)
high
high
high
>>> intro3(num_times=3, message='high')
high
high
high
```

В каждом вызове функции используются два именованных аргумента. Пишется это через имя параметра, знак равенства и значение аргумента.

Можно в начале функции передавать позиционные (обычные) аргументы, а затем именованные:

```
>>> intro3('high', num_times=3)
high
high
high
```

А вот после именованного аргумента вызывать позиционные уже нельзя:

```
>>> intro3(message='high', 3)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

В главе 5, говоря о сортировке списков, мы использовали именованный аргумент `reverse` при вызове метода `sort`. Разработчики Python решили, что параметр `reverse` должен быть именованным, что означает: задать его значение без явного указания имени нельзя. Python позволяет делать это и с нашими функциями, но в этой книге такой уровень контроля не понадобится.

Локальные переменные

Переданные функции параметры работают как обычные переменные, но они являются *локальными* для функции, в которой определены. То есть параметр функции не существует за ее пределами:

```
>>> def intro2(message):
...     line_length = len(message) + 2
...     print('*' * line_length)
...     print(f'*{message}*')
...     print('*' * line_length)
...
>>> intro2('hello')
*******
*hello*
*******
>>> message
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'message' is not defined
```

А что насчет переменной `line_length` — она тоже локальная? (Спойлер: да.)

```
>>> line_length
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'line_length' is not defined
```

Что произойдет, если у вас есть некоторая переменная и вы вызываете функцию, у которой есть параметр или локальная переменная с тем же именем? Внешняя переменная потеряется? Посмотрим:

```
>>> line_length = 999
>>> intro2('hello')
*******
*hello*
*******
>>> line_length
999
```

Значение `999` никуда не делось и осталось в том виде, в каком мы его оставили. Локальные переменные создаются при вызове функции и уничтожаются при ее завершении, не влияя на другие переменные с теми же именами.

Функция может обращаться к переменным, созданным вне ее. Но использовать такой механизм не рекомендуется, потому что тогда эта функция не будет самодостаточной и в работе будет полагаться на существование каких-то посторонних переменных. В книге мы будем писать функции так, чтобы они задействовали только локальные переменные. Вся информация, которая нужна функции, будет передаваться ей через параметры.

Изменяемые параметры

Поскольку параметр является псевдонимом для соответствующего аргумента, его можно использовать для изменения изменяемого значения. Приведенная далее функция удаляет все вхождения `value` из списка `lst`:

```
>>> def remove_all(lst, value):
...     while value in lst:
...         lst.remove(value)
...
>>> lst = [5, 10, 20, 5, 45, 5, 9]
>>> remove_all(lst, 5)
>>> lst
[10, 20, 45, 9]
```

Обратите внимание на то, что мы передали список в функцию `remove_all` с помощью переменной. Эта функция окажется бесполезной, если вы вызовете ее напрямую со значением списка, а не с переменной, ссылающейся на список:

```
>>> remove_all([5, 10, 20, 5, 45, 5, 9], 5)
```

Функция удалила все пятерки из списка, но у нас нет возможности снова обратиться к нему, поскольку мы не использовали переменных.

ПРОВЕРИМ ЗНАНИЯ

Какой результат даст следующий код?

```
def mystery(s, lst):
    s = s.upper()
    lst = lst + [2]

s = 'a'
lst = [1]
mystery(s, lst)

print(s, lst)
```

А. `a[1]`

Б. `a[1, 2]`

В. `A[1]`

Г. `A[1, 2]`

Ответ: **A**. Когда вызывается функция `mystery`, ее параметр `s` начинает ссылаться на любой аргумент, в данном случае строку `'a'`. Аналогично параметр `lst` ссылается на любой переданный аргумент, в данном случае на список `[1]`. Внутри функции `mystery` переменные `s` и `lst` являются локальными.

Теперь давайте изучим содержимое функции.

Первый оператор `s = s.upper()`. Строка в переменной `s` переводится в верхний регистр, превращаясь в `'A'`. Но это не изменило значения `s` вне функции. В ней все еще находится значение `'a'` (строка).

Второй оператор `lst = lst + [2]`. Оператор `+`, применяемый со списками, создает новый список (не меняет список!), поэтому локальная переменная `lst` теперь ссылается на новый список `[1, 2]`. Но опять же это не изменило внешней переменной `lst`, в которой все то же значение `[1]`.

Но разве я не говорил ранее, что функции могут изменять изменяемый параметр? Ну да, говорил, но для этого вам действительно нужно изменить само значение, а не переопределить локальную переменную.

Сравните предыдущую программу со следующей, вывод которой будет уже другим:

```
def mystery(s, lst):
    s.upper()     # метод upper создает новую строку
    lst.append(2) # метод append изменяет сам список
s = 'a'
lst = [1]
mystery(s, lst)

print(s, lst)
```

Возвращаемые значения

Вернемся к задаче с карточной игрой. Наша цель — определить функцию, которая сообщает, нет ли в некотором списке старших карт. Назовем эту функцию `no_high`. Мы еще не написали ее, но можем обозначить, какого результата хотим достичь. Вот что нам нужно:

```
>>> no_high(['two', 'six'])
True
>>> no_high(['eight'])
True
>>> no_high(['two', 'jack', 'four'])
False
>>> no_high(['queen', 'king', 'three', 'queen'])
False
```

Мы хотим, чтобы первые два вызова возвращали `True`, потому что в этих списках нет старших карт. Аналогично третий и четвертый вызовы должны возвращать `False`, потому что в этих списках карт есть по крайней мере одна старшая карта.

Как же определить функцию, которая возвращает значения `True` и `False`? Это последний кусок головоломки.

Чтобы вернуть значение из функции, используем ключевое слово Python `return`. Как только код доходит до него, выполнение функции прекращается и указанное значение возвращается вызывающей стороне.

Вот как мы можем написать функцию `no_high`:

```
>>> def no_high(lst):
...     if 'jack' in lst:
...         return False
...     if 'queen' in lst:
...         return False
...     if 'king' in lst:
...         return False
...     if 'ace' in lst:
...         return False
...     return True
...
```

Сначала проверим, есть ли в списке валеты. Если есть, то список уже точно содержит одну или несколько старших карт, поэтому можно сразу возвращать `False`.

Если мы все еще здесь, значит, валетов нет. Но могут быть другие старшие карты, поэтому нужно проверить и их тоже. Остальные операторы `if` проверяют наличие дам, королей и тузов, возвращая `False`, если какая-либо из этих карт есть в списке.

Если мы не попали ни в один из четырех `return`, это означает, что старших карт в списке нет. В этом случае возвращаем `True`.

Сам по себе оператор `return` без заданного значения возвращает значение `None`. Это полезно, если вы пишете функцию, которая не возвращает ничего полезного и нужно завершить функцию, не выполняя оставшийся в ней код.

Если оператор `return` встречается внутри цикла, функция все равно немедленно завершается независимо от того, насколько глубоко он вложен. Далее приведен пример, иллюстрирующий возврат из вложенного цикла:

```
>>> def func():
...     for i in range(10):
...         for j in range(10):
...             print(i, j)
...             if j == 4:
...                 return
...
>>> func()
0 0
```

```
0 1
0 2
0 3
0 4
```

`Return` — это как `break`, только круче! Некоторым программистам не нравится использовать возврат из цикла по той же причине, по которой не нравится и `break`: он мешает понять, что цикл делает. Задействовать `return` внутри цикла можно в любой момент, когда потребуется. В отличие от `break`, который может находиться где угодно, `return` можно использовать только внутри функции, изолированной от другого кода. Если функции, которые мы пишем, будут небольшими, то применение `return` в цикле поможет писать четкий код, не мешающий коду вокруг.

ПРОВЕРИМ ЗНАНИЯ

Правильно ли написана функция `no_high`? Возвращает ли она `True`, если в списке есть хоть одна старшая карта, и `False` в противном случае?

```
def no_high(lst):
    for card in lst:
        if card in ['jack', 'queen', 'king', 'ace']:
            return False
        else:
            return True
```

А. Да.

Б. Нет, например, будет выведен неправильный ответ для списка `['two', 'three']`.

В. Нет, например, будет выведен неправильный ответ для списка `['jack']`.

Г. Нет, например, будет выведен неправильный ответ для списка `['jack', 'two']`.

Д. Нет, например, будет выведен неправильный ответ для списка `['two', 'jack']`.

Ответ: **Д**. Оператор `if-else` всегда прерывает цикл на первой итерации. Если в начале находится старшая карта, функция прерывается и возвращает `False`. Если первая карта не старшая, функция прерывается и возвращает `True`. В обоих случаях две другие карты не рассматриваются.

Поэтому код не сработает на списке `['two', 'jack']`: первая карта не является старшей, и функция возвращает `True`. А значение `True` говорит нам, что старших карт в списке не осталось, но это не так: в списке есть валет. Функция сработала неверно и должна была вернуть `False`.

Документация по функциям

Теперь нам ясно, что делает функция `no_high` и когда ее надо вызывать. А что будет через несколько месяцев, когда вспомнить назначение старого кода окажется трудновато? А может, у нас будет большая коллекция собственных функций и станет трудно запоминать, что делает каждая из них?

К каждой функции, которую мы пишем, следует добавлять документацию, в которой нужно описать значение каждого параметра и то, что функция возвращает. Такая документация называется *строкой документации*. Ее следует использовать с первой строки блока функции. Далее приведена функция `no_high`, но теперь с документацией:

```
>>> def no_high(lst):
...     """
...     lst — это список строк, соответствующих картам.
...
...     Возвращает True, если в lst нет старших карт, иначе False.
...     """
...     if 'jack' in lst:
...         return False
...     if 'queen' in lst:
...         return False
...     if 'king' in lst:
...         return False
...     if 'ace' in lst:
...         return False
...     return True
...
```

Строка документации начинается и заканчивается тремя двойными кавычками (`"""`). Как и одинарная (`'`) или двойная кавычка (`"`), три двойные кавычки позволяют обозначать начало и конец любой строки. Строка, созданная с указанием трех кавычек, называется *строкой в тройных кавычках* (можно использовать и три одинарные кавычки, но по соглашению Python применяются именно двойные). Тройные кавычки позволяют писать многострочный комментарий, просто нажимая Enter после каждой строки, чего нельзя достичь одинарными кавычками. Тройные кавычки хорошо подходят для строк документации, так как позволяют включить в нее сколько угодно строк.

Строка документации в данной функции говорит нам, что `lst` — это список строк, обозначающих карты. Также она сообщает, что функция возвращает значение `True` или `False` и что именно означает каждое из них. Этой информации достаточно, чтобы другой пользователь мог вызвать функцию, не глядя на ее код. Если человек знает, что делает функция, он может просто брать ее и работать с ней. Мы ведь и раньше применяли функции Python, не глядя в их код. Как действует функция

print? А функция input? Мы не знаем! Это не имеет значения, так как нам известно, что делают функции, и мы можем просто вызывать их, не вдаваясь в детали.

Для функций с несколькими параметрами в строке документации должны быть перечислены все параметры с указанием ожидаемого типа. Далее приведена функция remove_all из этой главы с подходящей строкой документации:

```
>>> def remove_all(lst, value):
...     """
...     lst – это список.
...     value – это значение.
...
...     Удаляет все вхождения value из lst.
...     """
...     while value in lst:
...         lst.remove(value)
...
```

Обратите внимание на то, что в этой строке документации ничего не говорится о возвращаемом значении. Дело в том, что функция и не возвращает ничего полезного! Она удаляет значение из списка lst, как, собственно, и сказано в документации.

Решение задачи

Мы изучили основы определения и вызова функций. В оставшейся части книги всякий раз, сталкиваясь с большой задачей, сможем разбить ее решение на более мелкие подзадачи, каждая из которых будет решаться функцией.

Воспользуемся функцией no_high для решения задачи «Карточная игра». Код решения приведен в листинге 6.1.

Листинг 6.1. Решение задачи «Карточная игра»

```
❶ NUM_CARDS = 52

❷ def no_high(lst):
      """
      lst – это список строк, соответствующих картам.
...
      Возвращает True, если в lst нет старших карт,
      иначе – False.
      """
      if 'jack' in lst:
          return False
      if 'queen' in lst:
          return False
      if 'king' in lst:
          return False
```

```
    if 'ace' in lst:
        return False
    return True

❸ deck = []

❹ for i in range(NUM_CARDS):
    deck.append(input())

score_a = 0
score_b = 0
player = 'A'

❺ for i in range(NUM_CARDS):
    card = deck[i]
    points = 0
  ❻ remaining = NUM_CARDS - i - 1
  ❼ if card == 'jack' and remaining >= 1 and no_high(deck[i+1:i+2]):
        points = 1
    elif card == 'queen' and remaining >= 2 and no_high(deck[i+1:i+3]):
        points = 2
    elif card == 'king' and remaining >= 3 and no_high(deck[i+1:i+4]):
        points = 3
    elif card == 'ace' and remaining >= 4 and no_high(deck[i+1:i+5]):
        points = 4

  ❽ if points > 0:
        print(f'Player {player} scores {points} point(s).')
  ❾ if player == 'A':
        score_a = score_a + points
        player = 'B'
    else:
        score_b = score_b + points
        player = 'A'

print(f'Player A: {score_a} point(s).')
print(f'Player B: {score_b} point(s).')
```

Я ввел константу NUM_CARDS и присвоил ей число 52 ❶. Мы будем использовать его несколько раз в коде, и легче запомнить, что означает NUM_CARDS, чем понять, что значит 52.

Затем определяем функцию no_high, не забыв добавить в нее строку документации ❷, которую обсудили ранее. Мы всегда будем определять функции в начале программы, чтобы они были доступны для вызова любому последующему коду.

Основная часть программы начинается с создания списка, в котором будет храниться колода ❸. Затем мы читаем карты из входных данных ❹, добавляя каждую карту в колоду. Обратите внимание, что карты никогда в буквальном смысле не удаляются и не берутся из колоды (она остается неизменной на протяжении

всего выполнения программы). Мы могли бы реализовать и удаление, но я решил отслеживать, где мы находимся в колоде, чтобы знать, какая карта будет удалена следующей.

Дальше нужны еще три важные переменные: `score_a` — общий счет игрока A, `score_b` — общий счет игрока B и `player` — имя действующего игрока.

Теперь нам нужно просмотреть каждую карту в колоде и начислить игрокам их законные очки. Обычный цикл `for` позволит нам увидеть текущую карту. Но этого недостаточно: если данная карта является старшей, нам придется просматривать и несколько последующих карт. Чтобы облегчить задачу, воспользуемся функцией `range` цикла `for` ❺.

На каждой итерации этого цикла мы определяем количество очков, присуждаемых активному игроку, в зависимости от карты, которую он берет из колоды. Правило получения очков зависит от количества оставшихся в колоде карт. Переменная `remaining` ❻ сообщает, сколько карт осталось. Когда переменная `i` равна `0`, остается 51 карта, потому что мы только что взяли первую карту. Когда `i` равна `1`, количество оставшихся карт — 50, потому что мы только что взяли вторую карту. Получается, для подсчета количества оставшихся карт нужно взять их общее количество, вычесть `i` и вычесть 1.

Теперь у нас есть четыре проверки ❼, по одной на каждый способ получения очков. Во всех четырех случаях мы проверяем текущую карту и количество оставшихся карт. Если оба условия истинны, то вызывается функция `no_high` с частью колоды, содержащей соответствующее количество карт. Например, если текущая карта — валет и осталась хотя бы одна карта, то мы передаем в `no_high` список длиной `1` ❼. Если `no_high` возвращает `True`, то старших карт в рассмотренном фрагменте нет и текущий игрок получает очки. Переменная `points` определяет количество очков. На каждой итерации цикла она изначально равна `0`, а затем принимает значение `1`, `2`, `3` или `4` в зависимости от ситуации.

Если игрок набрал 0 очков ❽, то выводим сообщение с именем игрока, получившего очки, и их количеством.

На текущей итерации остается лишь добавить очки к счету активного игрока и передать ход другому игроку. Мы выполняем обе задачи с помощью оператора `if-else` ❾ (если на данной итерации количество очков остается равным 0, то к счету игрока добавляется безобидный 0 и избавляться от этого прибавления смысла нет).

Последние два вызова `print` выводят общее количество очков для каждого игрока.

Вот и все: мы решили задачу, воспользовавшись функцией для организации кода и повышения удобочитаемости. Можете отправить код на сайт — там вы должны увидеть, что все тестовые примеры пройдены.

Задача 15. Фигурки

В предыдущей задаче мы, еще не приступив к решению, сначала рассмотрели пример, и он показал, в какой части кода было бы полезно использовать функцию. Теперь решим другую задачу с помощью функций, но определим, где они нужны, применив более системный подход.

Задача с сайта Timus, код 2144. Это единственная в книге задача с этого ресурса. Чтобы найти ее, перейдите на сайт https://acm.timus.ru/, нажмите Problem set, затем Volume 12 и найдите задачу 2144 (она называется Cleaning the Room).

Постановка задачи

У Лены есть n неоткрытых коробок с фигурками. Открывать их нельзя, иначе фигурки теряют свою ценность, поэтому менять порядок фигурок в коробке тоже нельзя. Кроме того, коробку нельзя поворачивать, иначе фигурки будут смотреть не в ту сторону.

Каждая фигурка характеризуется определенной высотой. Например, в одной из коробок слева направо могут быть расположены три фигурки высотой 4, 5 и 7. Говоря о какой-либо коробке с фигурками, я всегда буду указывать высоту слева направо.

Лена хочет *расположить коробки* так, чтобы высота фигурок увеличивалась или оставалась неизменной слева направо.

Сможет ли она организовать коробки таким образом, зависит от высоты фигурок в них. Например, если в первой коробке есть фигурки высотой 4, 5 и 7, а во второй — фигурки высотой 1 и 2, то она может выполнить задачу, поместив вторую коробку слева от первой. Но если первая коробка останется прежней, а во второй окажутся фигурки высотой 6 и 8, то возможности организовать их уже не будет.

Определите, может ли Лена организовать коробки.

Входные данные

Входные данные состоят:

- из строки, содержащей целое число n — количество коробок от 1 до 100;
- n строк, по одной на коробку. Каждая из этих строк начинается с целого числа k, обозначающего количество фигурок в коробке. k находится в диапазоне от 1 до 100 (поскольку k равно как минимум 1, пустых коробок быть не может). После числа k идут k целых чисел, определяющих высоту фигурок слева направо в этом поле. Каждая высота представляет собой целое число от 1 до 10 000. Значения высот разделены пробелами.

Выходные данные

Если Лена может разложить коробки, выведите `YES`, в противном случае — `NO`.

Моделирование коробок

Эта задач состоит из нескольких подзадач поменьше, каждую из которых можно решить, написав функцию. Сначала посмотрим, как смоделировать сами коробки в Python, а затем создадим необходимые функции.

В главе 5, решая задачу «Бонус "Бейкера"», вы узнали, что списки можно вкладывать в другие списки в качестве значений. Получаются вложенные списки. Такую же структуру можно использовать для представления коробок с фигурками. Например, вот такой список моделирует две коробки:

```
>>> boxes = [[4, 5, 7], [1, 2]]
```

В первой коробке три фигурки, во второй — две. Мы можем обратиться к каждой коробке по отдельности:

```
>>> boxes[0]
[4, 5, 7]
>>> boxes[1]
[1, 2]
```

Будем считывать содержимое коробок и помещать эту информацию во вложенный список наподобие показанного ранее. Затем с помощью вложенного списка определим, можно ли организовать коробки.

Нисходящее проектирование

Чтобы решить эту задачу, мы воспользуемся подходом к разработке программ, называемым *нисходящим проектированием*. В нем большая задача разбивается на несколько более мелких. Это полезный подход, потому что каждую из более мелких задач решить легче. Затем можно собрать решения подзадач, чтобы решить исходную задачу.

Выполним проектирование

Рассмотрим, как работает нисходящее проектирование. Начнем с написания неполной программы на Python, в которой обозначим основные задачи общего решения. Некоторые из них не потребуют большого количества кода, поэтому можно приступить к их решению напрямую. Для других задач нужны будут чуть

бóльшие усилия, поэтому их мы превратим в функции. Саму же подзадачу будем решать, написав немножко кода *и* вызвав функцию. Вот только сами функции мы еще не написали, так что надо этим заняться!

Чтобы создать необходимую функцию, мы повторяем тот же процесс, но уже в отношении функции. Сперва определяем, что она должна делать. Если мы можем сразу написать код для задачи, то так и поступим, а если нет, сделаем еще одну функцию (которую напишем позже) для обработки этой задачи.

Этот процесс продолжается до тех пор, пока все функции не будут написаны. Когда этот произойдет, у нас будет готовое решение задачи.

Данный процесс называется нисходящим проектированием, потому что мы начинаем с верхнего, или самого высокого, уровня задачи и продвигаемся вниз, рассматривая внутренние подзадачи, пока все они не будут описаны в коде. Воспользуемся этим подходом для решения задачи «Фигурки».

Верхний уровень

Приступая к проектированию, мы ориентируемся на основные задачи, которые необходимо решить.

Нам обязательно нужно прочитать входные данные, это первая задача. Предположим, мы их прочитали. Что требуется сделать, чтобы определить, можно ли упорядочить коробки? Следует проверить каждую отдельно, чтобы убедиться: фигурки в ней имеют нужную высоту. Предположим, у нас есть коробка `[18, 20, 4]`. Высота фигурок в коробке не упорядочена, что означает: у нас нет возможности выстроить коробки нужным образом. Даже сама по себе она не организована!

Стала ясна вторая задача: определить, расположены ли фигурки в каждой коробке по порядку. Если в какой-либо из них это не так, можно сразу сделать вывод о том, что упорядочить коробки нельзя. Если же все коробки сами по себе удовлетворяют условиям, тогда нужно проверить еще что-нибудь.

Если внутри все коробки упорядочены, следующий вопрос заключается в том, можем ли мы организовать их. Здесь можно сделать одно важное наблюдение: с этого момента нас интересуют только крайняя левая и крайняя правая фигурки в коробках. Расположенные между ними фигурки больше не имеют значения.

Рассмотрим пример с тремя коробками:

```
[[9, 13, 14, 17, 25],
 [32, 33, 34, 36],
 [1, 6]]
```

Первая коробка начинается с фигурки высотой 9 и заканчивается фигуркой высотой 25. Фигурки, которые будут находиться слева от этой коробки, должны иметь высоту 9 или меньше. Так что с этой стороны можно разместить третью коробку. Все фигурки, расположенные справа от первой коробки, должны иметь высоту 25 или более. То есть там поставить разместить вторую коробку. Фигурки высотой 13, 14 и 17 сути дела не меняют, словно их там и нет.

Обрисовалась третья задача: убрать все фигурки, кроме тех, что находятся в крайних положениях в коробках. По итогу ее решения мы получим список, который выглядит следующим образом:

```
[[9, 25],
 [32, 36],
 [1, 6]]
```

Чтобы определить, сможем ли мы организовать коробки, отсортируем их:

```
[[1, 6],
 [9, 25],
 [32, 36]]
```

Теперь ясно видно, какие коробки должны стоять рядом друг с другом (мы использовали аналогичный подход при решении задачи «Деревни у дороги» в главе 5). Итак, четвертая задача — отсортировать коробки.

Пятая и последняя задача — определить, организованы ли отсортированные коробки. Они организованы, если фигурки упорядочены по высоте слева направо. Фигурки высотой 1, 6, 9, 25, 32 и 36 расположены правильно, поэтому коробки могут быть организованы. А теперь рассмотрим другой пример:

```
[[1, 6],
 [9, 50],
 [32, 36]]
```

Эти коробки нельзя организовать из-за огромной фигурки во второй коробке. Высота фигурок в ней от 9 до 50, поэтому третья коробка не может стоять справа, так как высота фигурок в ней слишком мала.

Мы закончили работу над задачей и выделили несколько подзадач.

1. Прочитать входные данные.
2. Проверить, упорядочено ли содержимое всех коробок.
3. Получить новый список коробок, в которых останутся только крайние левые и правые фигурки.
4. Отсортировать новые коробки.
5. Определить, упорядочены ли отсортированные коробки.

Вы спросите: а почему у нас есть задача «чтение входных данных», но нет «запись выходных данных». Дело в том, что здесь достаточно вывести слово YES или NO, так что в этой подзадаче нет ничего особенного. Кроме того, мы выведем YES или NO, как только узнаем ответ, поэтому вывод будет пересекаться с другими задачами. Поэтому я решил не включать запись выходных данных в список задач. При самостоятельной работе над нисходящим проектированием вы тоже можете упустить какую-нибудь подзадачу. Но ничего страшного в этом нет — добавьте ее и продолжите работу.

Записать требуемые задачи в виде кода можно следующим образом:

```
❶ # Основная программа

  # TODO: Чтение входных данных

  # TODO: Проверка корректности всех коробок

  # TODO: Получение нового списка коробок, в которых
  #       остаются только крайние фигурки

  # TODO: Сортировка коробок

  # TODO: Определение того, организованы ли коробки
```

Я назвал этот блок основной программой ❶. Любые функции, которые мы напишем, будут расположены выше этого комментария.

Пока что каждая подзадача представлена в виде комментария. Обозначения TODO служат для того, чтобы подчеркнуть: это задачи, которые нужно превратить из описания на естественном языке в код Python. Закончив задачу, мы сразу удалим ее TODO. Это позволит отслеживать, какие задачи выполнены, а какие — нет. Поехали!

Подзадача 1. Чтение входных данных

Нужно прочитать строку, содержащую число n (количество коробок), а затем прочитать сами коробки. Чтение целого числа выполняется одной строкой, поэтому прочитаем n напрямую. А вот чтение информации о коробках — это четко определенная задача, для решения которой потребуются несколько строк кода. Решим ее с помощью функции, которую назовем read_boxes. Тогда в основной программе получится вот такой блок:

```
  # Основная программа

❶ # Чтение входных данных
  n = int(input())
  boxes = read_boxes(n)

  # TODO: Проверка корректности всех коробок
```

```
# TODO: Получение нового списка коробок, в которых
#       остаются только крайние фигурки

# TODO: Сортировка коробок

# TODO: Определение того, организованы ли коробки
```

Я удалил TODO из первого комментария ❶, так как с точки зрения основной программы мы эту задачу решили. Конечно, функцию read_boxes тоже нужно написать, так что сделаем это прямо сейчас.

Функция read_boxes принимает целое число n в качестве параметра и считывает и возвращает n коробок. Вот ее код:

```
def read_boxes(n):
    """
    n — это число коробок, которые надо прочитать.

    Коробки читаются из консоли, после чего функция возвращает
    список коробок, где есть список высот фигурок в ней.
    """
    boxes = []
❶   for i in range(n):
        box = input().split()
❷       box.pop(0)
        for i in range(len(box)):
            box[i] = int(box[i])
        boxes.append(box)
    return boxes
```

Поскольку считать нужно n коробок, цикл выполняется n раз ❶. На каждой его итерации мы читаем текущую строку и разбиваем ее на отдельные высоты фигурок. Строка начинается с целого числа, обозначающего количество фигурок в коробке, поэтому удаляем это значение из списка (оно имеет индекс 0) ❷. Затем конвертируем каждую высоту в целое число и добавляем текущую коробку в список. Наконец, возвращаем список коробок.

Поскольку мы не передавали ничего из read_boxes какой-то еще написанной функции, то с этой задачей покончено! Эту функцию, как и другие, поставим перед комментарием # Основная программа.

Подзадача 2. Проверяем, все ли коробки упорядочены

В каждой ли коробке фигурки расположены по возрастанию? Хороший вопрос, и ответить на него парой строк кода не получится. Напишем новую функцию all_boxes_ok, которая даст ответ. Если эта функция возвращает False, значит, по

крайней мере в одной коробке фигурки перепутаны и мы не сможем организовать коробки. В этом случае нужно вывести NO. Если `all_boxes_ok` возвращает `True`, следует выполнить оставшиеся задачи, чтобы определить, можно ли организовать коробки. Давайте допишем в программу эту часть логики `if-else`. Вот что у нас получится:

```
  # Основная программа

  # Чтение входных данных
  n = int(input())
  boxes = read_boxes(n)

  # Проверка упорядоченности всех коробок
❶ if not all_boxes_ok(boxes):
      print('NO')
  else:
      # TODO: Получение нового списка коробок,
      #       в которых остаются лишь крайние фигурки

      # TODO: Сортировка коробок

      # TODO: Определение того, организованы ли коробки
```

Осталось написать функцию `all_boxes_ok`, которую мы уже вызываем в коде ❶. Надо проверить каждую коробку, чтобы определить, подходит ли она нам. Если это не так, сразу же возвращаем `False`. Если да, проверяем следующую коробку. Если все коробки будут в порядке, вернем `True`.

Ага, значит, нам нужна возможность проверить каждую коробку отдельно! Похоже, требуется еще одна функция. Назовем ее `box_ok`.

Вот таким получился код функции `all_boxes_ok`:

```
def all_boxes_ok(boxes):
    """
    boxes — это список коробок, а каждая коробка — это
    список высот фигурок.

    Возвращает True, если во всех коробках в списке
    фигурки расположены в неубывающем порядке,
    иначе возвращает False.
    """
    for box in boxes:
        if not box_ok(box):
            return False
    return True
```

В комментарии я назвал порядок неубывающим, а не возрастающим, чтобы допустить одинаковые высоты соседних фигурок. Например, коробка вида `[4, 4, 4]` нам вполне подойдет, но при этом слово «возрастание» к ней применить нельзя.

Мы поместили часть работы функции `all_boxes_ok` в функцию `box_ok`, так что теперь напишем эту функцию. Вот она:

```
def box_ok(box):
    """
    box — это список высот фигурок в коробке.

    Возвращает True, если фигурки в коробке
    расположены в неубывающем порядке, иначе
    возвращает False.
    """
    for i in range(len(box)):
        if box[i] > box[i + 1]:
            return False
    return True
```

Если какая-либо фигурка выше расположенной справа от нее, возвращаем `False`, поскольку фигурки не упорядочены. Если мы проходим цикл `for`, значит, нарушений высоты нет, поэтому возвращаем `True`.

Одним из приятных побочных эффектов нисходящего проектирования является то, что мы получаем небольшие фрагменты кода, упакованные в виде функций, которые позже можно будет тестировать изолированно. Например, введите код функции `box_ok` в оболочку Python. Затем протестируйте ее:

```
>>> box_ok([4, 5, 6])
```

Мы ожидаем, что здесь вернется `True`, потому что коробки упорядочены. И уж определенно не ожидали того, что получили:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in box_ok
IndexError: list index out of range
```

Ошибки — это всегда грустно, а еще грустнее становится, когда приходится пролистывать множество страниц кода, чтобы их найти. Но здесь мы знаем, что ошибка локализована в этой маленькой функции, поэтому найти ее значительно проще. Проблема здесь в том, что мы в конечном итоге сравниваем высоту крайней правой фигурки с высотой фигурки справа от нее, которой не существует! Поэтому нужно остановить цикл на итерацию раньше, сравнив предпоследнюю высоту с последней. Далее приведен обновленный код:

```
def box_ok(box):
    """
    box – это список высот фигурок в коробке.

    Возвращает True, если фигурки в коробке
    расположены в неубывающем порядке, иначе
    возвращает False.
    """
❶   for i in range(len(box) - 1):
        if box[i] > box[i + 1]:
            return False
    return True
```

Единственное изменение находится в функции `range` ❶. Если вы протестируете эту версию функции, то увидите, что она работает правильно. Мы справились с задачей номер 21

Подзадача 3. Оставить в коробках только крайние фигурки

Не забывайте, что мы осваиваем нисходящее проектирование. В этой задаче требуется найти способ превратить коробки со всеми фигурками в содержащие только крайние левые и крайние правые фигурки. Называть крайние фигурки будем `boxes_endpoints`.

Один из вариантов решения — создать новый список коробок с крайними фигурками, именно так я и поступлю. Можно также удалить ненужные числа из исходных коробок, но это немного сложнее.

Функцию для этой задачи я назвал `box_endpoints`. Основная часть программы, дополненная вызовом этой функции, теперь выглядит так:

```
# Основная программа

# Чтение входных данных
n = int(input())
boxes = read_boxes(n)

# Проверка корректности всех коробок
if not all_boxes_ok(boxes):
    print('NO')
else:
    # Получение нового списка коробок, в которых
    # остаются только крайние фигурки
❶   endpoints = boxes_endpoints(boxes)

    # TODO: Сортировка коробок

    # TODO: Определение того, организованы ли коробки
```

Когда мы вызываем функцию `box_endpoints` со списком коробок ❶, то ожидаем, что получим новый список коробок с крайними фигурками. Далее приведен код функции `box_endpoints`, который выполняет это требование:

```
def boxes_endpoints(boxes):
    """
    boxes – это список коробок, а каждая коробка –
    это список высот фигурок.

    Возвращает список, каждое значение в котором –
    это список двух значений: высот крайней левой
    и крайней правой фигурок.
    """
❶   endpoints = []
    for box in boxes:
❷     endpoints.append([box[0], box[-1]])
    return endpoints
```

Мы создаем новый список ❶, который будет содержать крайние фигурки каждой коробки. Затем перебираем коробки. В каждой из них с помощью индексации находим крайние левую и правую фигурки и добавляем их в список крайних фигурок. Наконец, возвращаем этот список ❷.

Но что произойдет, если в коробке будет только одна фигурка? Как будет работать функция `box_endpoints`? В строке документации сказано, что она возвращает список из двух значений для всех коробок. Так оно и должно работать, а иначе функция не выполняет то, что в ней заявлено. Давайте проверим. Введите имя функции `box_endpoints` в оболочку Python и передайте ей список из одной коробки с одной фигуркой:

```
>>> boxes_endpoints([[2]])
[[2, 2]]
```

Получилось! Самая левая высота равна 2, самая правая — тоже 2, поэтому мы получаем список с двумя цифрами 2. В этом случае функция работает правильно, потому что `box [0]` и `box [-1]` — это одно и то же значение, если в коробке одна фигурка (а по условию задачи пустых коробок нет).

Подзадача 4. Сортируем коробки

На данный момент у нас есть список коробок с крайними фигурками:

```
>>> endpoints = [[9, 25], [32, 36], [1, 6]]
>>> endpoints
[[9, 25], [32, 36], [1, 6]]
```

Нужно их отсортировать. Требуется ли для этого еще одна функция — что-то вроде `sort_endpoints`?

Не в этот раз! Метод `sort` делает именно то, что нам нужно:

```
>>> endpoints.sort()
>>> endpoints
[[1, 6], [9, 25], [32, 36]]
```

При сортировке двумерных списков метод `sort` сортирует по первому значению (а при их равенстве выполняется сортировка по второму значению).

Сортировку можно сразу же добавить в основную часть программы и закрыть тем самым одно `TODO`. Обновленный код:

```
# Основная программа

# Чтение входных данных
n = int(input())
boxes = read_boxes(n)

# Проверка упорядоченности всех коробок
if not all_boxes_ok(boxes):
    print('NO')
else:
    # Получение нового списка коробок, в которых
    # остаются только крайние фигурки
    endpoints = boxes_endpoints(boxes)

    # Сортировка коробок
    endpoints.sort()

    # TODO: Определение того, организованы ли коробки
```

Мы почти у цели. Осталось только одно `TODO`.

Подзадача 5. Определить, организованы ли коробки

Последняя задача — проверить крайние фигурки. Они могут быть упорядочены, например:

```
[[1, 6],
 [9, 25],
 [32, 36]]
```

А могут и не быть:

```
[[1, 6],
 [9, 50],
 [32, 36]]
```

В первом случае мы должны вывести YES, в последнем — NO. Нам нужна функция, которая сообщит, упорядочены ли крайние фигурки. Обновим основную часть программы в последний раз, и получится вот что:

```
# Основная программа

# Чтение входных данных
n = int(input())
boxes = read_boxes(n)

# Проверка упорядоченности всех коробок
if not all_boxes_ok(boxes):
    print('NO')
else:
    # Получение нового списка коробок, в которых
    # остаются только крайние фигурки
    endpoints = boxes_endpoints(boxes)

    # Сортировка коробок
    endpoints.sort()

    # Определение того, организованы ли коробки
❶   if all_endpoints_ok(endpoints):
        print('YES')
    else:
        print('NO')
```

Для полного счастья не хватает лишь функции `all_endpoints_ok`, которую мы вызываем ❶. Она принимает список, в котором каждое значение представляет собой список крайних фигурок, и возвращает `True`, если они упорядочены, и `False`, если нет.

Давайте подумаем, как реализовать эту функцию, рассмотрев пример. Далее приведен список фигурок:

```
[[1, 6],
 [9, 25],
 [32, 36]]
```

В первой коробке справа находится фигурка высотой 6. Значит, во второй коробке слева должна быть фигурка высотой 6 или более. Если это не так, возвращаем `False`, указывая на то, что фигурки не упорядочены. Но у нас все хорошо, потому что во второй коробке слева стоит 9.

Повторим эту проверку для числа 25 — высоты правой фигурки из второй коробки. Левая фигурка в третьей коробке — 32, так что все хорошо.

Получается, что если левая фигурка из некоторой коробки ниже правой фигурки из предыдущей коробки, нужно вернуть `False`. Если же все эти проверки пройдены, мы возвращаем `True`.

Тогда код будет вот таким:

```
def all_endpoints_ok(endpoints):
    """
    endpoints — это список, каждое значение в котором — это список
    двух значений: высот крайней левой и крайней правой фигурок.

 ❶ Требуется отсортировать endpoints по высотам фигурок.

    Возвращает True, если фигурки во всех коробках
    можно упорядочить, и False в противном случае.
    """
 ❷ maximum = endpoints[0][1]
    for i in range(1, len(endpoints)):
        if endpoints[i][0] < maximum:
            return False
     ❸ maximum = endpoints[i][1]
    return True
```

Я добавил в строку документации кое-какую информацию о том, что требуется функция при ее вызове ❶. В частности, перед вызовом этой функции важно отсортировать фигурки. В противном случае она может вернуть неправильное значение.

В переменной `endpoints` находится список с двумя значениями: индекс `0` — крайняя левая (минимальная) высота, а индекс `1` — крайняя правая (максимальная). В переменной `maximum` отслеживается максимальная высота фигурки в коробке. До цикла `for` в этой переменной содержалась наибольшая высота из первой коробки ❷. Цикл `for` сравнивает минимум следующей коробки с этим максимумом. Если минимум следующей коробки слишком мал, возвращаем `False`, потому что эти коробки стоят неправильно. Последнее, что нужно сделать на каждой итерации, — обновить `maximum`, чтобы переменная указывала уже на другую коробку ❸.

Собираем все вместе

Написав код для всех задач, включая функции, которые возникли в ходе проектирования, можем собрать всю проделанную работу в единое целое. Вы можете оставить комментарии в основной части программы, а можете не делать этого. Я сохранил их, но на практике иногда возникают случаи чрезмерного документирования кода, поскольку имена функций обычно сами говорят о том, как работают. В листинге 6.2 приведен полный код.

Листинг 6.2. Решение задачи «Фигурки»

```
def read_boxes(n):
    """
    n — это количество коробок, которые нужно
    считать.

    Коробки считываются из ввода и возвращаются как
    список коробок, каждая из которых — это список
    высот фигурок.
    """
    boxes = []
    for i in range(n):
        box = input().split()
        box.pop(0)
        for i in range(len(box)):
            box[i] = int(box[i])
        boxes.append(box)
    return boxes

def box_ok(box):
    """
    box — это список высот фигурок в коробке.

    Возвращает True, если фигурки в коробке
    расположены в неубывающем порядке, иначе
    возвращает False.
    """
    for i in range(len(box) - 1):
        if box[i] > box[i + 1]:
            return False
    return True

def all_boxes_ok(boxes):
    """
    boxes — это список коробок, а каждая коробка — список высот фигурок.

    Возвращает True, если во всех коробках в списке
    фигурки расположены в неубывающем порядке,
    иначе возвращает False.
    """
    for box in boxes:
        if not box_ok(box):
            return False
    return True

def boxes_endpoints(boxes):
    """
    boxes — это список коробок, а каждая коробка — это список высот фигурок.
```

```
    Возвращает список, каждое значение в котором –
    это список двух значений: высот крайней левой
    и крайней правой фигурок.
    """
    endpoints = []
    for box in boxes:
        endpoints.append([box[0], box[-1]])
    return endpoints

def all_endpoints_ok(endpoints):
    """
    endpoints – это список, каждое значение в котором –
    список двух значений: высот крайней левой
    и крайней правой фигурок.

    Требуется отсортировать endpoints по высотам фигурок.

    Возвращает True, если фигурки из всех коробок
    можно упорядочить, и False в противном случае.
    """
    maximum = endpoints[0][1]
    for i in range(1, len(endpoints)):
        if endpoints[i][0] < maximum:
            return False
        maximum = endpoints[i][1]
    return True

# Основная программа

# Чтение входных данных
n = int(input())
boxes = read_boxes(n)

# Проверка упорядоченности всех коробок
if not all_boxes_ok(boxes):
    print('NO')
else:
    # Получение нового списка коробок, в которых
    # остаются только крайние фигурки
    endpoints = boxes_endpoints(boxes)

    # Сортировка коробок
    endpoints.sort()

    # Определение того, организованы ли коробки
    if all_endpoints_ok(endpoints):
        print('YES')
    else:
        print('NO')
```

Таких больших программ мы в этой книге еще не писали. Но посмотрите, насколько аккуратна и минималистична ее главная часть: в основном она состоит из вызова функций и парочки `if-else`, которые объединяют функции.

Вызов каждой функции выполняется лишь один раз. А вот в задаче «Карточная игра» функцию `no_high` мы вызывали четыре раза. Даже если функция вызывается лишь раз, она по-прежнему вносит свой вклад в организованный, читаемый код.

Отправляйте код на сайт Timus. Вы увидите, что все тестовые примеры выполняются правильно.

ПРОВЕРИМ ЗНАНИЯ

В задаче 2 мы написали функцию `box_ok`, которая определяет упорядоченность коробки. В ней мы использовали цикл `for`. Верна ли приведенная далее реализация той же функции с помощью цикла `while`?

```
def box_ok(box):
    """
    box — это список высот фигурок в коробке.

    Возвращает True, если фигурки в коробке
    расположены в неубывающем порядке, иначе
    возвращает False.
    """
    ok = True
    i = 0
    while i < len(box) - 1 and ok:
        if box[i] > box[i + 1]:
            ok = False
        i = i + 1
    return ok
```

А. Да.

Б. Нет, будет выведена ошибка `IndexError`.

В. Нет, но ошибок не будет, вернется неправильное значение.

Ответ: **А**. Эта функция выполняет то же, что и цикл `for` ранее. Переменная `ok` начинается со значения `True`, то есть все значения высоты предполагаются правильными. Цикл `while` работает до последней коробки или до первой неупорядоченной коробки. Если таковая найдется, переменной `ok` присваивается `False` и цикл прекращается. Если не найдется, до конца дойдет значение `True` и мы возвращаем `True` из функции.

Резюме

В этой главе вы узнали о функциях. Так называется автономный блок кода, который решает небольшую часть более крупной задачи. Мы узнали, как передавать в функцию информацию (с помощью аргументов) и получать сведения из функций (через возвращаемое значение).

Чтобы определить, какие функции вообще нам нужны, можно применить нисходящее проектирование. Оно помогает разбить решение большой задачи на ряд более мелких. Мы либо решаем каждую задачу напрямую, либо, если не получается, пишем функцию. Если задача слишком громоздка, можем выполнить дальнейшее проектирование на уровень ниже.

В следующей главе вы узнаете, как работать с файлами, вместо того чтобы использовать стандартные ввод и вывод. Продолжая раздвигать границы своих познаний, найдете множество применений для функций в этой главе и в остальной части книги. Практикуйтесь с приведенными далее упражнениями, чтобы получше набить руку в работе с функциями.

Упражнения

Здесь приведены несколько упражнений, которые вы можете попробовать выполнить. Воспользуйтесь нисходящим проектированием, чтобы определить одну или несколько функций, которые помогут организовать код. И не забудьте добавить в каждую из них строку документации!

1. DMOJ, задача From 1987 to 2013 с кодом `ccc13s1`.
2. DMOJ, задача Are we there yet? с кодом `ccc18j3`.
3. DMOJ, задача Decoding DNA с кодом `ecoo12r1p2`.
4. DMOJ, задача Platforme с кодом `crci07p1`.
5. DMOJ, задача Misa с кодом `coci13c2p2`.
6. Вернитесь к некоторым упражнениям из главы 5 и улучшите свои решения, используя функции. Особенно рекомендую вернуться к задачам DMOJ `coci18c2p1` (Preokret) и DMOJ `ccc00s2` (Babbling Brooks).

Примечания

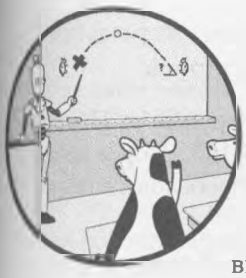
Задача «Карточная игра» взята из Canadian Computing Competition 1999 года. «Фигурки» — с конкурса Ural School Programming Contest 2019 года.

Многие современные языки программирования, включая Python, поддерживают две различные парадигмы программирования. Одна из них основана на функциях,

и именно их мы изучали в этой главе. Другая основана на *объектах* и называется *объектно-ориентированным программированием (ООП)*. ООП подразумевает определение новых типов и методов их написания. На протяжении всей книги мы используем типы Python, такие как целые числа и строки, но саму парадигму ООП обсуждать не будем. Если интересно почитать про ООП и рассмотреть практические примеры его применения, я рекомендую книгу Эрика Маттеса *Python Crash Course*, 2-е издание (No Starch Press, 2019).

7

Чтение из файлов и запись в них



Пока что во всех программах мы читали входные данные функцией `input`, а результаты выводили функцией `print`. Эти функции читают данные со стандартного ввода (по умолчанию это клавиатура), а запись выполняют в стандартный вывод (по умолчанию на экран) соответственно. У нас есть возможность изменить это поведение по умолчанию с помощью техники перенаправления ввода и вывода, но программам все равно часто требуется возможность управления файлами. Например, ваш текстовый процессор позволяет открыть любой нужный файл документа и сохранить его под любым именем, не заставляя возиться со стандартными вводом и выводом.

В этой главе вы узнаете, как писать программы, позволяющие работать с текстовыми файлами. С помощью файлов мы решим две задачи: правильно отформатируем эссе и настроим ферму для кормления коров.

Задача 16. Форматирование эссе

Отмечу одно важное различие между этой задачей и всеми задачами, которые мы успели решить: здесь требуется выполнять чтение из определенных файлов и запись в конкретные файлы! Обращайте на это внимание, читая описание задачи.

Рассмотрим задачу Word Processor с конкурса January Bronze Contest USACO 2020. Это первая задача на сайте USACO (Олимпиада по вычислительной технике США). Чтобы найти ее, перейдите на сайт http://usaco.org/, щелкните на вкладке Contests, затем на 2020 January Contest Results, далее нажмите кнопку View problem под задачей Word Processor.

Постановка задачи

Корова Бесси пишет сочинение. Каждое слово в нем содержит только строчные или прописные буквы. Учитель сказал ей, из какого максимального количества символов, не считая пробелов, может состоять строка. Чтобы удовлетворить это требование, Бесси пишет эссе по следующим правилам.

- Если следующее слово помещается в текущую строку, добавляет его в нее. Все слова разделяются пробелами.
- В противном случае пишет слово с новой строки, и она становится текущей.

Требуется вывести эссе с правильной расстановкой слов.

Входные данные

Входные данные находятся в файле `word.in`. Они состоят из двух строк.

- Первая строка содержит два целых числа, разделенных пробелом. Первое целое число, `n` — количество слов в эссе в диапазоне от 1 до 100. Второе целое число, `k` — максимальное количество символов (не считая пробелов), которое может стоять в строке, в диапазоне от 1 до 80.
- Вторая строка содержит n слов, разделенных пробелами. В каждом слове не более k символов.

Выходные данные

Выходные данные записываются в файл с именем `word.out`. Требуется вывести эссе в правильном формате.

Работа с файлами

Для решения задачи требуется прочитать данные из файла `word.in` и записать их в файл `word.out`. Чтобы сделать это, нужно сперва научиться открывать файлы в наших программах.

Открытие файла

Используя текстовый редактор, создайте файл с именем `word.in`. Поместите его в папку с программой, которую мы будем писать.

Мы впервые создаем файл с расширением, отличным от `.py`. В этот раз нужно расширение `.in`. Обязательно назовите файл `word.in`, а не `word.py`. `in` — это сокращение

от input, и это расширение часто применяется для имен файлов, содержащих входные данные программы.

В этот файл поместим допустимые данные для задачи форматирования эссе. Введите в файл следующее:

```
12 13
perhaps better poetry will be written in the language of digital computers
```

Сохраните файл.

Чтобы открыть файл с помощью Python, воспользуемся функцией `open`. Ей мы передаем два аргумента: первый — имя файла, второй — режим его открытия. Режим определяет, как мы можем взаимодействовать с файлом.

Откроем файл `word.in`:

```
  >>> open('word.in', 'r')
❶ <_io.TextIOWrapper name='word.in' mode='r' encoding='cp1252'>
```

В этом вызове функции мы указали режим `r`. Он означает «чтение (read)», и функция открывает файл, чтобы мы могли считать из него данные. Режим `r` не является обязательным параметром, так как это значение по умолчанию, поэтому его можно и не указывать. Но для единообразия я буду явно прописывать букву `r` по всей книге.

Когда мы используем функцию `open`, Python выдает некоторую информацию о выполненном действии ❶, например, сообщает имя файла и режим. Параметр `encoding` указывает на то, как файл был декодирован из его состояния на диске в форму, которую мы можем прочитать. Файлы кодируются с помощью множества кодировок, но мы об этом думать не будем.

Если мы попытаемся открыть для чтения несуществующий файл, то получим ошибку:

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

Встретив эту ошибку при открытии файла `word.in`, проверьте, правильно ли назван файл и находится ли он в той же папке, где и ваша программа на Python.

Помимо режима `r`, который позволяет выполнять чтение, есть режим `w` — режим записи. С помощью параметра `w` мы можем открыть файл и что-нибудь записать в него.

С режимом `w` следует обращаться аккуратно. Если вы используете его с уже существующим файлом, содержимое последнего будет удалено. На файле `word.in` я показал это для примера, но его легко воссоздать, поэтому не страшно. Но если мы случайно перезапишем важный файл, никто за это спасибо не скажет.

Если вы задействуете `w` с именем файла, которого не существует, будет создан пустой файл.

Воспользуемся режимом `w` для создания пустого файла с именем `blah.in`:

```
>>> open('blah.in', 'w')
<_io.TextIOWrapper name='blah.in' mode='w' encoding='cp1252'>
```

Теперь файл `blah.in` создан, и мы можем открыть его для чтения, не получив ошибок:

```
>>> open('blah.in', 'r')
<_io.TextIOWrapper name='blah.in' mode='r' encoding='cp1252'>
```

А что за `_io.TextIOWrapper` постоянно вылезает? Это тип возвращаемого функцией `open` значения:

```
>>> type(open('word.in', 'r'))
<class '_io.TextIOWrapper'>
```

Можно считать, что это тип для работы с файлами. Его значения представляют открытые файлы, и мы увидим, что у этого типа есть методы, которые можно вызывать.

Как и в случае с любой функцией, если мы не присваиваем переменной ее возвращаемое значение, оно теряется. То, как мы вызывали функцию `open` до сих пор, не позволяет воспользоваться файлом, который открыли!

А вот так можно научить переменную ссылаться на открытый файл:

```
>>> input_file = open('word.in', 'r')
>>> input_file
<_io.TextIOWrapper name='word.in' mode='r' encoding='cp1252'>
```

Теперь мы сможем применять переменную `input_file` для чтения из файла `word.in`.

Для решения задачи нам понадобится также способ записи данных в файл `'word.out'`. Определим переменную, которая поможет в этом:

```
>>> output_file = open('word.out', 'w')
>>> output_file
<_io.TextIOWrapper name='word.out' mode='w' encoding='cp1252'>
```

Чтение из файла

Чтобы прочитать строку из открытого файла, воспользуемся методом файла `readline`. Он возвращает строку, включающую в себя содержимое следующей строки файла. По сути, этот метод работает аналогично функции `input`. Но, в отличие от нее, `readline` читает из файла, а не из стандартного ввода.

Давайте откроем файл `word.in` и прочитаем две его строки:

```
>>> input_file = open('word.in', 'r')
>>> input_file.readline()
'12 13\n'
>>> input_file.readline()
'perhaps better poetry will be written in the language of digital computers\n'
```

В конце каждой строки появился неожиданный символ `\n`, которого не было при использовании функции `input`. Символ `\` в строке — это *экранирующий символ*. Он влияет на стандартную интерпретацию следующих за ним символов и меняет их значение. Пара `\n` рассматривается как единый символ, а именно символ новой строки. Все строки в файле, кроме, может быть, последней, заканчиваются символом новой строки. Если бы их не было, весь текст был бы написан одной строкой! Метод `readline` буквально дает нам всю строку, включая завершающий символ новой строки.

Мы можем вставлять символ новой строки в собственные строки:

```
>>> 'one\ntwo\nthree'
'one\ntwo\nthree'
>>> print('one\ntwo\nthree')
one
two
three
```

Оболочка Python не обрабатывает работу экранирующих символов, а вот функция `print` — другое дело.

Последовательность `\n` позволяет разделить текст на несколько строк. Однако, когда мы читаем строки из файлов, нам редко нужны эти символы. Чтобы избавиться от них, можно применить строковый метод `rstrip`. Он похож на `strip`, но удаляет пробелы только справа от строки (а слева не удаляет). Символы новой строки тоже обрабатываются как пробелы:

```
>>> 'hello\nthere\n\n'
'hello\nthere\n\n'
>>> 'hello\nthere\n\n'.rstrip()
'hello\nthere'
```

Попробуем снова выполнить чтение из файла, на этот раз убрав лишние символы:

```
>>> input_file = open('word.in', 'r')
>>> input_file.readline().rstrip()
'12 13'
>>> input_file.readline().rstrip()
'perhaps better poetry will be written in the language of digital computers'
```

Итак, мы прочитали две строки и читать из файла больше нечего. Метод `readline` сигнализирует об этом, возвращая пустую строку.

```
>>> input_file.readline().rstrip()
''
```

Она означает, что мы достигли конца файла. Если нужно прочитать его содержимое, следует снова открыть файл и начать чтение с начала.

Так и сделаем, но на этот раз сохраним считанные данные в переменных:

```
>>> input_file = open('word.in', 'r')
>>> first = input_file.readline().rstrip()
>>> second = input_file.readline().rstrip()
>>> first
'12 13'
>>> second
'perhaps better poetry will be written in the language of digital computers'
```

Если нужно прочитать все строки из файла независимо от их количества, можно использовать цикл `for`. Файлы в Python обрабатываются как последовательности строк, поэтому мы можем перебирать их так же, как строки и списки:

```
>>> input_file = open('word.in', 'r')
>>> for line in input_file:
...     print(line.rstrip())
...
12 13
perhaps better poetry will be written in the language of digital computers
```

В отличие от строк, перебрать файл второй раз нельзя, потому что при первом переборе мы доходим до конца. Если попробуем, ничего не получится:

```
>>> for line in input_file:
...     print(line.rstrip())
...
```

ПРОВЕРИМ ЗНАНИЯ

Мы хотим с помощью цикла `while` вывести на экран содержимое файла `input_file` построчно (это может быть любой файл, не обязательно связанный с рассматриваемой задачей). Какой из приведенных фрагментов кода позволяет правильно сделать это?

А
```
while input_file.readline() != '':
    print(input_file.readline().rstrip())
```

Б
```
line = 'x'
while line != '':
    line = input_file.readline()
    print(line.rstrip())
```

В
```
line = input_file.readline()
while line != '':
    line = input_file.readline()
    print(line.rstrip())
```

Г. Все перечисленные.

Д. Ни один.

Я рекомендую вам, прежде чем смотреть ответ, создать файл с четырьмя или пятью строками и опробовать на нем каждый фрагмент кода. Можете также добавить символ * в начало каждой выводимой строки, чтобы можно было опознать пустые строки.

Ответ: **Д**. В каждом фрагменте кода есть небольшая ошибка.

Код **А** выводит только каждую вторую строку файла. Например, логическое выражение цикла `while` вызывает чтение первой строки... а затем оно теряется, потому что не присваивается переменной. Таким образом, первая итерация цикла выводит вторую строку файла.

Код **Б** очень близок к правде. Он выводит все строки файла, но добавляет лишнюю пустую строку в конце.

Код **В** не выводит первую строку файла. Все дело в том, что первая строка читается перед циклом, но затем он читает вторую строку, не выводя первую. К тому же код создает лишнюю пустую строку в конце, как и код **Б**.

А вот правильный код для чтения и вывода каждой строки:

```
line = input_file.readline()
while line != '':
    print(line.rstrip())
    line = input_file.readline()
```

Запись в файл

Чтобы записать строку в открытый файл, используем метод `write` файла. Мы передаем методу строку, и она добавляется в конец файла.

Для решения задачи нужно выполнять запись в файл `word.out`. Мы еще не готовы решить эту задачу, поэтому вместо этого выполним запись в `blah.out`. Вот как можно записать в файл одну строку:

```
>>> output_file = open('blah.out', 'w')
>>> output_file.write('hello')
5
```

А откуда там цифра `5`? Метод `write` возвращает количество записанных символов. Это позволяет подтвердить, что мы записали все, что планировали.

Открыв файл `blah.out` в текстовом редакторе, вы должны увидеть в нем слово `hello`.

Попробуем записать в файл три строки:

```
>>> output_file = open('blah.out', 'w')
>>> output_file.write('sq')
2
>>> output_file.write('ui')
2
>>> output_file.write('sh')
2
```

Исходя из своих новых знаний, вы могли ожидать, что в файл `blah.out` попадет вот такой текст:

```
sq
ui
sh
```

На самом деле это не так:

```
squish
```

Символы находятся в одной строке, потому что метод `write` не добавляет новые строки! Если нужны отдельные строки, об этом нужно сказать явно, например:

```
>>> output_file = open('blah.out', 'w')
>>> output_file.write('sq\n')
3
>>> output_file.write('ui\n')
3
>>> output_file.write('sh\n')
3
```

Обратите внимание на то, что в каждом случае записывают три символа, а не два. Новая строка считается символом. Теперь, если вы откроете `blah.out` в текстовом редакторе, записанный туда текст будет разбит на три строки:

```
sq
ui
sh
```

В отличие от функции `print` метод `write` работает только со строками. Чтобы записать в файл число, его сначала нужно преобразовать в строку:

```
>>> num = 7788
>>> output_file = open('blah.out', 'w')
>>> output_file.write(str(num) + '\n')
5
```

Закрытие файлов

Хорошая практика — закрывать файл по окончании работы с ним. Это позволяет читающим ваш код понять, что он больше не используется.

Закрытие файлов также помогает операционной системе управлять ресурсами вашего компьютера. Когда вы применяете метод `write`, передаваемая строка попадает в файл не сразу. Вместо этого Python или операционная система могут подождать, пока не получат несколько запросов на запись, а затем выполнят их все сразу. Закрытие файла, в который вы что-то записали, гарантирует, что записанное в файл будет там в целости и сохранности.

Чтобы закрыть файл, нужно вызвать метод `close`. Далее приведен пример открытия файла, чтения строки и ее закрытия:

```
>>> input_file = open('word.in', 'r')
>>> input_file.readline()
'12 13\n'
>>> input_file.close()
```

Закрыв файл, вы больше не сможете взаимодействовать с ним:

```
>>> input_file.readline()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Решение задачи

Вернемся к задаче о форматировании эссе. Теперь мы знаем, как читать из файла `word.in` и записывать в файл `word.out`. То есть с требованиями к вводу и выводу разобрались. Пора подумать о сути задачи.

Начнем с изучения тестового примера, чтобы убедиться, что решение нам понятно. А потом рассмотрим код.

Тестовый пример

Вот файл `word.in`, который я использовал:

```
12 13
perhaps better poetry will be written in the language of digital computers
```

В нем всего 12 слов, а максимальное количество символов в строке, не считая пробелов, — 13. Добавлять слова в текущую строку можно до тех пор, пока они умещаются в ней. Как только слово перестает умещаться, начинаем с него следующую строку.

Слово `perhaps` состоит из семи символов, поэтому оно помещается в первой строке. Слово `better` состоит из шести символов. Оно тоже помещается в первую строку. У нас получилось уже 13 символов, не считая пробела между двумя словами.

Слово `poetry` не помещается в первой строке, поэтому начнем с него новую строку. Слово `will` помещается во второй строке. Слово `be` — тоже. Пока во второй строке 12 символов, не являющихся пробелами. Слово `written` в оставшийся один символ никак не влезет, и мы вынуждены начать с него следующую строку.

Изучив таким образом входную строку до конца, получим следующее содержимое файла `word.out`:

```
perhaps better
poetry will be
written in the
language of
digital
computers
```

Код

Решение задачи приведено в листинге 7.1.

Листинг 7.1. Решение задачи «Форматирование эссе»

```
❶ input_file = open('word.in', 'r')
❷ output_file = open('word.out', 'w')

❸ lst = input_file.readline().split()
  n = int(lst[0]) # n не требуется
  k = int(lst[1])
  words = input_file.readline().split()

❹ line = ''
  chars_on_line = 0
```

```
    for word in words:
    ❺ if chars_on_line + len(word) <= k:
            line = line + word + ' '
            chars_on_line = chars_on_line + len(word)
        else:
        ❻ output_file.write(line[:-1] + '\n')
            line = word + ' '
            chars_on_line = len(word)

❼ output_file.write(line[:-1] + '\n')

    input_file.close()
    output_file.close()
```

Сперва открываем входной файл ❶ и выходной файл ❷. Обратите внимание на режимы: мы открываем входной файл в режиме `r` (для чтения), а выходной файл — в режиме `w` (для записи). Можно было бы открыть выходной файл немного позже, непосредственно перед его использованием, но я решил открыть оба в начале, чтобы упростить организацию программы. Аналогично мы могли бы закрыть файл, как только он станет не нужен, но в этой книге я решил закрывать все файлы одновременно в конце программы. Для долго работающих программ, оперирующих множеством файлов, следует держать файлы открытыми только тогда, когда это необходимо.

Далее мы читаем первую строку входного файла ❸. Она содержит два целых числа, разделенных пробелами: `n` — количество слов и `k` — максимальное количество разрешенных символов (не считая пробелов) в строке. Как всегда при чтении значений, разделенных пробелами, мы используем метод `split`. Затем читаем вторую строку, содержащую текст эссе. Снова применяем `split`, чтобы разбить ее на список слов. С входными данными разобрались.

В основной части программы есть две ключевые переменные: `line` и `chars_on_line`. Переменная `line` — это текущая строка, и изначально мы в нее помещаем пустую строку ❹. В переменной `chars_on_line` хранится число символов в текущей строке, не считая пробелов.

Вы можете спросить: а зачем вообще нам переменная `chars_on_line`? Не проще ли использовать функцию `len(line)`? Можно, но тогда мы бы считали и пробелы, а их учитывать нельзя. Да, это легко исправить, вычтя количество пробелов, но применить этот метод можете попробовать самостоятельно, если сочтете такое решение более интуитивным, чем хранение числа символов в переменной `chars_on_line`.

Теперь пора перебрать все слова. Нужно определить, куда поместить каждое слово — в текущую строку или в следующую.

Если количество непробельных символов в текущей строке плюс количество символов в слове не превышает `k`, то оно помещается в эту строку ❺. При этом мы добавляем слово и пробел перед ним в текущую строку и пересчитываем количество непробельных символов в ней.

В противном случае слово не помещается в текущей строке. Значит, с ней мы закончили, записываем ее в выходной файл ❻ и обновляем переменные `line` и `chars_on_line`, чтобы учесть, что теперь это единственное слово в текущей строке.

И вот тут следует отметить два момента, касающихся вызова `write` ❻. Во-первых, мы используем срез `[: -1]`, чтобы в файл не выводился пробел, следующий за последним словом в строке. Во-вторых, возможно, вы ожидали, что мы применим форматированные строки:

```
output_file.write(f'{line[:-1]}\n')
```

Но на момент написания этой книги на сайте USACO используется старая версия Python, не поддерживающая форматированные строки.

Зачем мы выводим переменную `line` еще раз после окончания цикла ❼? Дело в том, что после каждой итерации цикла `for` в переменной `line` гарантированно оказывается одно или несколько слов, которые мы еще не вывели. Подумайте, что происходит с каждым словом, которое мы обрабатываем. Если оно умещается в текущей строке, мы ничего не выводим. А если не помещается, выводим текущую строку, но не слово в следующей строке. Поэтому нам необходимо вывести строку в выходной файл *после* цикла ❼, иначе она будет потеряна.

Последнее, что мы делаем, — закрываем оба файла.

Неприятная особенность записи в файл вместо вывода на экран заключается в том, что мы не видим вывод при запуске программы. Чтобы увидеть результат, приходится открывать выходной файл в текстовом редакторе.

Поэтому дам совет: разрабатывайте программу, используя вызовы печати, а не записи, чтобы все написанное выводилось на экран. Это позволит упростить поиск ошибок в программе и избежать необходимости переключаться между кодом и файлом. Когда будете довольны кодом, выполните запись в файл. Затем обязательно протестируйте код дополнительно, чтобы убедиться, что все попадает в файл так, как и должно.

Код готов к отправке на сайт USACO. Все тесты должны пройти.

Задача 17. Посевная на ферме

С помощью цикла мы можем прочитать из файла указанное количество строк. В этой задаче мы так и поступим и в процессе увидим, что этот механизм похож на использование цикла с функцией `input` для чтения из стандартного ввода.

В главе 6, в задаче «Фигурки», мы узнали, как выполнять нисходящее проектирование с применением функций. Составление нескольких функций для решения задачи — важный навык. И поскольку о файлах я рассказал все, что хотел, теперь я выбрал задачу, на которой можно поупражняться в нисходящем проектировании.

Это сложная задача. Сначала нам нужно будет в точности сделать все, о чем нас просят. Затем потребуется разработать решение задачи, хорошенько подумав, правильное ли оно.

Задача с конкурса USACO 2019 February Bronze Contest под названием The Great Revegetation.

Постановка задачи

У фермера Джона есть n пастбищ, и он хотел бы их все засеять травой. Пастбища пронумерованы от 1 до n.

У фермера Джона есть четыре разных типа семян травы с номерами 1, 2, 3 и 4. Для каждого пастбища будет выбран один из них.

А еще у фермера Джона есть m коров. У каждой из них два любимых пастбища, на которых она ест траву. Каждую корову интересуют только два ее любимых пастбища. Для здорового питания каждой буренке необходимо, чтобы на двух любимых пастбищах росли разные виды травы. Например, для некоторой коровы было бы идеально, если бы на одном из ее пастбищ была трава 1-го типа, а на другом — 4-го. А если на обоих будет трава одного типа — это очень плохо, нас это не устраивает.

Пастбище может быть любимым более чем у одной коровы. Но таких животных для одного пастбища не более трех.

Определите сорт травы для каждого пастбища. На каждом из них необходимо использовать траву от 1-го до 4-го типа, и два любимых пастбища каждой коровы должны иметь разные типы травы.

Входные данные

Входные данные находятся в файле с именем `revegetate.in`. Они состоят:

- из строки, содержащей два целых числа, разделенных пробелом. Первое целое число, `n`, — это количество пастбищ, обозначенное цифрами от 2 до 100. Второе целое число, `m`, — количество коров в диапазоне от 1 до 150;
- m строк, каждая из которых содержит два любимых номера пастбища данной коровы. Номера пастбищ представляют собой целые числа от 1 до n, разделенные пробелом.

Выходные данные

Выходные данные записываются в файл с именем `revegetate.out`.

Выведите подходящий способ засевания пастбищ. Результатом будет строка из n символов `'1'`, `'2'`, `'3'` или `'4'`. Первая цифра — трава для пастбища 1, вторая — для

пастбища 2 и т. д. Можно интерпретировать эти *n* символов как целое число с *n* цифрами. Например, если мы получили ответ `'11123'`, то можем вывести целое число 11 123.

Целочисленная интерпретация может использоваться, если есть выбор, что выводить. Если существует несколько подходящих способов засеять пастбища, мы должны вывести тот, который в целочисленной интерпретации является наименьшим. Например, если допустимы варианты `'11123'` и `'22123'`, выводим строку `'11123'`, потому что 11 123 меньше 22 123.

Тестовый пример

Мы хотели найти решение этой задачи с помощью нисходящего проектирования. Тестовый пример поможет разобраться:

```
8 6
5 4
2 4
3 5
4 1
2 1
5 2
```

Первая строка тестового примера говорит о том, что у фермера восемь пастбищ. Они пронумерованы цифрами с 1 до 8. Первая строка также показывает, что коров шесть. В условии про нумерацию коров ничего не сказано, поэтому я начну ее с 0. Два любимых пастбища каждой коровы указаны в табл. 7.1.

Таблица 7.1. Предпочтения коров

Корова	Пастбище 1	Пастбище 2
0	5	4
1	2	4
2	3	5
3	4	1
4	2	1
5	5	2

В этой задаче нас просят принять *n* решений. Какой сорт травы использовать для пастбища 1? А для пастбища 2? Ну и так далее. Одна из стратегий разрешения таких задач — принимать одно решение за раз, ничего не делая, если возникнет ошибка. Если решение *n* без ошибок, значит, общее решение должно быть правильным.

Давайте пройдемся по пастбищам с 1-го по 8-е и посмотрим, сможем ли мы назначить каждому из них тип травы. Первоочередное внимание нужно будет уделить типам травы с меньшим числом, чтобы в целочисленном виде ответ получился наименьшим.

Какой сорт травы выбрать для пастбища 1? Его любят коровы 3 и 4, поэтому их и рассмотрим. Если бы мы уже выбрали типы травы для некоторых пастбищ этих коров, то нам пришлось бы осторожно выбирать ее для пастбища 1. Нельзя назначать корове два пастбища с одинаковым типом травы, потому что это против правил! Мы еще не выбрали ни одного типа травы, поэтому все может пойти не так, как надо, независимо от того, что мы выберем для пастбища 1. Поскольку нас интересуют значения поменьше, выберем тип травы 1.

Все решения по типу травы я занесу в таблицу. Вот решение, которое мы только что приняли, выбрав тип травы 1 для пастбища 1.

Пастбище	Тип травы
1	1

Поехали дальше. Какой сорт травы выбрать для пастбища 2? Его любят коровы 1, 4 и 5, поэтому сосредоточимся на них. Одно из пастбищ коровы 4 — номер 1, мы выбрали для него траву типа 1, поэтому она для пастбища 2 не подходит. Если бы мы использовали траву 1 для пастбища 2, то у коровы 4 получилось бы два пастбища с одинаковым типом травы, а это нарушит правила. Однако коровам 1 и 5 пока что все равно. Поэтому выбираем тип травы 2, наименьший из доступных. Получаем следующее.

Пастбище	Тип травы
1	1
2	2

Какой сорт травы выбрать для пастбища 3? Единственная корова, которая любит его, — это корова 2. У нее пастбища 3 и 5. Ограничений для этой коровы пока нет, потому что мы не присвоили тип травы пастбищу 5, поэтому возьмем для пастбища 3 тип травы 1. Получаем следующее.

Пастбище	Тип травы
1	1
2	2
3	1

Мало-помалу начинают вырисовываться три задачи. Во-первых, нам нужно узнать, каким коровам нравится текущее пастбище. Во-вторых, необходимо определить, какие виды травы для этих коров исключить из рассмотрения. В-третьих, следует выбрать из доступных типов травы тот, что с наименьшим номером. Это может быть наше решение.

Продолжим. У нас есть три коровы, которые любят пастбище 4: коровы 0, 1 и 3. Для коровы 0 не исключен ни один из типов травы, так как их мы еще не назначали. Для коровы 1 исключена трава 2-го типа, а для коровы 3 — 1-го типа, так как их мы присвоили другим пастбищам для тех же коров. Таким образом, наименьший доступный тип травы — 3, поэтому именно его мы и возьмем для пастбища 4.

Пастбище	Тип травы
1	1
2	2
3	1
4	3

Теперь пастбище 5. К нему относятся коровы 0, 2 и 5. Для коровы 0 исключена трава типа 3, для коровы 2 — типа 1 и для коровы 5 — типа 2. Таким образом, травы 1, 2 и 3-го типов брать нельзя. Остается только тип 4.

Еле справились! У нас почти закончились виды травы. К счастью, пастбище 5 больше никому не нужно и тип 4 не придется исключать.

Или стоп... Все так и должно быть, потому что в описании задачи сказано, что каждое пастбище любимо не более чем тремя коровами. Это означает, что для каждого пастбища можно исключить не более трех видов травы. То есть такой ситуации и не будет! Поэтому нам не нужно беспокоиться о влиянии принятых ранее решений на текущее. Независимо от того, что мы делали в прошлом, у нас всегда будет хотя бы один доступный тип травы.

Добавим пастбище 5 в таблицу.

Пастбище	Тип травы
1	1
2	2
3	1
4	3
5	4

Осталось рассмотреть три пастбища. Они не интересны ни одной корове, поэтому можно просто использовать для них траву типа 1. Это дает нам следующее.

Пастбище	Тип травы
1	1
2	2
3	1
4	3
5	4
6	1
7	1
8	1

Прочитав типы травы сверху вниз, мы можем сформировать правильный ответ. Результат выглядит следующим образом:

```
12134111
```

Нисходящее проектирование

Теперь, когда мы понимаем, какие задачи предстоит реализовать, начнем решать их сверху вниз.

Верхний уровень

В предыдущем разделе, работая с тестовым примером, мы выделили три подзадачи. Прежде чем программа сможет решить любую из них, нужно прочитать входные данные, и это — четвертая подзадача. А еще нужно вывести результат. На него тоже потребуется несколько строк кода, так что давайте назовем это пятой подзадачей.

Итого, пять основных подзадач.

1. Прочитать входные данные.
2. Определить, какие коровы любят текущее пастбище.
3. Понять, какие виды травы использовать нельзя.
4. Выбрать для текущего пастбища тип травы с наименьшим номером.
5. Вывести результат.

Как и при решении задачи «Фигурки» в главе 6, начнем с комментариев TODO, а затем будем удалять ненужные TODO по мере их решения.

Напишем комментарии. Поскольку в начале нам нужно открывать файлы, а в конце закрывать их, это я тоже добавил в код.

Вот с чего мы начинаем:

```
# Основная программа

input_file = open('revegetate.in', 'r')
output_file = open('revegetate.out', 'w')

# TODO: Чтение входных данных

# TODO: Определяем, каким коровам нравится это пастбище

# TODO: Исключаем типы травы для пастбища

# TODO: Выбираем траву с наименьшим номером

# TODO: Выводим результат

input_file.close()
output_file.close()
```

Подзадача 1. Прочитать входные данные

В первой строке входных данных мы считаем числа n и m. Это довольно просто, и функция для этого не нужна, поэтому сделаем все напрямую. Затем нужно будет считать информацию о пастбищах для m коров, и здесь функция уже не помешает. Удалим TODO в комментарии Чтение входных данных, обработаем первую строку входных данных и вызовем функцию read_cows, которую мы скоро напишем:

```
# Основная программа

input_file = open('revegetate.in', 'r')
output_file = open('revegetate.out', 'w')

# Чтение входных данных
lst = input_file.readline().split()
num_pastures = int(lst[0])
num_cows = int(lst[1])
❶ favorites = read_cows(input_file, num_cows)

# TODO: Определяем, каким коровам нравится это пастбище
```

```
  # TODO: Исключаем типы травы для пастбища

  # TODO: Выбираем траву с наименьшим номером

  # TODO: Выводим результат

  input_file.close()
  output_file.close()
```

Функция `read_cows`, которую мы вызываем ❶, берет уже открытый для чтения файл и будет считывать два любимых пастбища каждой коровы. Затем функция вернет список списков, где каждый внутренний список будет содержать два номера пастбища для данной коровы. Код:

```
def read_cows(input_file, num_cows):
    """
    input_file – это открытый для чтения файл
    с информацией о коровах.
    num_cows – это количество коров в файле.

    Требуется прочитать число коров из файла
    и информацию о них.
    Возвращает список двух любимых пастбищ коровы.
    """

    favorites = []
    for i in range(num_cows):
     ❶ lst = input_file.readline().split()
        lst[0] = int(lst[0])
        lst[1] = int(lst[1])
     ❷ favorites.append(lst)
    return favorites
```

Функция добавляет любимые пастбища коров в список `favorites`. Для этого используется цикл `range for`, который повторяется `num_cows` раз, по одному разу для каждой коровы. Цикл необходим, так как количество строк, которые нужно считать, зависит от количества коров в файле.

На каждой итерации цикла мы читаем следующую строку и разбиваем ее на две части ❶. Затем с помощью функции `int` преобразуем компоненты из строк в целые числа. Затем добавляем список из двух целых чисел в `favorites` ❷.

Последнее, что мы делаем, — это возвращаем список любимых пастбищ.

Прежде чем продолжить, давайте убедимся, что правильно вызываем эту функцию. Попробуем вызвать ее самостоятельно, отдельно от более крупной программы, которую мы создаем. Такие функции полезно тестировать, чтобы исправить ошибки, которые могут возникнуть в процессе разработки.

С помощью текстового редактора создайте файл `revegetate.in` со следующим содержимым (оно такое же, как в тестовом примере, изученном ранее):

```
8 6
5 4
2 4
3 5
4 1
2 1
5 2
```

Теперь в оболочке Python введите код функции `read_cows`. Вот как вызвать ее:

```
  >>> input_file = open('revegetate.in', 'r')
❶ >>> input_file.readline()
  '8 6\n'
❷ >>> read_cows(input_file, 6)
    [[5, 4], [2, 4], [3, 5], [4, 1], [2, 1], [5, 2]]
```

Функция `read_cows` читает только информацию о коровах. Поскольку мы тестируем ее изолированно, отдельно от основной программы, нам нужно прочитать первую строку файла, прежде чем вызывать функцию ❶. Вызвав функцию `read_cows`, мы получим список любимых пастбищ каждой коровы. Обратите внимание на то, что мы передаем `read_cows` объект открытого файла, а *не* его имя ❷.

Обязательно добавьте функцию `read_cows` и другие функции в начало программы, до комментария `# Основная программа`. Пора перейти к подзадаче 2.

Подзадача 2. Определить коров

Стратегия решения этой задачи заключается в том, что мы поочередно рассматриваем каждое пастбище, чтобы решить, какой тип травы применять. Выполняем эту работу внутри цикла, каждая его итерация отвечает за засевание одного пастбища. Для каждого пастбища нужно определить коров, которым оно нравится, исключить используемые типы травы и выбрать из доступных тип травы с наименьшим номером. Эти три задачи должны выполняться для каждого пастбища, поэтому сделаем отступ внутри цикла.

Напишем функцию `cows_with_favorite`, возвращающую коров, которым нравится текущее пастбище.

Пока что получаем вот такой текст основной программы:

```
# Основная программа

input_file = open('revegetate.in', 'r')
output_file = open('revegetate.out', 'w')
```

```
# Чтение входных данных
lst = input_file.readline().split()
num_pastures = int(lst[0])
num_cows = int(lst[1])
favorites = read_cows(input_file, num_cows)

for i in range(1, num_pastures + 1):

    # Определяем, каким коровам нравится это пастбище
 ❶  cows = cows_with_favorite(favorites, i)

    # TODO: Исключаем типы травы для пастбища

    # TODO: Выбираем траву с наименьшим номером

# TODO: Выводим результат

input_file.close()
output_file.close()
```

Функция `cows_with_favorite`, которую мы вызываем в строке ❶, берет список любимых пастбищ коров и номер пастбища и возвращает коров, которые любят данное пастбище. Код функции:

```
def cows_with_favorite(favorites, pasture):
    """
    favorites — это список любимых пастбищ, получаемый
    из функции read_cows. pasture — это номер пастбища.

    Возвращает список коров, которые любят это пастбище.
    """
    cows = []
    for i in range(len(favorites)):
        if favorites[i][0] == pasture or favorites[i][1] == pasture:
            cows.append(i)
    return cows
```

Функция перебирает список `favorites` и ищет в нем коров, которым нравится пастбище. Каждая корова, которая любит это пастбище, добавляется в список коров, который в итоге возвращается.

Проведем небольшой тест. Вызовите функцию `cows_with_favorite` в оболочке Python, вот так:

```
>>> cows_with_favorite([[5, 4], [2, 4], [3, 5]], 5)
```

У нас есть три коровы, и мы спрашиваем, какие из них любят пастбище 5. Это коровы 0 и 2, что функция нам и сообщает:

```
[0, 2]
```

Подзадача 3. Исключение недоступных типов травы

Теперь мы знаем, каким коровам нравится текущее пастбище. Следующий шаг — выяснить, какие виды травы для этих коров нужно исключить из рассмотрения для данного пастбища. Исключить нужно виды травы, уже примененные для одной или нескольких коров. Мы напишем функцию `types_used`, возвращающую типы травы, которые уже использовались и которые, соответственно, больше брать нельзя.

Далее приведена основная программа, обновленная вызовом этой функции:

```
# Основная программа

input_file = open('revegetate.in', 'r')
output_file = open('revegetate.out', 'w')

# Чтение входных данных
lst = input_file.readline().split()
num_pastures = int(lst[0])
num_cows = int(lst[1])
favorites = read_cows(input_file, num_cows)

❶ pasture_types = [0]

for i in range(1, num_pastures + 1):

    # Определяем, каким коровам нравится это пастбище
    cows = cows_with_favorite(favorites, i)

    # Исключаем типы травы для пастбища
  ❷ eliminated = types_used(favorites, cows, pasture_types)

    # TODO: Выбираем траву с наименьшим номером

# TODO: Выводим результат

input_file.close()
output_file.close()
```

Помимо вызова функции `types_used` ❷, я добавил переменную `pasture_types` ❶. Это список, в котором будет храниться тип травы для каждого пастбища.

Напомню, что пастбища пронумерованы начиная с 1. Однако списки Python индексируются с 0. Это несоответствие вызывает неудобство, и если бы мы просто начали добавлять типы травы в `pasture_types`, то тип травы для пастбища 1 попал бы на индекс `0`, тип травы для пастбища 2 — на индекс `1` и т. д. Поэтому я добавил в начало списка ❶ число `0`, чтобы тип травы для пастбища 1 соответствовал индексу `1`.

Предположим, мы определили типы травы для первых четырех пастбищ. Список `pasture_types` в данный момент выглядит так:

```
[0, 1, 2, 1, 3]
```

Теперь тип травы для пастбища 1 находится на индексе `1`, тип для пастбища 2 — на индексе `2` и т. д. А тип травы для пастбища 5? Неизвестно, ведь его мы еще не объявили. Если длина списка `pasture_types` равна 5, это означает, что мы определили типы травы только для первых четырех пастбищ. Количество типов травы, которое нам известно, всегда на единицу меньше длины списка.

Перейдем к функции `types_used`. Она принимает три параметра: список любимых пастбищ каждой коровы, список коров, которые любят текущее пастбище, и уже выбранные типы травы. Функция возвращает список типов травы, которые уже задействуются и, следовательно, исключены для текущего пастбища. Вот она:

```
def types_used(favorites, cows, pasture_types):
    """
    favorites — это список любимых пастбищ, получаемый
    из функции read_cows.
    cows — это список коров.
    pasture_types — это список типов травы.

    Возвращает список типов травы, уже "занятых" коровами.
    """

    used = []
    for cow in cows:
        pasture_a = favorites[cow][0]
        pasture_b = favorites[cow][1]
     ❶ if pasture_a < len(pasture_types):
            used.append(pasture_types[pasture_a])
     ❷ if pasture_b < len(pasture_types):
            used.append(pasture_types[pasture_b])
    return used
```

У каждой коровы есть два любимых пастбища, которые я назвал `pasture_a` и `pasture_b`. Проверяем, выбран ли для каждого из них сорт трав в строках ❶ и ❷. Если он уже выбран, пастбище находится в списке в `pasture_types`. Все эти типы травы добавляются в список, который функция возвращает, проанализировав всех коров.

А что будет делать код, если несколько коров любят одно и то же? Давайте передадим функции простой тестовый пример и ответим на этот вопрос.

Вызовите функцию `types_used` в оболочке Python. А теперь подумаем, что она вернет:

```
>>> types_used([[5, 4], [2, 4], [3, 5]], [0, 1], [0, 1, 2, 1, 3])
```

Здесь нужно крепко подумать. Первый аргумент — это любимые пастбища трех коров. Второй аргумент говорит, что коровы 0 и 1 любят первое пастбище. В третьем аргументе передаются типы травы, которые мы выбрали ранее.

Какие виды травы уже используются и, следовательно, исключаются для коров 0 и 1? Корова 0 любит пастбище 4, где растет трава 3-го типа, поэтому она исключается. Корова 1 любит пастбище 2, поэтому трава 2-го типа, которая там задействована, исключается. Корова 1 любит пастбище 4, но мы уже знаем по корове 0, что трава 3 исключена.

Возвращаемое значение функции следующее:

```
[3, 2, 3]
```

Получились две тройки — одна от коровы 0, другая от коровы 1.

Может показаться, что ответ с одной тройкой выглядел бы лучше, но и наличие дубликата вполне нормально. Если тип травы есть в этом списке, то он исключается независимо от того, попадет он в список один, два или три раза.

Подзадача 4. Выбираем тип травы с наименьшим номером

Получив исключенные типы травы, мы можем перейти к следующей задаче: выбрать для текущего пастбища возможный тип травы с наименьшим номером. Чтобы решить эту задачу, вызовем новую функцию `smallest_available`. Она вернет тип травы, который можно использовать для текущего пастбища.

Далее приведена основная программа с вызовом функции `smallest_available`:

```
# Основная программа

input_file = open('revegetate.in', 'r')
output_file = open('revegetate.out', 'w')

# Чтение входных данных
lst = input_file.readline().split()
num_pastures = int(lst[0])
num_cows = int(lst[1])
favorites = read_cows(input_file, num_cows)

pasture_types = [0]

for i in range(1, num_pastures + 1):

    # Определяем, каким коровам нравится это пастбище
    cows = cows_with_favorite(favorites, i)
```

```
    # Исключаем типы травы для пастбища
    eliminated = types_used(favorites, cows, pasture_types)

    # Выбираем траву с наименьшим номером
❶   pasture_type = smallest_available(eliminated)
❷   pasture_types.append(pasture_type)

# TODO: Выводим результат

input_file.close()
output_file.close()
```

Получив наименьший тип травы для текущего пастбища ❶, добавляем его в список выбранных типов травы ❷.

А вот и сама функция `smallest_available`:

```
def smallest_available(used):
    """
    used – это список использованных типов травы.

    Возвращает наименьший номер еще
    не задействованной травы.
    """
    grass_type = 1
    while grass_type in used:
        grass_type = grass_type + 1
    return grass_type
```

Функция начинает работу с типа травы 1. Затем она выполняет цикл до тех пор, пока не найдет еще не использованный тип травы, на каждой итерации увеличивая его на единицу. Как только находится свободный тип травы, функция возвращает его. Мы помним, что по меньшей мере один тип травы всегда доступен, поэтому функция обязательно вернет результат.

Подзадача 5. Запись вывода

Собственно, нужный ответ уже у нас в руках, в списке `pasture_types`! Осталось лишь вывести его. Покажем код программы в последний раз:

```
# Основная программа

input_file = open('revegetate.in', 'r')
output_file = open('revegetate.out', 'w')

# Чтение входных данных
lst = input_file.readline().split()
```

```
    num_pastures = int(lst[0])
    num_cows = int(lst[1])
    favorites = read_cows(input_file, num_cows)

    pasture_types = [0]

    for i in range(1, num_pastures + 1):

        # Определяем, каким коровам нравится
        # это пастбище
        cows = cows_with_favorite(favorites, i)

        # Исключаем типы травы для пастбища
        eliminated = types_used(favorites, cows, pasture_types)

        # Выбираем траву с наименьшим номером
        pasture_type = smallest_available(eliminated)
        pasture_types.append(pasture_type)

    # Выводим результат
❶   pasture_types.pop(0)
❷   write_pastures(output_file, pasture_types)

    input_file.close()
    output_file.close()
```

Перед записью вывода удаляем фиктивный `0` в начале списка `pasture_types` ❶. Выводить его мы не хотим, так как это не настоящий тип травы. Затем вызываем функцию `write_pastures`, которая выводит результат ❷.

Осталось написать саму функцию `write_pastures`. Она берет файл, открытый для записи, сформированный список типов травы и выводит типы травы в файл. Ее код:

```
def write_pastures(output_file, pasture_types):
    """
    output_file — это файл, в который будет записан
    результат.
    pasture_types — это список типов травы.

    Функция выводит pasture_types в output_file.
    """
    pasture_types_str = []
❶   for pasture_type in pasture_types:
        pasture_types_str.append(str(pasture_type))
❷   output = ''.join(pasture_types_str)
❸   output_file.write(output + '\n')
```

Сейчас `pasture_types` — это список целых чисел. Но уже через секунду мы поймем, что здесь удобнее работать со списком строк, поэтому создадим новый список, превратив числа в строки ❶. Я не изменяю сам список `pasture_types`, потому что это некорректно с точки зрения вызывающего кода, так как он ожидает лишь записи вывода `output_file`, а не того, что список `pasture_types` окажется изменен. Функция вывода данных не должна менять передаваемые ей списки.

Для вывода нужно вызвать функцию `write`, передав ей строку, а не список, а в этой строке не должно быть пробелов. Здесь замечательно работает метод `join`. Как мы узнали в главе 5, строка, на которой вызывается метод `join`, служит разделителем между значениями в списке. Поскольку нам не нужен разделитель, используем пустую строку ❷. Метод `join` работает только со списком строк, поэтому я преобразовал список целых чисел в список строк в начале функции ❶.

Получив выходные данные в нужном формате, записываем их в файл ❸.

Собираем все вместе

Полная программа приведена в листинге 7.2.

Листинг 7.2. Решение задачи «Посевная на ферме»

```
def read_cows(input_file, num_cows):
    """
    input_file – это открытый для чтения файл
    с информацией о коровах.
    num_cows – это количество коров в файле.

    Требуется прочитать число коров из файла
    и информацию о них.
    Возвращает список двух любимых пастбищ коровы.
    """
    favorites = []
    for i in range(num_cows):
        lst = input_file.readline().split()
        lst[0] = int(lst[0])
        lst[1] = int(lst[1])
        favorites.append(lst)
    return favorites

def cows_with_favorite(favorites, pasture):
    """
    favorites – это список любимых пастбищ, получаемый
    из функции read_cows.
    pasture – это номер пастбища.
```

```
    Возвращает список коров, которые любят это пастбище.
    """

    cows = []
    for i in range(len(favorites)):
        if favorites[i][0] == pasture or favorites[i][1] == pasture:
            cows.append(i)
    return cows

def types_used(favorites, cows, pasture_types):
    """
    favorites — это список любимых пастбищ, получаемый
    из функции read_cows.
    cows — это список коров.
    pasture_types — это список типов травы.

    Возвращает список типов травы, уже "занятых" коровами.
    """

    used = []
    for cow in cows:
        pasture_a = favorites[cow][0]
        pasture_b = favorites[cow][1]
        if pasture_a < len(pasture_types):
            used.append(pasture_types[pasture_a])
        if pasture_b < len(pasture_types):
            used.append(pasture_types[pasture_b])
    return used

def smallest_available(used):
    """
    used — это список использованных типов травы.

    Возвращает наименьший номер еще не тронутой травы.
    """

    grass_type = 1
    while grass_type in used:
        grass_type = grass_type + 1
    return grass_type

def write_pastures(output_file, pasture_types):
    """
    output_file — это файл, в который будет записан результат.
    pasture_types — это список типов травы.

    Функция выводит pasture_types в output_file.
    """
```

```
    pasture_types_str = []
    for pasture_type in pasture_types:
        pasture_types_str.append(str(pasture_type))
    output = ''.join(pasture_types_str)
    output_file.write(output + '\n')

# Основная программа

input_file = open('revegetate.in', 'r')
output_file = open('revegetate.out', 'w')

# Чтение входных данных
lst = input_file.readline().split()
num_pastures = int(lst[0])
num_cows = int(lst[1])
favorites = read_cows(input_file, num_cows)

pasture_types = [0]

for i in range(1, num_pastures + 1):

    # Определяем, каким коровам нравится это пастбище
    cows = cows_with_favorite(favorites, i)

    # Исключаем типы травы для пастбища
    eliminated = types_used(favorites, cows, pasture_types)

    # Выбираем траву с наименьшим номером
    pasture_type = smallest_available(eliminated)
    pasture_types.append(pasture_type)

# Выводим результат
pasture_types.pop(0)
write_pastures(output_file, pasture_types)

input_file.close()
output_file.close()
```

Мы справились! Это была сложная задача, и решили мы ее с помощью нисходящего проектирования. Отправьте работу на сайт USACO.

Читая задачу впервые, легко зайти в тупик. Но повторюсь, что вам не нужно решать все одним огромным шагом. Разбейте ее на части, решите каждую из них — это сделать проще, и так получите решение задачи в целом. Вы добились огромных успехов в изучении Python и теперь способны разрабатывать программы и решать задачи. Все эти проблемы вам по силам!

ПРОВЕРИМ ЗНАНИЯ

Давайте подумаем о новой версии «Посевная на ферме», в которой не было бы ограничений на количество коров, любящих определенное пастбище. То есть у каждого пастбища их может быть четыре, пять или больше. При этом по-прежнему не разрешается назначать корове два пастбища с одинаковым типом травы.

Предположим, что мы решаем новую версию задачи и у нас есть тестовый пример, в котором пастбище является любимым у более чем трех коров. Какое из утверждений для тестового примера верно?

А. *Гарантируется*, что решить задачу, используя всего четыре типа травы, *нельзя*.

Б. Возможно, есть способ решить эту задачу. И если он есть, то, может быть, код из листинга 7.2 с этим справится.

В. Возможно, есть способ решить эту задачу. Если есть, то код из листинга 7.2 с этим точно справится.

Г. Возможно, есть способ решить эту задачу. Если он есть, то код из листинга 7.2 точно не сработает.

Ответ: **Б**. Можно найти тестовый пример, который правильно решается нашей программой, и такой, который можно решить, но не нашей программой. Соответственно, ответы **А**, **В** и **Г** исключаются.

Далее приведен тестовый пример, который наша программа решит правильно:

```
2 4
1 2
1 2
1 2
1 2
```

Каждое пастбище является любимым у четырех коров. Мы можем решить этот тестовый пример, используя лишь два типа травы. Вы можете убедиться в этом, попробовав выполнить пример.

А этот пример наша программа решить не сможет:

```
6 10
2 3
2 4
3 4
```

```
2 5
3 5
4 5
1 6
3 6
4 6
5 6
```

Программа по ошибке назначает пастбищу 1 тип травы 1, поэтому приходится для пастбища 6 использовать тип 5, а так нельзя. Наша программа не сработает, но это не значит, что пример неразрешим. В данном случае пастбищу 1 можно назначить тип 2, и после этого решение найдется. Здесь потребуется более сложная программа, можете попробовать создать ее сами.

Резюме

В этой главе вы узнали, как открывать, читать, записывать и закрывать файлы. Файлы полезны, если нужно сохранить информацию и позже использовать ее в качестве входных данных. Еще их можно задействовать для передачи информации пользователям. Вы также узнали, что файлы можно обрабатывать так же, как стандартный ввод и стандартный вывод.

В следующей главе узнаете, как можно хранить значения в множествах и словарях Python. Вы уже умеете хранить наборы значений в списках, но увидите, что множества и словари способны упростить решение некоторых категорий задач.

Упражнения

Далее приведены несколько упражнений, которые вы можете попробовать выполнить. Все они находятся на сайте USACO и связаны с чтением из файлов и записью в файлы. Заодно повторите материалы предыдущих глав.

1. USACO, задача Mixing Milk, 2018 December Bronze Contest.
2. USACO, задача Why Did the Cow Cross the Road, 2017 February Bronze Contest.
3. USACO, задача The Lost Cow, 2017 US Open Bronze Contest.
4. USACO, задача Cow Gymnastics, 2019 December Bronze Contest.
5. USACO, задача Bovine Genomics, 2017 US Open Bronze Contest.
6. USACO, задача Team Tic Tac Toe, 2018 US Open Bronze Contest.
7. USACO, задача Sleepy Cow Herding, 2019 February Bronze Contest.

Примечания

Задача «Форматирование эссе» взята с USACO 2020 January Bronze Contest. Задача «Посевная на ферме» — с USACO 2019 February Bronze Contest.

Помимо текстовых файлов существует множество других типов. Возможно, вам захочется поработать с файлами HTML, электронными таблицами Excel, файлами PDF, документами Word или файлами изображений. Python может все! Если интересна эта тема, почитайте книгу Ала Звейгарта *Automate the Boring Stuff with Python*, 2-е издание (No Starch Press, 2019).

Строка *perhaps better poetry* позаимствована у Дж. К. Р. Ликлидера, это цитата из книги *Computers and the World of the Future* под редакцией Мартина Гринбергера (MIT Press, 1962). Вот ее перевод: «Но некоторые люди пишут стихи на том языке, на котором мы говорим. Возможно, компьютеры когда-нибудь напишут стихи, превосходящие все то, что до сих пор написано на английском».

8

Организация данных с помощью множеств и словарей

Списки Python полезны, когда нам требуется хранить последовательность значений, например высоту фигурок или слова из эссе. Списки позволяют упорядочивать значения и обращаться к ним по индексу, однако, как вы увидите в этой главе, есть операции, для которых списки не оптимизированы, например определение того, находится ли конкретное значение в коллекции, или создание логической связи между парами значений.

В этой главе мы узнаем о множествах и словарях Python — новых для нас структурах, которые, как и списки, можно использовать для хранения коллекций значений. Мы увидим, что множества предпочтительнее, когда нужны определенные значения и неважен их порядок, а словари удобны, когда требуется работать с парами значений.

С помощью новых коллекций мы решим три задачи: определение количества уникальных адресов электронной почты, поиск общих слов в списках слов и определение количества пар городов и штатов.

Задача 18. Адреса электронной почты

В этой задаче мы организуем хранение адресов электронной почты. Нас не волнует, сколько раз встретится каждый адрес электронной почты, и порядок адресов тоже неважен. Такие упрощенные требования к хранению означают, что мы можем отказаться от использования списков и взять множества Python, которые здесь побеждают с огромным отрывом. Изучим множества как следует.

Задача с сайта DMOJ, код `ecoo19r2p1`.

Постановка задачи

Знали ли вы, что на Gmail существует много способов записать адрес электронной почты?

Можно взять любой адрес Gmail, добавить к нему символ плюса (+) и строку перед символом @, и на исходный адрес будут идти все письма, отправленные на измененный адрес. В адресах электронной почты Gmail все символы между + и @ игнорируются. Например, мой почтовый адрес daniel.zingaro@gmail.com, но это лишь один способ записать его. Если вы отправите электронное письмо на daniel.zingaro+book@gmail.com или daniel.zingaro+hi.there@gmail.com, оно тоже придет мне (можете написать, кстати).

Точки перед символом @ игнорируются. Например, если вы отправите электронные письма на адреса danielzingaro@gmail.com, daniel..zingaro@gmail.com (две точки подряд), da.nielz.in.gar.o..@gmail.com (много точек) и т. д., я их все тоже получу.

И последнее: различия в верхнем и нижнем регистрах во всем адресе игнорируются. Наверное вы уже поняли, что Daniel.Zingaro@gmail.com, DAnIELZIngARO+Flurry@gmAIL.COM и т. д. — одинаковые адреса.

В этой задаче будут даны адреса электронной почты и потребуется определить, сколько среди них уникальных. Правила распознавания адресов электронной почты в этой задаче такие же, как и у Gmail: символы от + до символа перед @ игнорируются, точки перед @ игнорируются, регистр во всем адресе игнорируется.

Входные данные

Входные данные состоят из десяти тестовых примеров. Каждый из них содержит:

- строку с целым числом n — количеством адресов электронной почты от 1 до 100 000;
- n строк, каждая из которых содержит адрес электронной почты. Все адреса содержат как минимум один символ перед @ и хотя бы один символ после @. До символа @ могут стоять буквы, цифры, точки и плюсы. После него допустимы буквы, цифры и точки.

Выходные данные

Для каждого тестового примера выведите количество уникальных адресов электронной почты.

Лимит времени на решение составляет 30 секунд.

Использование списков

Мы прочитали уже семь глав книги. В каждой из них я поставил задачу, а затем вы изучали функции Python, которые позволяли решить ее. Наверное, вы думаете, что и перед этой задачей вы изучите что-то новое.

А еще вы можете удивиться: разве мы уже не знаем все, что нужно? Ведь мы можем написать функцию, которая принимает адрес электронной почты и возвращает его чистую версию, без добавленных частей, без точек перед символом @ и все в нижнем регистре. Мы также можем составлять список чистых адресов электронной почты. Для каждого адреса электронной почты выполняем чистку и проверяем, находится ли он в списке чистых адресов. Если нет, добавляем его в список, а если он уже там, ничего не делаем. Когда мы пройдемся по всем адресам электронной почты, длина списка будет равна количеству уникальных адресов электронной почты.

Ну да, действительно, у нас уже есть все, что нужно. Попробуем решить задачу.

Очистка адреса электронной почты

Рассмотрим адрес электронной почты DAnIELZIngARO+Flurry@gmAIL.COM. Нужно очистить его, то есть превратить в danielzingaro@gmail.com.

Следует убрать часть +Flurry, точки перед символом @, а также перевести все в нижний регистр. Мы можем рассматривать чистую версию как настоящий адрес электронной почты. Любой другой адрес электронной почты, который приводится к этому виду после очистки, уникальным не будет.

Очистка адреса электронной почты — это небольшая обособленная задача, поэтому напишем для нее функцию. Она будет принимать строку, представляющую собой адрес электронной почты, очищать ее и возвращать очищенный адрес электронной почты. Внутри функции выполним три этапа очистки: удалим символы между + и @, удалим точки перед @ и преобразуем оставшееся в нижний регистр. Код функции показан в листинге 8.1.

Листинг 8.1. Очистка адреса электронной почты

```
def clean(address):
    """
    address — это адрес электронной почты.

    Возвращает очищенный адрес.
    """

    # Удаление символов от  '+' до '@'
❶   plus_index = address.find('+')
    if plus_index != -1:
```

```
    ❷ at_index = address.find('@')
      address = address[:plus_index] + address[at_index:]

    # Удаление точек до символа @1
    at_index = address.find('@')
    before_at = ''
    i = 0
    while i < at_index:
     ❸ if address[i] != '.':
          before_at = before_at + address[i]
      i = i + 1

❹ cleaned = before_at + address[at_index:]

    # Преобразование в нижний регистр
❺ cleaned = cleaned.lower()

    return cleaned
```

Первым делом нужно удалить символы от `+` до `@`. Здесь будет полезен строковый метод `find`. Он возвращает индекс крайнего левого вхождения его аргумента или `-1`, если вхождение не найдется:

```
>>> 'abc+def'.find('+')
3
>>> 'abcdef'.find('+')
-1
```

С помощью метода `find` я нахожу индекс крайнего левого символа `+` ❶. Если символа `+` нет, то на этом шаге больше ничего делать не нужно. Но если он есть, мы находим индекс символа `@` ❷ и удаляем все от `+` до `@`, не включая последний.

Второй шаг — удалить все точки перед символом `@`. Для этого я использую новую строку `before_at`, в которую буду помещать часть адреса до символа `@`. Все символы, отличные от «`.`», будут добавляться в строку `before_at` ❸.

Строка `before_at` не будет содержать символ `@` или любые следующие за ним. Мы не хотим терять эту часть адреса электронной почты, поэтому я введу переменную `cleaned`, в которой будет храниться весь адрес электронной почты ❹.

Третий шаг — преобразовать весь адрес электронной почты в нижний регистр ❺. После этого адрес электронной почты будет очищен и его можно вернуть.

Проверим работу функции. Введите код функции `clean` в оболочку Python. Попробуем очистить пару адресов электронной почты:

```
>>> clean('daniel.zingaro+book@gmail.com')
'danielzingaro@gmail.com'
>>> clean('da.nielz.in.gar.o..@gmail.com')
'danielzingaro@gmail.com'
>>> clean('DAnIELZIngARO+Flurry@gmAIL.COM')
```

```
'danielzingaro@gmail.com'
>>> clean('a.b.c@d.e.f')
'abc@d.e.f'
```

Если адрес электронной почты уже чистый, функция `clean` вернет его как есть:

```
>>> clean('danielzingaro@gmail.com')
'danielzingaro@gmail.com'
```

Основная программа

Эту функцию можно использовать для очистки любого адреса электронной почты. Теперь нам нужен список чистых адресов электронной почты. Добавим в него очищенный адрес электронной почты, если сейчас там такого нет. Таким образом мы избежим дубликатов одного и того же чистого адреса электронной почты.

Основная часть программы находится в листинге 8.2. Обязательно добавьте функцию `clean` (листинг 8.1) перед этим кодом, чтобы задача могла быть решена.

Листинг 8.2. Основная программа

```
# Основная программа

for dataset in range(10):
    n = int(input())
 ❶ addresses = []
    for i in range(n):
        address = input()
        address = clean(address)
     ❷ if not address in addresses:
            addresses.append(address)

 ❸ print(len(addresses))
```

Нам нужно обработать десять тестовых примеров, поэтому обернем остальную часть программы циклом `for`, который будет выполняться десять раз.

В каждом примере требуется прочитать количество адресов электронной почты и начать заполнять список чистых адресов.

Затем во внутреннем цикле мы должны перебрать все адреса электронной почты. Считываем каждый адрес и очищаем его. Затем, если ранее он не попадался, добавляем его в список чистых адресов электронной почты ❶.

По завершении внутреннего цикла мы получим список всех чистых адресов электронной почты без дубликатов. Количество уникальных адресов ❷ — это длина списка, поэтому мы ее выводим ❸.

Неплохо, а? Получается, мы могли бы решить эту задачу сразу же, как только изучили функции в главе 6. Или даже после ознакомления со списками в главе 5.

Нет, не смогли бы. Если вы отправите код на сайт, то увидите, что есть нюансы. Первым признаком проблемы является то, что сайт долго думает, прежде чем вывести результаты. Например, мне пришлось ждать их появления целую минуту. А раньше я получал ответ сразу же.

Второй нюанс заключается в том, что, когда мы наконец получим результаты, балл за задачу окажется неполным! Я получил 3,25 балла из 5. Вы тоже получите примерно столько же, а не полные 5 баллов.

Дело тут не в том, что программа работает неправильно. С ней все в порядке. Независимо от тестового примера она выдает правильное число — количество уникальных адресов.

Но если программа правильная, в чем тогда проблема?

Проблема в том, что она слишком медленная. Именно об этом нам сообщает тестер припиской `TLE` в начале каждого теста. TLE означает timeout exceeded — «превышение допустимого времени выполнения». Чтобы задача считалась решенной, все примеры должны выполняться за 30 секунд. Если программа работает более 30 секунд, судья завершает ее и оставшиеся тестовые примеры не проверяются.

Возможно, это первая полученная вами ошибка превышения лимита времени, а может, вы видели ее и прежде, выполняя упражнения.

Первое, что нужно проверить, получив сообщение о такой ошибке, — не застревает ли программа в бесконечном цикле. Если это так, то она будет работать бесконечно и сайт прервет программу по истечении отведенного времени.

Если бесконечного цикла нет, то проблема в эффективности самой программы. Когда программисты говорят об эффективности, они имеют в виду время, которое требуется программе на выполнение. Программа, которая выполняется быстрее (занимает меньше времени), более эффективна, чем выполняющаяся медленнее (занимает больше времени). Чтобы тестовые задачи уложились в отведенные для их решения сроки, нужно сделать программу более эффективной.

Эффективность поиска по списку

Добавление элемента в список Python выполняется очень быстро. Не имеет значения, содержит ли список всего несколько или несколько тысяч значений, этот процесс занимает одинаковое количество времени.

Но вот работа оператора `in` — совсем другая история. В нашей программе оператор `in` позволяет определить, встречался ли очищенный адрес электронной почты ранее. В тестовом примере может быть до 100 000 адресов электронной почты. В худшем

случае программа должна перебрать 100 000 значений. Оказывается, оператор `in` работает очень медленно при использовании в длинных списках, и это снижает эффективность программы. Чтобы определить, находится ли значение в списке, Python перебирает последний от начала до конца до тех пор, пока не найдет искомое значение или не дойдет до конца. Чем больше значений нужно просмотреть, тем медленнее работает программа.

Легко ощутить, как замедляется программа по мере увеличения длины списка. Напишем функцию, которая принимает список и значение и ищет его в заданном списке. Попробуем найти значение 50 000, чтобы можно было ощутить изменения быстродействия.

Функция приведена в листинге 8.3. Введите его код в оболочку Python.

Листинг 8.3. Многократный поиск в коллекции

```
def search(collection, value):
    """
    Многократный поиск значения в коллекции.
    """
    for i in range(50000):
        found = value in collection
```

Давайте создадим список целых чисел от 1 до 5000 и выполним поиск числа 5000. Если будем искать самое правое значение в списке, то заставим функцию `in` работать максимально долго. Не имеет значения, что мы исследуем список целых чисел, а не список адресов электронной почты, так как эффективность будет аналогична, а числа генерировать намного проще, чем адреса электронной почты!

Результат:

```
>>> search(list(range(1, 5001)), 5000)
```

На моем ноутбуке выполнение заняло около 3 секунд. Здесь не нужно точное время, достаточно лишь понять, что происходит, когда мы увеличиваем длину списка.

Создадим список целых чисел от 1 до 10 000 и выполним поиск числа 10 000:

```
>>> search(list(range(1, 10001)), 10000)
```

На моем ноутбуке выполнение занимает около 6 секунд. Получается, для списка длиной 5000 поиск занимает 3 секунды и, удвоив длину списка, мы удвоили и время поиска.

А что, если список будет длиной 20 000? Попробуйте:

```
>>> search(list(range(1, 20001)), 20000)
```

На моем ноутбуке выполнение занимает около 12 секунд. Время снова удвоилось.

Попробуем список длиной 50 000. Тут уже придется какое-то время подождать:

```
>>> search(list(range(1, 50001)), 50000)
```

На выполнение ушло чуть больше 30 секунд. Помните, что функция поиска выполняет поиск в списке 50 000 раз. Итак, поиск в списке длиной 50 000 занимает 30 секунд.

Теперь, если бы у нас был тестовый пример, требующий аналогичных поисков, например список из 100 000 уникальных адресов электронной почты, то уже на середине его выполнения мы получили бы 50 000 значений, а нам уже известно, сколько времени занимает поиск.

И это только для одного из десяти тестов! А нам нужно пройти все десять тестовых случаев в общей сложности за 30 секунд. Если лишь один тестовый пример может занять около 30 секунд, у нас нет шансов.

Поиск в списке выполняется слишком медленно. Значит, списки Python для этой задачи не подходят, нужен другой тип данных, например множества Python. Вы удивитесь тому, как быстро выполняется поиск в множестве.

Множества

Множество — это тип Python, в котором хранится коллекция значений и при этом повторение значений недопустимо. Для создания множества используются открывающие и закрывающие фигурные скобки. В отличие от списков, значения в множестве хранятся в произвольном порядке. Пример множества целых чисел:

```
>>> {13, 15, 30, 45, 61}
{45, 13, 15, 61, 30}
```

Обратите внимание на то, что Python перепутал порядок значений. На вашем компьютере он тоже будет другим. Важно запомнить, что порядка в множествах просто нет, а если он нужен, то множества вам не подходят.

Если мы попытаемся включить в множество несколько одинаковых значений, то сохранится только одно из них:

```
>>> {1, 1, 3, 2, 3, 1, 3, 3, 3}
{1, 2, 3}
```

Множества считаются равными, если в них содержатся одинаковые значения, пусть и в разном порядке:

```
>>> {1, 2, 3} == {1, 2, 3}
True
>>> {1, 1, 3, 2, 3, 1, 3, 3, 3} == {1, 2, 3}
True
>>> {1, 2} == {1, 2, 3}
False
```

Мы можем создать множество из строк, например:

```
>>> {'abc@d.e.f', 'danielzingaro@gmail.com'}
{'abc@d.e.f', 'danielzingaro@gmail.com'}
```

Но не можем создать множество из списков:

```
>>> {[1, 2], [3, 4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Значения в множестве должны быть неизменяемыми, поэтому мы не можем помещать списки в множества. Это ограничение связано с тем, как Python ищет в них значения. Когда Python добавляет в множество значение, он использует само значение, чтобы определить, где именно оно хранится. Позже Python может найти его, посмотрев в том месте, где оно должно быть расположено. Если бы значение множества было изменяемым, Python не мог бы найти его. Создать множество из списков нельзя, а список множеств — можно:

```
>>> lst = [{1, 2, 3}, {4, 5, 6}]
>>> lst
[{1, 2, 3}, {4, 5, 6}]
>>> len(lst)
2
>>> lst[0]
{1, 2, 3}
```

Вы можете использовать функцию `len` для определения количества значений в множестве:

```
>>> len({2, 4, 6, 8})
4
```

Можно перебирать значения в множестве:

```
>>> for value in {2, 4, 6, 8}:
...     print('I found', value)
...
I found 8
I found 2
I found 4
I found 6
```

Срезов и индексации в множестве нет, так как нет порядка и индексов.

Чтобы создать пустое множество, не получится использовать пустую пару фигурных скобок {}. Тут есть некоторая несогласованность синтаксиса:

```
>>> type({2, 4, 6, 8})
<class 'set'>
>>> {}
{}
>>> type({})
<class 'dict'>
```

Использование {} дает нам неправильный тип: `dict` (словарь), а не множество.

Мы поговорим о словарях позже в этой главе. Чтобы создать пустое множество, применяется функция `set()`, например:

```
>>> set()
set()
>>> type(set())
<class 'set'>
```

Методы множеств

Множества изменяемы, поэтому мы можем добавлять в них значения и удалять их оттуда. Это делается с помощью методов.

Узнать список методов множеств можно с помощью функции `dir(set())`. Чтобы получить справку по определенному методу, используйте функцию `help`, как мы делали со строками и списками, — `help(set().add)`.

Метод `add` позволяет добавлять значения в множество аналогично методу `append` списков:

```
>>> s = set()
>>> s
set()
>>> s.add(2)
>>> s
{2}
>>> s.add(4)
>>> s
{2, 4}
>>> s.add(6)
>>> s
{2, 4, 6}
>>> s.add(8)
>>> s
{8, 2, 4, 6}
```

```
>>> s.add(8)
>>> s
{8, 2, 4, 6}
```

Чтобы удалить значение, задействуйте метод `remove`:

```
>>> s.remove(4)
>>> s
{8, 2, 6}
>>> s.remove(8)
>>> s
{2, 6}
>>> s = {2, 6}
>>> s.remove(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 8
```

ПРОВЕРИМ ЗНАНИЯ

С помощью функции `help` почитайте о методах `update` и `intersection`. Что выведет функция `Print` в приведенном коде?

```
s1 = {1, 3, 5, 7, 9}
s2 = {1, 2, 4, 6, 8, 10}
s3 = {1, 4, 9, 16, 25}
s1.update(s2)
s1.intersection(s3)
print(s1)
```

А. `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`

Б. `{1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`

В. `{1, 4, 9}`

Г. `{1, 4, 9, 16, 25}`

Д. `{1}`

Ответ: **А**. Метод `update` добавляет все, что есть в множестве `s2`, но отсутствует в множестве `s1`. После вызова функции `update` множество `s1` получается таким: `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`.

Теперь метод `intersection` — пересечение множеств. Так называется новое множество, состоящее из значений, которые есть в обоих множествах. В нашем случае пересечение `s1` и `s3` — это `{1, 4, 9}`. Пересечение *не* изменяет переданное множество, а создает новое! Значение `s1` не меняется.

Эффективность поиска по множеству

Вернемся к решению задачи.

Важен ли порядок очищенных адресов электронной почты? Нет! Нужно лишь знать, встречался нам данный адрес электронной почты или нет.

Разрешены ли дубликаты в очищенных адресах электронной почты? Снова нет! Наоборот, нужно избавиться от повторяющихся адресов почты.

Порядок не имеет значения, дубликаты не допускаются. Именно эти два пункта говорят о том, что для задачи нужны множества.

Попытавшись использовать список, мы потерпели неудачу, потому что поиск по нему идет слишком медленно. Множества будут работать лучше, так как поиск по множеству выполняется быстрее.

Функция поиска по списку была приведена в листинге 8.3, но она не делает ничего такого, для чего обязательно требовался бы список! Задействованный в ней оператор `in` работает как со списками, так и с множествами. Получается, мы можем использовать эту функцию без изменений.

Введите функцию `search` из листинга 8.3 в оболочку Python. Выполните приведенные далее действия на своем компьютере, чтобы ощутить разницу в быстродействии множеств и списков:

```
  >>> search(list(range(1, 50001)), 50000)
❶ >>> search(set(range(1, 50001)), 50000)
```

В строке ❶ функция `set` создает именно множество целых чисел, а не список.

На моем ноутбуке поиск по списку занимает около 30 секунд. Для сравнения: поиск в множестве происходит очень быстро, почти мгновенно.

Невероятная скорость. Со списками этого делать не стоит, но попробуйте выполнить поиск в множестве из 500 000 значений:

```
>>> search(set(range(1, 500001)), 500000)
```

Ну вот, ничего сложного.

Python управляет списком так, чтобы мы могли использовать любой его индекс в любой момент. С порядком значений он обходится строго: первое должно быть на индексе `0`, второе — на индексе `1` и т. д. А значения множеств хранятся как угодно, поскольку в них порядок неважен. Именно это упрощение позволяет Python оптимизировать поиск в множестве с точки зрения скорости.

Существуют и другие операции, которые по аналогичным причинам выполняются очень медленно в больших списках, но очень быстро в больших множествах. Например, удаление значения из списка занимает много времени, потому что Python после этого уменьшает индекс каждого значения справа от удаленного. А удаление значения из множества происходит очень быстро, так как индексов у них нет!

Решение задачи

У нас уже есть функция очистки адреса электронной почты (см. листинг 8.1), и для решения с множествами она годится. Что касается основной программы, то часть листинга 8.2 перекочует туда, но нужно использовать множества вместо списков.

Новая основная программа приведена в листинге 8.4. Но не забудьте добавить в начало код из листинга в 8.1.

Листинг 8.4. Основная программа, переделанная под множества

```
# Основная программа

for dataset in range(10):
    n = int(input())
❶   addresses = set()
    for i in range(n):
        address = input()
        address = clean(address)
❷       addresses.add(address)

    print(len(addresses))
```

Обратите внимание на то, что теперь используется множество ❶, а не список адресов электронной почты. После очистки каждого адреса электронной почты добавляем его в множество методом `add` ❷.

В листинге 8.2 мы применили оператор `in` для проверки того, встречается ли адрес электронной почты в списке, чтобы не добавлять дубликаты. В решении с множествами такой проверки нет, и мы добавляем каждый адрес в множество, даже не проверяя, есть он там или еще нет.

Дело в том, что можно обойтись без проверки, потому что множества сами по себе не допускают дубликатов. Метод `add` сам проверяет дубликаты и не пропускает их. О времени беспокоиться не нужно, все происходит быстро.

Если вы отправите это решение на сайт, тесты должны быть выполнены вовремя.

Как вы уже видели, выбор подходящего типа Python может повлиять на успех решения. Прежде чем приступить к написанию кода, спросите себя, какие операции вы будете выполнять в коде и какой тип Python идеально подходит для них.

Прежде чем продолжить, можете попробовать выполнить упражнения 1 и 2, приведенные в конце главы.

Задача 19. Общие слова

В этой задаче нам нужно будет связать некоторые слова с количеством их вхождений. Это выходит за рамки того, что позволяют делать множества, поэтому мы не будем их использовать. Вместо этого задействуем словари Python, попутно выяснив, что они собой представляют.

Задача с сайта DMOJ, код `cco99p2`.

Постановка задачи

Даны m слов. Они не обязательно должны быть разными — могут повторяться. Также дано целое число k. Наша задача — найти k-е по частоте слово. Слово w является k-м по частоте, если ровно $k - 1$ различных слов встречается чаще, чем w. В зависимости от исходных данных k-е по частоте слово может быть одно, их может быть несколько или ни одного.

Убедимся, что мы хорошо понимаем, что такое k-е по частоте слово. Если $k = 1$, то нам нужны слова, чаще которых встречается 0 других слов, то есть нас спрашивают, какие слова встречаются чаще всего. Если $k = 2$, то надо узнать, у какого слова есть лишь один более часто встречающийся конкурент, и т. д.

Входные данные

Во входных данных мы получаем строку с количеством тестовых примеров, за которой идут сами тестовые примеры. В каждый из них входят:

- строка, содержащая целые числа m (количество слов в тестовом примере) и k, разделенные пробелом. Число m лежит в диапазоне от 0 до 1000, k не менее 1;
- m строк, каждая из которых содержит слово. Каждое слово состоит не более чем из 20 символов и написано в нижнем регистре.

Выходные данные

Для каждого тестового примера выведите:

- строку формата

  `p most common word(s):`

  где p = `1st`, если k = `1`, `2nd` при k = `2` и т. д.;
- все слова, находящиеся на k-м месте по частоте. Если таких слов нет, ничего не выводите;
- пустую строку.

Ограничение по времени — 1 секунда.

Тестовый пример

Начнем с тестового примера, чтобы лучше понять задачу и выбрать нужный тип Python (словарь, конечно же).

Предположим, что нас интересуют самые распространенные слова, то есть k равно 1. Тестовый пример:

```
1
14 1
storm
cut
magma
cut
brook
gully
gully
storm
cliff
cut
blast
brook
cut
gully
```

Слово `cut` встречается чаще всего — четыре раза, ни одно другое слово не встречается так часто. Правильный вывод будет выглядеть так:

```
  1st most common word(s):
❶ cut
```

Обратите внимание на обязательную пустую строку в конце ❶.

А что делать, если $k = 2$? Для ответа на этот вопрос можно снова просмотреть слова и посчитать их вхождения, но есть и другой способ организовать слова, который значительно упростит нашу задачу. Давайте вместо списка слов возьмем все слова и подсчитаем их (табл. 8.1).

Таблица 8.1. Слова и количество их вхождений

Слово	Количество вхождений
cut	4
gully	3
storm	2
brook	2
magma	1
cliff	1
blast	1

Я отсортировал слова по частоте. В верхней строке видно, что `cut` является правильным ответом для $k = 1$. Глядя на вторую строку, мы видим, что для $k = 2$ нужно слово `gully`, так как выше него только `cut`.

Теперь рассмотрим $k = 3$. На этот раз нужно вывести два слова, `storm` и `brook`, потому что у них одинаковое количество вхождений. Получается, иногда приходится выводить более одного слова.

Также возможно, что выводить не нужно будет ничего! Например, возьмем $k = 4$. У нас нет слов, для которых имеются ровно три слова с бóльшим числом вхождений. Вы можете задать вопрос: почему для $k = 4$ мы не выводим слово `magma`? Дело в том, что это слово стоит на пятом месте, а не на четвертом, так как третье и четвертое заняты словами `storm` и `brook`.

Для $k = 5$ нужно вывести три слова: `magma`, `cliff` и `blast`. Для значений $k = 6$ и более слов уже нет.

Таблица 8.1 немного упрощает работу, осталось лишь понять, как организовать такую же информацию в Python.

Словари

Словарь — это тип Python, в котором хранится отображение одной группы элементов, называемых *ключами*, на другую группу элементов, называемых *значениями*.

Чтобы ограничить словарь, используются открывающие и закрывающие фигурные скобки. Их же мы применяли для множеств, но в Python множества от словаря отличаются содержимым фигурных скобок. У множеств мы перечисляем значения, а у словарей — пары «ключ:значение».

Далее приведен словарь, где некоторые строки отображаются в числа:

```
>>> {'cut':4, 'gully':3}
{'cut': 4, 'gully': 3}
```

В этом словаре используются ключи `'cut'` и `'gully'` и значения 4 и 3. Ключ `'cut'` отображается на значение 4, а ключ `'gully'` — на значение 3.

Познакомившись с множествами, вы можете задаться вопросом, находятся ли значения в словаре в том же порядке, в котором мы их вводим. Например, может ли быть так:

```
>>> {'cut':4, 'gully':3}
{'gully': 3, 'cut': 4}
```

В Python 3.7 ответ отрицательный: словари сохраняют порядок, в котором добавляются значения. В более ранних версиях Python словари его не сохраняли, чтобы можно было добавлять пары в одном порядке, а возвращать в другом. Желательно писать код, не ориентированный на Python 3.7, поскольку более старые версии, вероятно, какое-то время еще будут использоваться.

Словари равны, если они содержат одинаковые пары «ключ:значение», даже если те находятся там в разном порядке:

```
>>> {'cut':4, 'gully':3} == {'cut':4, 'gully':3}
True
>>> {'cut':4, 'gully':3} == {'gully': 3, 'cut': 4}
True
>>> {'cut':4, 'gully':3} == {'gully': 3, 'cut': 10}
False
>>> {'cut':4, 'gully':3} == {'cut': 4}
False
```

Ключи словаря должны быть уникальными. Если вы попытаетесь включить один и тот же ключ несколько раз, сохранится только одна пара с ним:

```
>>> {'storm': 1, 'storm': 2}
{'storm': 2}
```

А вот повторяющиеся значения допустимы:

```
>>> {'storm': 2, 'brook': 2}
{'storm': 2, 'brook': 2}
```

Ключи должны быть неизменяемого типа, например числа и строки. Значения могут быть и неизменными, и изменяемыми. Это означает, что мы не можем использовать списки в качестве ключа, а вот в качестве значения — вполне:

```
>>> {['storm', 'brook']: 2}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> {2: ['storm', 'brook']}
{2: ['storm', 'brook']}
```

Функция `len` возвращает количество пар «ключ:значение» в словаре:

```
>>> len({'cut':4, 'gully':3})
2
>>> len({2: ['storm', 'brook']})
1
```

Чтобы создать пустой словарь, используются фигурные скобки `{}`. Именно поэтому для создания множества мы применяли функцию `set()` — красивый синтаксис забронирован словарями:

```
>>> {}
{}
>>> type({})
<class 'dict'>
```

Тип называется `dict`, а не `dictionary`. Вообще оба этих слова в Python взаимозаменяемы, но далее в книге я буду писать `dictionary`.

ПРОВЕРИМ ЗНАНИЯ

Что из перечисленного лучше реализовать словарем, а не списком или множеством?

А. Порядок, в котором спортсмены пришли к финишу.

Б. Ингредиенты для рецепта.

В. Названия стран и их столиц.

Г. Пятьдесят случайных чисел.

Ответ: **В**. Только в этом случае потребуется соответствие между ключом и значением. В данном случае ключами могут стать страны, а значениями — их столицы.

ПРОВЕРИМ ЗНАНИЯ

Какого типа значения (ключи можно не рассматривать) находятся в приведенном далее словаре:

```
{'MLB': {'Bluejays': [1992, 1993],
         'Orioles': [1966, 1970, 1983]},
 'NFL': {'Patriots': ['too many']}}
```

А. Целые числа.

Б. Строки.

В. Списки.

Г. Словари.

Д. Несколько типов из указанных.

Ответ: **Г**. Значение для каждого ключа этого словаря само по себе является словарем. Например, ключу `'MLB'` соответствует словарь, в котором есть две свои пары «ключ:значение».

Индексирование словарей

С помощью квадратных скобок можно найти значение, с которым сопоставляется ключ. Механизм аналогичен тому, как индексируется список, но вместо индексов указываются ключи:

```
>>> d = {'cut':4, 'gully':3}
>>> d
{'cut': 4, 'gully': 3}
>>> d['cut']
4
>>> d['gully']
3
```

При использовании несуществующего ключа возникает ошибка:

```
>>> d['storm']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'storm'
```

Чтобы застраховаться от этой ошибки, сначала надо применить оператор `in` для проверки того, есть ли ключ в словаре. При использовании для словаря оператор `in` проверяет только ключи, а не значения. Далее показано, как проверить, существует ли ключ, прежде чем найти его значение:

```
>>> if 'cut' in d:
...     print(d['cut'])
...
4
>>> if 'storm' in d:
...     print(d['storm'])
...
```

Индексирование и поиск в словаре — чрезвычайно быстрые операции. Для них не выполняется перебор никаких списков независимо от того, сколько ключей в словаре.

Иногда для поиска значения ключа удобнее использовать метод `get`, чем индексацию. Метод `get` не выдает ошибок, даже если ключа не существует:

```
>>> print(d.get('cut'))
4
>>> print(d.get('storm'))
None
```

Если ключ существует, `get` возвращает его значение. В противном случае возвращается `None`, что говорит об отсутствии ключа.

Мы можем использовать квадратные скобки не только для поиска значения ключа, но и для добавления ключей в словарь или изменения значения, которому соответствует ключ. Далее приведен код, который показывает, как это сделать:

```
>>> d = {}
>>> d['gully'] = 1
>>> d
{'gully': 1}
>>> d['cut'] = 1
>>> d
{'gully': 1, 'cut': 1}
>>> d['cut'] = 4
>>> d
{'gully': 1, 'cut': 4}
>>> d['gully'] = d['gully'] + 1
>>> d
{'gully': 2, 'cut': 4}
>>> d['gully'] = d['gully'] + 1
>>> d
{'gully': 3, 'cut': 4}
```

ПРОВЕРИМ ЗНАНИЯ

С помощью функции `help({}.get)` посмотрите, как работает метод `get`. Какой результат даст приведенный далее код?

```
d = {3: 4}
d[5] = d.get(4, 8)
d[4] = d.get(3, 9)
print(d)
```

А. `{3: 4, 5: 8, 4: 9}`

Б. `{3: 4, 5: 8, 4: 4}`

В. `{3: 4, 5: 4, 4: 3}`

Г. Ошибка, вызванная методом `get`.

Ответ: **Б**. Первый вызов возвращает 8: ключ 4 в словаре есть, поэтому добавляется ключ 5 со значением 8. Второй вызов метода `get` возвращает 4: ключ 3 в словаре уже есть, поэтому второй параметр игнорируется и добавляется ключ 4 со значением 4.

Перебор словарей в цикле

Перебрав словарь циклом `for`, мы получим ключи словаря:

```
>>> d = {'cut': 4, 'gully': 3, 'storm': 2, 'brook': 2}
>>> for word in d:
...     print('a key is', word)
...
a key is cut
a key is gully
a key is storm
a key is brook
```

Нам может потребоваться также доступ к значению, связанному с каждым ключом, и получить его мы можем с помощью ключей. Приведенный далее цикл обращается как к ключу, так и к его значению:

```
>>> for word in d:
...     print('key', word, 'has value', d[word])
...
key cut has value 4
key gully has value 3
key storm has value 2
key brook has value 2
```

У словарей есть методы, позволяющие получить доступ к ключам, значениям или и к тому и к другому.

Метод `keys` дает нам ключи, а метод `values` — значения:

```
>>> d.keys()
dict_keys(['cut', 'gully', 'storm', 'brook'])
>>> d.values()
dict_values([4, 3, 2, 2])
```

Получаются не списки, но с помощью функции `list` можно преобразовать их:

```
>>> keys = list(d.keys())
>>> keys
['cut', 'gully', 'storm', 'brook']
>>> values = list(d.values())
>>> values
[4, 3, 2, 2]
```

Когда ключи сохранены в виде списка, мы можем отсортировать их, а затем пройтись по словарю в порядке, получившемся после сортировки:

```
>>> keys.sort()
>>> keys
['brook', 'cut', 'gully', 'storm']
>>> for word in keys:
...     print('key', word, 'has value', d[word])
...
key brook has value 2
key cut has value 4
key gully has value 3
key storm has value 2
```

Мы также можем перебирать значения:

```
>>> for num in d.values():
...     print('number', num)
...
number 4
number 3
number 2
number 2
```

Перебор ключей часто предпочтительнее перебора значений. От ключа к его значению перейти легко. Однако, как мы увидим в следующем разделе, вернуться от значения к его ключу не так просто.

Еще один важный для работы метод — это `items`. Он дает доступ сразу к ключам и значениям:

```
>>> pairs = list(d.items())
>>> pairs
[('cut', 4), ('gully', 3), ('storm', 2), ('brook', 2)]
```

Получается, у нас есть один способ перебрать пары «ключ:значение» в словаре:

```
>>> for pair in pairs:
...     print('key', pair[0], 'has value', pair[1])
...
key cut has value 4
key gully has value 3
key storm has value 2
key brook has value 2
```

Внимательно посмотрим на переменную `pairs`:

```
>>> pairs
[('cut', 4), ('gully', 3), ('storm', 2), ('brook', 2)]
```

Здесь есть что-то подозрительное: каждое внутреннее значение заключено в круглые, а не квадратные скобки. Оказывается, это не список списков, а список *кортежей*:

```
>>> type(pairs[0])
<class 'tuple'>
```

Кортежи похожи на списки в том смысле, что в них хранится последовательность значений. Наиболее важное различие между кортежами и списками состоит в том, что кортежи неизменяемы. Можно перебирать их, обращаться к ним по индексу и брать с них срезы, но нельзя их изменять. Попытавшись изменить кортеж, вы получите сообщение об ошибке:

```
>>> pairs[0][0] = 'river'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Можете создавать собственные кортежи, используя круглые скобки. Для кортежа с одним значением нужна конечная запятая. Для кортежа с несколькими значениями — не нужна:

```
>>> (4,)
(4,)
>>> (4, 5)
(4, 5)
>>> (4, 5, 6)
(4, 5, 6)
```

У кортежей есть методы, но их мало, потому что методы, которые изменяют кортеж, недопустимы. Я рекомендую подробнее почитать о кортежах, если вам интересно, но мы в этой книге использовать их не будем.

Инвертирование словаря

Мы почти поняли, как решить задачу «Общие слова» с помощью словарей. План такой: у нас будет словарь, который отображает слова на количество их вхождений. Каждый раз, обрабатывая новое слово, мы проверяем, есть ли оно в словаре. Если это не так, добавляем в словарь его ключ со значением 1. Если же это так, увеличиваем его значение на 1.

Далее приведен пример добавления двух слов, одно из которых мы видели, а другое — нет:

```
>>> d = {'storm': 1, 'cut': 1, 'magma': 1}
>>> word = 'cut' # ключ 'cut' уже есть в словаре
>>> if not word in d:
...     d[word] = 1
... else:
...     d[word] = d[word] + 1
...
>>> d
{'storm': 1, 'cut': 2, 'magma': 1}
>>> word = 'brook' # ключа 'brook' в словаре нет
>>> if not word in d:
...     d[word] = 1
... else:
...     d[word] = d[word] + 1
...
>>> d
{'storm': 1, 'cut': 2, 'magma': 1, 'brook': 1}
```

Словари упрощают переход от ключа к значению. Например, имея ключ `'brook'`, мы легко найдем значение 1:

```
>>> d['brook']
1
```

Это похоже на переход от слова в левом столбце табл. 8.1 к количеству его вхождений — в правом. Но мы не знаем, какие именно слова имеют указанное количество вхождений. На самом-то деле нам нужно уметь переходить от правого столбца к левому — от количества вхождений к словам. Тогда мы сможем отсортировать количество вхождений от наибольшего к наименьшему и найти нужные слова.

То есть нужно перейти из такого словаря:

```
{'storm': 2, 'cut': 4, 'magma': 1, 'brook': 2,
 'gully': 3, 'cliff': 1, 'blast': 1}
```

к *перевернутому виду*:

```
{2: ['storm', 'brook'], 4: ['cut'], 1: ['magma', 'cliff', 'blast'],
 3: ['gully']}
```

Исходный словарь отображает строки в числа. Перевернутый — числа в строки. А точнее, даже в *списки* строк. Помните, что каждый ключ можно использовать в словаре только один раз. В перевернутом словаре нам нужно отобразить каждый ключ на несколько значений, поэтому эти значения нужно сохранить в список.

Чтобы инвертировать словарь, каждый ключ нужно сделать значением, а каждое значение — ключом. Если ключа в инвертированном словаре еще нет, мы создаем список для его значения. Если ключ уже есть в инвертированном словаре, то добавляем значение в его список.

Пора написать функцию, возвращающую инвертированную версию словаря (листинг 8.5).

Листинг 8.5. Инвертирование словаря

```
def invert_dictionary(d):
    """
    d — это словарь, отображающий строки в числа.

    Возвращает инвертированный словарь d.
    """

    inverted = {}
❶   for key in d:
❷       num = d[key]
        if not num in inverted:
❸           inverted[num] = [key]
        else:
❹           inverted[num].append(key)
    return inverted
```

С помощью цикла `for` мы проходимся по словарю `d` ❶ и получаем все ключи. Индексируем `d`, чтобы получить значение, отображаемое данным ключом ❷. Затем добавляем эту пару «ключ:значение» в инвертированный словарь. Если `num` еще не является ключом в инвертированном словаре, добавляем его и сопоставляем с соответствующим ключом в `d` ❸. Если `num` уже является ключом в инвертированном словаре, то в его значении есть список, поэтому мы можем использовать метод `append`, чтобы добавить ключ `d` в список значений ❹.

Введите код функции `invert_dictionary` в оболочку Python. Давайте попробуем:

```
>>> d = {'a': 1, 'b': 1, 'c': 1}
>>> invert_dictionary(d)
{1: ['a', 'b', 'c']}
>>> d = {'storm': 2, 'cut': 4, 'magma': 1, 'brook': 2,
...      'gully': 3, 'cliff': 1, 'blast': 1}
>>> invert_dictionary(d)
{2: ['storm', 'brook'], 4: ['cut'], 1: ['magma', 'cliff', 'blast'],
3: ['gully']}
```

Теперь мы готовы решить задачу «Общие слова» с помощью перевернутого словаря.

Решение задачи

Если вы хотите попрактиковаться в нисходящем проектировании, можете решить задачу самостоятельно. Для экономии места я не буду воспроизводить весь процесс проектирования, а покажу решение полностью, затем мы обсудим каждую функцию и то, как она используется.

Код

Решение приведено в листинге 8.6.

Листинг 8.6. Решение задачи

```
def invert_dictionary(d):
    """
    d — это словарь, отображающий строки в числа.

    Возвращает инвертированный словарь d.
    """
    inverted = {}
    for key in d:
        num = d[key]
        if not num in inverted:
            inverted[num] = [key]
        else:
            inverted[num].append(key)
    return inverted

❶ def with_suffix(num):
    """
    num — это число >= 1.

    Возвращает строку с числом и соответствующим суффиксом.
    """
  ❷ s = str(num)
```

```
    ❽ if s[-1] == '1' and s[-2:] != '11':
          return s + 'st'
      elif s[-1] == '2' and s[-2:] != '12':
          return s + 'nd'
      elif s[-1] == '3' and s[-2:] != '13':
          return s + 'rd'
      else:
            return s + 'th'

❾ def most_common_words(num_to_words, k):
      """
      num_to_words — это словарь, отображающий число
      вхождений на список слов.
      k — это число >= 1.

      Возвращает список из k-х по частоте слов
      из num_to_words.
      """
      nums = list(num_to_words.keys())
      nums.sort(reverse=True)

      total = 0
      i = 0
      done = False
    ❺ while i < len(nums) and not done:
         num = nums[i]
      ❻ if total + len(num_to_words[num]) >= k:
            done = True
         else:
            total = total + len(num_to_words[num])
            i = i + 1

    ❼ if total == k - 1 and i < len(nums):
          return num_to_words[nums[i]]
      else:
          return []

❽ n = int(input())

  for dataset in range(n):
      lst = input().split()
      m = int(lst[0])
      k = int(lst[1])

      word_to_num = {}

      for i in range(m):
          word = input()
          if not word in word_to_num:
              word_to_num[word] = 1
          else:
              word_to_num[word] = word_to_num[word] + 1
```

```
❾ num_to_words = invert_dictionary(word_to_num)
  ordinal = with_suffix(k)
  words = most_common_words(num_to_words, k)

  print(f'{ordinal} most common word(s):')
  for word in words:
      print(word)

  print()
```

Первая функция — `invert_dictionary`. Мы уже обсуждали ее ранее в этой главе. Рассмотрим остальные части программы.

Добавление суффикса

Функция `with_suffix` ❶ принимает число и возвращает строку с добавленным к ней суффиксом. Эта функция нам нужна из-за дурацкого требования выводить k с суффиксом. Например, если $k = 1$, то вывод будет таким:

```
1st most common word(s):
```

Если $k = 2$, нужно будет вывести:

```
2nd most common word(s):
```

и т. д. Функция `with_suffix` добавляет к числу нужный суффикс. Сначала мы преобразуем число в строку ❷, чтобы можно было обработать его цифры. Затем с помощью серии проверок определяем нужный суффикс — `st`, `nd`, `rd` или `th`. Например, если последняя цифра — 1, но две последние цифры не 11 ❸, то правильный суффикс — `st`. Получается `1st`, `21st` и `31st`, но не `11st` (это неправильно).

Поиск k-го по частоте слова

Именно функция `most_common_words` ❹ в конечном итоге находит нужные нам слова. Она принимает перевернутый словарь (он превращает количество вхождений в список слов) и целое число k и возвращает список k-х по частоте слов.

Чтобы понять, как это работает, рассмотрим пример перевернутого словаря. Я расположил ключи от наибольшего к наименьшему числу вхождений, так как именно в таком порядке функция `most_common_words` перебирает ключи. Этот словарь приведен далее:

```
{4: ['cut'],
 3: ['gully'],
 2: ['storm', 'brook'],
 1: ['magma', 'cliff', 'blast']}
```

Предположим, что $k = 3$. Следовательно, ровно два слова должны встречаться чаще, чем то, которое функция возвращает. В первом ключе словаря нужных слов нет — там лишь одно слово `cut`, которое на третьем месте находиться не может. К второму ключу тоже относится одно слово — `gully`. Мы обработали уже два слова, но еще не нашли третье по частоте. Нужные нам слова относятся к третьему ключу словаря. Там мы найдем два слова, чаще которых только два других слова, то есть они подходят для $k = 3$.

А что в случае $k = 4$? На этот раз ровно три слова должны быть более распространенными, чем те, которые мы возвращаем. Но на третьем ключе два слова, и для них $k = 3$, а для следующего ключа уже $k = 5$, то есть для $k = 4$ ответа *нет*.

Таким образом, нужно складывать количество изученных слов, пока не будет найден ключ, который может содержать нужные слова. Если ровно k слов встречается чаще рассматриваемого, то ответ найден, в противном случае выводить ничего не нужно.

Теперь пройдемся по самому коду. Начнем с получения списка ключей словаря и их сортировки от большего к меньшему. Затем будем перебирать ключи в обратном отсортированном порядке ❺. Переменная `done` сообщает, просмотрели ли мы к данному моменту k или более слов. Как только это произойдет ❻, выходим из цикла.

По завершении цикла проверяем, есть ли слова, удовлетворяющие значению k. Если есть ровно $k - 1$ слов, которые встречаются чаще, и не все ключи еще перебраны ❼, то слова, которые нужно вернуть, найдены. В противном случае таких слов нет, поэтому возвращаем пустой список.

Основная программа

Сейчас мы попадаем в основную часть программы ❽. Составляем словарь `word_to_num`, который сопоставляет каждое слово с количеством его вхождений. Затем создаем перевернутый словарь `num_to_words` ❾, который отображает каждое количество вхождений в соответствующий список слов. Обратите внимание на то, как названия этих словарей указывают на направление отображения: `word_to_num` переходит от слов к числам, а `num_to_words` — от чисел к словам.

Остальная часть кода вызывает другие вспомогательные функции и выводит подходящие слова.

Теперь можно отправить код на сайт. Это здорово, ведь это первая задача, которую вы решили с помощью словарей. Всякий раз, когда нужно сопоставить два типа значений, подумайте, можно ли организовать информацию с помощью словаря. Если да, то, скорее всего, вы быстро найдете путь к эффективному решению!

Задача 20. Города и штаты

Рассмотрим еще одну задачу, в которой можно задействовать словарь. Читая ее условие, подумайте о том, что будет удобно использовать в качестве ключей, а что — в качестве значений.

Задача «Города и штаты» взята с конкурса USACO 2016 December Silver Contest.

Постановка задачи

Соединенные Штаты разделены на географические регионы, называемые *штатами*, в каждом из которых есть один или несколько городов. Каждому штату присвоено двухсимвольное сокращение. Например, для штата Пенсильвания используется аббревиатура PA, а для Южной Каролины — SC. Запишем названия городов и аббревиатуры их штатов в верхнем регистре.

Рассмотрим пару городов, SCRANTON PA и PARKER SC. Эта пара городов *особенная*, потому что первые два символа каждого из них соответствуют аббревиатуре штата, в котором расположен другой. То есть первые два символа названия SCRANTON — это SC (так обозначается штат, где находится PARKER), а первые два символа названия PARKER — это PA (штат, где находится SCRANTON).

Пара городов считается *особенной*, если они соответствуют этому свойству и находятся в разных штатах.

Требуется определить количество особых пар городов в предоставленных входных данных.

Входные данные

Входные данные нужно прочитать из файла `citystate.in`. Они состоят:

- из строки с числом n — количеством городов в диапазоне от 1 до 200 000;
- n строк, по одной на город. В каждой строке имеются название города в верхнем регистре, пробел и аббревиатура штата, написанная прописными буквами. Названия городов имеют длину от 2 до 10 символов, аббревиатуры всех штатов — ровно два символа. Одно и то же название города может встречаться в нескольких штатах, но не более одного раза в одном и том же штате. Названием города или штата в этой задаче будет считаться любая строка, отвечающая этим требованиям, даже если это не реальный город или штат США.

Выходные данные

Выходные данные нужно записать в файл `citystate.out`.

Выведите количество особых пар городов.

Ограничение по времени на решение каждого тестового примера — 4 секунды.

Тестовый пример

Возможно, вы думаете, что эту задачу можно решить с помощью списков. Это хорошая мысль! Если вам интересно, можете попробовать сперва такой способ. В этом случае вам понадобились бы два вложенных цикла для рассмотрения каждой пары городов и проверки того, является ли она особенной. Используя этот подход, можно найти верное решение.

Вот только верное решение может оказаться медленным. Список городов может быть огромным — до 200 000 пар, и любое решение, где бы применялся поиск совпадающих городов в списке, обречено быть слишком медленным. Давайте рассмотрим тестовый пример и подумаем, чем могут помочь словари.

Вот тестовый пример:

```
12
SCRANTON PA
MANISTEE MI
NASHUA NH
PARKER SC
LAFAYETTE CO
WASHOUGAL WA
MIDDLEBOROUGH MA
MADISON MI
MILFORD MA
MIDDLETON MA
COVINGTON LA
LAKEWOOD CO
```

Первый город — `SCRANTON PA`. Чтобы найти особые пары, включающие его, нужно найти другие города, название которых начинается с `PA`, а штат — это `SC`. Единственный другой город, соответствующий этому описанию, — `PARKER SC`.

Обратите внимание: нам важно лишь то, что название `SCRANTON` начинается с `SC`, а штат — это `PA`. Подошла бы любая пара вроде `SCMERWIN PA` или `SCSHOCK PA`.

Но вернемся к начальным буквам — назовем их *комбинацией*. Для `SCRANTON PA` комбинацией будет `SCPA`, а для `PARKER SC` — `PASC`.

Вместо того чтобы искать особые пары городов, теперь можем искать особые пары комбинаций. Давай попробуем.

В примере есть два города с комбинацией MAMI. Это MANISTEE MI и MADISON MI, но нас интересует лишь их количество. Города с комбинациями MAMI начинаются с MA и находятся в штате MI. Чтобы подсчитать специальные пары, включающие города MAMI, нам нужно знать города, которые начинаются с MI и находятся в штате MA. То есть нужно знать количество городов MIMA. Таких три — MIDDLEBOROUGH MA, MILFORD MA и MIDDLETON MA, но нам нужно лишь их количество. Итак, у нас есть два города MAMI и три города MIMA. Таким образом, общее количество особых пар для этих комбинаций составляет $2 \cdot 3 = 6$, потому что для каждого из двух городов MAMI есть выбор из трех городов MIMA.

Все шесть особых пар приведены далее:

- MANISTEE MI и MIDDLEBOROUGH MA;
- MANISTEE MI и MILFORD MA;
- MANISTEE MI и MIDDLETON MA;
- MADISON MI и MIDDLEBOROUGH MA;
- MADISON MI и MILFORD MA;
- MADISON MI и MIDDLETON MA.

Если нужно сопоставить комбинации SCPA, PASC, MAMI, MIMA и т. д. с числом их вхождений, можно просмотреть комбинации и найти количество особых пар городов. Словарь — идеальный инструмент для хранения подобного сопоставления.

Далее приведен словарь, который мы хотели бы создать для тестового примера:

```
{'SCPA': 1, 'MAMI': 2, 'NANH': 1, 'PASC': 1, 'LACO': 2,
 'MIMA': 3, 'COLA': 1}
```

С помощью этого словаря можем узнать количество особых пар городов. Давайте поработаем над процессом.

Первый ключ — `'SCPA'`, его значение — `1`. Чтобы найти особые пары городов с комбинацией `'SCPA'`, требуется найти количество `'PASC'`. Оно тоже равно `1`. Мы перемножаем значения и получаем $1 \cdot 1 = 1$, то есть из этих комбинаций образована одна особая пара городов. Следует проделать ту же процедуру для всех прочих ключей в словаре.

Следующий ключ — `'MAMI'`, его значение — `2`. Чтобы найти особые пары городов с комбинацией `'MAMI'`, нам нужно найти значение для `'MIMA'`. Оно равно `3`. Мы перемножаем значения и получаем $2 \cdot 3 = 6$ особых пар городов. Итого теперь их 7.

Следующий ключ — 'NANH', его значение равно 1. Чтобы найти особые пары городов с комбинацией 'NANH', требуется найти значение для 'NHNA'. Вот только таких нет, поэтому и новых пар нет. Значит, их по-прежнему 7.

Теперь будьте внимательнее. Следующий ключ — 'PASC', его значение равно 1. Чтобы найти специальные пары городов с комбинацией 'PASC', требуется найти значение для 'SCPA'. Оно также равно 1. Мы перемножаем значения, получаем $1 \cdot 1 = 1$ особую пару городов, включающую эти комбинации. Но подождите, мы уже учли эту пару, когда обрабатывали ключ 'SCPA'. Если мы добавим ее снова, то она будет учтена дважды. Фактически, обрабатывая все ключи, мы всегда дважды учитываем каждую особую пару городов. Но не волнуйтесь, мы исправим это позже, когда будем выводить окончательный ответ. А пока прибавим 1 к ранее найденным 7, в результате получая 8.

Следующий ключ — 'LACO', его значение равно 2. Значение 'COLA' равно 1, что дает $2 \cdot 1 = 2$ особые пары городов. Теперь у нас их 10.

Осталось два ключа: 'MIMA' и 'COLA'. Первый добавляет к общей сумме 6 новых пар, а второй — 2 пары. С учетом 10, которые мы нашли ранее, теперь их 18.

Помните, что мы дважды учли каждую особую пару городов. Значит, у нас не 18, а $18 / 2 = 9$ особых пар городов. Чтобы получить правильный ответ, достаточно разделить результат на 2.

Если вы сравните словарь, который мы только что просмотрели, с городами в тестовом примере, то заметите, что в словаре чего-то не хватает. Это город WASHOUGAL WA. Его комбинация — WAWA, но в нашем словаре нет ключа 'WAWA'. Мы не учитываем этот город, и вот почему. Первые два символа WASHOUGAL WA — WA. Это означает, что для образования особой пары нужен еще один город с тем же началом в штате WA. И сам город WASHOUGAL WA тоже находится в штате WA. Но в задаче сказано, что два города в особой паре должны быть из разных штатов. Следовательно, мы не сможем найти особую пару городов для WASHOUGAL WA. Чтобы случайно не насчитать лишнего, не включаем WASHOUGAL WA в словарь.

Решение задачи

Можно приступать к решению задачи! Для этого можем использовать словарь. Код приведен в листинге 8.7.

Листинг 8.7. Решение задачи

```
input_file = open('citystate.in', 'r')
output_file = open('citystate.out', 'w')

n = int(input_file.readline())
```

```
❶ combo_to_num = {}

  for i in range(n):
      lst = input_file.readline().split()
   ❷ city = lst[0][:2]
      state = lst[1]
   ❸ if city != state:
          combo = city + state
          if not combo in combo_to_num:
              combo_to_num[combo] = 1
          else:
              combo_to_num[combo] = combo_to_num[combo] + 1

  total = 0

❹ for combo in combo_to_num:
   ❺ other_combo = combo[2:] + combo[:2]
         if other_combo in combo_to_num:
          ❻ total = total + combo_to_num[combo] * combo_to_num[other_combo]

❼ output_file.write(str(total // 2) + '\n')
  input_file.close()
  output_file.close()
```

В этой задаче нужно использовать файлы, а не стандартный ввод и стандартный вывод.

Словарь, который мы создадим, называется `combo_to_num` ❶. Он отображает комбинации из четырех символов, такие как `'SCPA'`, на количество городов с этой комбинацией.

Для каждого города из входных данных мы задействуем переменные, в которых хранятся первые два символа названия города ❷ и его штат. Затем, если эти значения не совпадают ❸, объединяем их и добавляем комбинацию в словарь. Если в нем такого ключа еще не было, добавляем его со значением 1, а если был, увеличиваем его значение на 1.

Словарь готов. Теперь перебираем его ключи ❹. Для каждого из них создаем комбинацию, которую нужно искать, чтобы определить особые пары. Если, например, ключом является `'SCPA'`, то другая комбинация должна быть `'PASC'`. Для этого мы берем два крайних правых символа ключа и приставляем справа от них два левых символа ❺. Если такая комбинация также есть в словаре, то перемножаем значения двух ключей и прибавляем их к сумме ❻.

Теперь нам осталось лишь вывести в выходной файл общее количество особых пар городов. Как объяснялось в предыдущем разделе, нужно разделить полученную сумму на 2 ❼, чтобы компенсировать двойной подсчет, возникающий в результате обработки каждого ключа в словаре.

Вот и все, еще один пример решения задачи с помощью словарей. Можете смело отправлять код!

Резюме

В этой главе вы узнали о множествах и словарях Python. Множество — это совокупность значений без порядка и дублирования. Словарь — набор пар «ключ:значение». Как вы видели в задачах главы, иногда эти коллекции лучше подходят для решения, чем списки. Например, определение того, входит ли значение в множество, выполняется невероятно быстро по сравнению с той же операцией в списке. Если неважен порядок значений или нужно исключить дубликаты, стоит подумать о том, можно ли использовать множество.

Словарь тоже позволяет легко определить значение, сопоставленное с ключом. Если в задаче нужно отображение ключей в значения, стоит рассмотреть возможность применения словаря.

Благодаря множествам и словарям у вас появились новые способы хранить свои значения. Однако такая гибкость означает, что придется делать выбор. Не задействуйте списки, не рассмотрев альтернативу! Разница между использованием того или иного типа может определить, решите вы задачу или нет.

Мы дошли до важного этапа, и вы уже изучили большую часть функций языка Python, которые я собирался представить в этой книге. Это не означает, что ваше путешествие в мир Python завершено, так как узнать вам предстоит гораздо больше. А еще это означает, что к настоящему моменту вы узнали достаточно для того, чтобы решить немало задач как из соревновательного программирования, так и из реальной практики.

В следующей главе книги мы переключим передачу и перейдем от изучения новых функций Python к улучшению навыков решения задач. Сосредоточимся на одном типе задач, которые можно решить несколькими способами.

Упражнения

Далее приведены несколько упражнений, которые вы можете попробовать выполнить. Для решения этих задач используйте множество или словарь. Иногда множества или словари позволяют написать код, который работает быстрее или получается более строгим и легким для чтения.

1. DMOJ, задача Bard с кодом `crci06p1`.
2. DMOJ, задача Conspicuous Cryptic Checklist с кодом `dmopc19c5p1`.
3. DMOJ, задача Marko с кодом `coci15c2p1`.
4. DMOJ, задача Attack of the CipherTexts с кодом `ccc06s2`.
5. DMOJ, задача Mode Finding с кодом `dmopc19c3p1`.
6. DMOJ, задача Utrka с кодом `coci14c2p2`. (Попробуйте решить эту задачу тремя способами: с помощью словаря, множества или списков!)

7. DMOJ, задача ZigZag с кодом `coci17c2p2`. (Подсказка: вам нужны будут два словаря. Первый отображает каждую начальную букву со списком слов, а второй — каждую начальную букву с индексом следующего слова, которое будет выводиться. Таким образом вы сможете циклически перебирать слова для каждой буквы без необходимости явно обновлять количество вхождений или изменять списки.)

Примечания

Задача «Адреса электронной почты» взята из второго тура конкурса Ontario Programming Contest 2019 года, Educational Computing Organization. Задача «Общие слова» взята из Canadian Computing Olympiad 1999 года. Задача «Города и штаты» взята из конкурса USACO 2016 December Silver Contest.

Если вы хотите узнать больше о Python, я рекомендую почитать книгу Эрика Маттеса *Python Crash Course*, 2-е издание (No Starch Press, 2019). Когда будете готовы перейти на следующий уровень, можете прочитать книгу Бретта Слаткина *Effective Python*, 2-е издание (Addison-Wesley Professional, 2020), где приведен сборник советов, которые помогут вам лучше писать код на Python.

9

Разработка алгоритмов полного поиска

Алгоритм — это последовательность шагов, которая решает некоторую задачу. Для решения задач, представленных в книге, мы писали алгоритмы в виде кода Python. В этой главе сосредоточимся на разработке алгоритмов. Столкнувшись с новой задачей, иногда сложно с ходу понять, как ее решить. И как тогда написать алгоритм? К счастью, не нужно каждый раз начинать с нуля. Ученые-информатики и программисты давно придумали несколько общих типов алгоритмов, и вполне вероятно, что по крайней мере один из них подойдет для вашей задачи.

Один из таких алгоритмов называется алгоритмом *полного поиска*, он заключается в переборе всех возможных решений и выборе лучшего. Например, если в какой-то задаче требуется найти максимум, мы перебираем все решения и находим наибольшее число, а если нужно найти минимум — выбираем наименьшее. Алгоритмы полного поиска известны также как алгоритмы *перебора*, но я буду избегать этого термина. Компьютер-то действительно выполняет перебор, проверяя решение за решением, но это не считается методом грубой силы с точки зрения алгоритмов.

Мы использовали алгоритм полного поиска для решения задачи «Деревни у дороги» в главе 5. Там требовалось определить наименьший размер окрестностей деревни, и мы нашли решение, просматривая каждую территорию и запоминая размер самой маленькой из встреченных. В этой главе для решения задач снова будем задействовать алгоритмы полного поиска. Мы увидим, сколько усилий требует само определение того, что именно нужно искать.

С помощью полного поиска решим две задачи: определим спасателя, которого нужно уволить, и вычислим минимальные затраты на выполнение требований для тренировочного лыжного лагеря. Затем рассмотрим третью задачу — подсчет троек коров, которые соответствуют наблюдениям. Вот тут придется подумать.

Задача 21. Спасатели

В этой задаче нужно определить, какого спасателя уволить, чтобы при этом расписание работы бассейна осталось максимально охвачено. Мы попробуем уволить каждого и посмотреть, что получится, — это и есть алгоритм полного поиска!

Задача с конкурса USACO 2018 January Bronze Contest под названием «Спасатели».

Постановка задачи

Фермер Джон купил для своих коров бассейн. Он открыт в период времени с 0 до 1000.

Для наблюдения за бассейном Джон нанимает n спасателей. Каждый из них следит за бассейном в течение заданного интервала времени. Например, спасатель может начать работу в момент времени 2 и закончить в момент 7. Такой интервал будет обозначен как 2–7. Длительность времени работы будет равняться времени окончания минус время начала. Например, спасатель, работающий с 2 до 7, охватывает 5 единиц времени, то есть с 2 до 3, с 3 до 4, с 4 до 5, с 5 до 6 и с 6 до 7.

К сожалению, фермер Джон может оплатить работу лишь $n - 1$ спасателя, поэтому одного нужно уволить.

Определите максимальное количество единиц времени, которое можно охватить после увольнения одного спасателя.

Входные данные

Входные данные читаются из файла `lifeguards.in`. Они состоят:

- из строки, содержащей число n — количество нанятых спасателей от 1 до 100;
- n строк, в каждой из которой указаны время начала работы спасателя, интервал и время окончания работы. Время начала и окончания — целые числа от 0 до 1000, обязательно отличные друг от друга.

Выходные данные

Выходные данные требуется записать в файл `lifeguards.out`.

Выведите максимальное количество единиц времени, которое могут охватить $n - 1$ спасатель.

Ограничение времени на каждый тестовый пример — 4 секунды.

Тестовый пример

Рассмотрим тестовый пример, который поможет понять, почему алгоритм полного поиска для этой задачи будет удобнее. Данные примера:

```
4
5 8
10 15
17 25
9 20
```

Первый способ, который вы можете испытать для решения этой задачи, — уволить спасателя с кратчайшим сроком работы. Интуитивно такое решение напрашивается, потому что кажется, что он меньше всех участвует в наблюдении за бассейном.

Даст ли этот метод правильный алгоритм? Посмотрим. Если исходить из таких соображений, нужно уволить спасателя 5–8, так как у него самый короткий рабочий интервал. Остаются три спасателя с интервалами 10–15, 17–25 и 9–20. Эти три спасателя охватывают интервал 9–25, длительность которого составляет 25 – 9 = 16 единиц времени. Получается, 16 — это правильный ответ?

К сожалению, нет. Оказывается, надо было уволить спасателя 10–15. В этом случае у нас останутся три спасателя с временными интервалами 5–8, 17–25 и 9–20. Они охватывают интервалы 5–8 и 9–25 (важно заметить, что интервал времени от 8 до 9 пропускается). Первый из них составляет 8 – 5 = 3 единицы времени, а второй — 25 – 9 = 16 единиц, всего получается 19 единиц времени.

Правильный ответ — 19, а не 16. Увольнение спасателя с наименьшим временем работы не дает правильного ответа.

Не всегда получается с ходу найти простое правило, которое позволило бы решить эту задачу. Однако нам беспокоиться не о чем, так как, применяя алгоритм полного поиска, мы можем вообще ничего не придумывать.

Далее показано, как алгоритм полного поиска найдет решение тестового примера.

1. Сперва уволим первого спасателя и посчитаем количество времени, которое охватывают трое оставшихся. Получим значение 16 и запомним его как лучший результат.
2. Затем уволим второго спасателя и посчитаем количество времени, в течение которого работают трое оставшихся. Получим значение 19 и запомним его как лучший из имеющихся результатов.
3. Затем уволим третьего спасателя и посчитаем, сколько времени трудятся трое оставшихся. Получим значение 14. Результат 19 остается лучшим.
4. Наконец, уволим четвертого спасателя и посчитаем количество времени, которое охватывают трое оставшихся. Получим ответ 16. Значит, 19 — лучший результат.

Рассмотрев последствия увольнения каждого спасателя, алгоритм делает вывод, что 19 — это правильный ответ. Лучшего ответа быть не может, потому что мы перепробовали все варианты, то есть выполнили полный поиск среди возможных решений.

Решение задачи

Чтобы реализовать полный поиск, часто бывает полезно начать с написания функции, которая проверяет один конкретный вариант решения. Затем мы вызовем эту функцию для каждого варианта.

Увольнение одного спасателя

Давайте напишем функцию для определения количества единиц времени, которые окажутся охваченными при увольнении одного конкретного спасателя. Код приведен в листинге 9.1.

Листинг 9.1. Решение задачи, когда уволен один конкретный спасатель

```
def num_covered(intervals, fired):
    """
    intervals — это список интервалов работы спасателей;
    каждый интервал представляет собой список вида
    [start, end].
    fired — это индекс увольняемого спасателя.

    Возвращает число интервалов времени,
    охваченных спасателями, не считая уволенного.
    """
❶   covered = set()
    for i in range(len(intervals)):
        if i != fired:
            interval = intervals[i]
        ❷ for j in range(interval[0], interval[1]):
            ❸ covered.add(j)
    return len(covered)
```

Первый параметр — это список временных интервалов спасателей, второй — индекс спасателя, которого мы увольняем. Введите этот код в оболочку Python. Рассмотрим два примера вызова функции:

```
>>> num_covered([[5, 8], [10, 15], [9, 20], [17, 25]], 0)
16
>>> num_covered([[5, 8], [10, 15], [9, 20], [17, 25]], 1)
19
```

Вызовы функции показывают, что мы можем охватить 16 единиц времени, если уволим спасателя 0, и 19 единиц времени, если уволим спасателя 1.

Теперь давайте разберемся, как работает эта функция. Сперва создадим множество, которое будет содержать охваченные спасателями промежутки времени ❶. Если определенный интервал охвачен, код добавляет его начальное значение. Например, если охвачен промежуток от 0 до 1, в множество добавляется 0, если от 4 до 5 — добавляется 4.

Последовательно перебираем временные интервалы спасателя. Если он не уволен, мы рассматриваем его временной интервал ❷ и каждую единицу охваченного им времени, добавляя каждую из охваченных единиц времени в множество ❸. Напомню, что наборы не сохраняют повторяющиеся значения, поэтому ничего страшного не случится, если мы попробуем добавить одну и ту же единицу времени несколько раз. Перебрав всех неуволенных спасателей, получаем множество всех охваченных единиц времени.

В конце программы возвращаем количество значений в наборе.

Основная программа

Основная часть программы приведена в листинге 9.2. В ней мы используем функцию `num_covered`, определяющую количество единиц времени, которые оказываются охваченными при увольнении каждого спасателя. Для полного решения задачи обязательно добавьте код функции `num_covered` (см. листинг 9.1) перед приведенным далее кодом.

Листинг 9.2. Основная программа

```
input_file = open('lifeguards.in', 'r')
output_file = open('lifeguards.out', 'w')

n = int(input_file.readline())

intervals = []

for i in range(n):
 ❶ interval = input_file.readline().split()
    interval[0] = int(interval[0])
    interval[1] = int(interval[1])
    intervals.append(interval)

  max_covered = 0

❷ for fired in range(n):
  ❸ result = num_covered(intervals, fired)
    if result > max_covered:
        max_covered = result

output_file.write(str(max_covered) + '\n')
input_file.close()
output_file.close()
```

В этой задаче мы работаем с файлами, а не со стандартными вводом и выводом.

Программа начинается с чтения числа спасателей, а затем с помощью цикла `for-range` мы считываем временные интервалы всех спасателей. Считав каждый временной интервал ❶, преобразуем границы интервала в целые числа и добавляем их в виде списка из двух значений в сводный список интервалов.

В переменной `max_covered` будем хранить максимальное количество единиц времени, которые могут быть охвачены.

Затем начинаем поочередно увольнять каждого спасателя, используя цикл `for-range` ❷. С помощью функции `num_covered` ❸ определяем количество охваченных единиц времени с учетом увольнения одного спасателя. Мы обновляем переменную `max_covered` всякий раз, когда удается охватить больший промежуток времени.

По завершении цикла все варианты увольнений оказываются проверенными и ответ находится в переменной `max_covered`. Мы выводим этот максимум как решение задачи.

Этот код можно отправлять на сайт USACO. Для кода Python используется ограничение по времени 4 секунды на тестовый пример, но наше решение гораздо быстрее. Например, я только что запустил приведенный код, и каждый тестовый пример был выполнен не более чем за 130 миллисекунд.

Эффективность программы

Причина быстродействия нашего кода заключается в том, что спасателей всегда мало — не более 100. Если бы их было много, то код перестал бы укладываться в отведенные сроки. Несколько сотен спасателей мы обработаем без проблем. Более или менее справимся, если получим от 3000 до 4000 спасателей. Но если их будет больше, то код станет слишком медленным. Например, обработать 5000 спасателей алгоритм может уже не успеть, и тогда нужно будет разработать новый алгоритм, в котором использовалось бы нечто более быстрое, чем полный поиск.

Вы можете подумать, что 5000 — это слишком много спасателей и даже хорошо, что наш алгоритм подобным не занимается. Но это не так! Вспомните задачу с адресами электронной почты из главы 8. В ней мы обрабатывали 100 000 адресов электронной почты. А в задаче «Города и штаты» в той же главе во входных данных могло быть до 200 000 городов. И вот уже 5000 спасателей кажется не таким уж большим числом.

Решения методом полного поиска хорошо работают при небольшом объеме входных данных. Но на больших тестовых примерах методы полного поиска часто не срабатывают.

Причина медлительности решения методом полного поиска на больших тестовых данных заключается в том, что код выполняет много повторяющейся работы. Представьте, что в тестовом примере будет 5000 спасателей. Мы так же увольняем спасателя 0 и вызываем функцию `num_covered`, чтобы определить количество единиц времени, охваченных всеми оставшимися спасателями. Затем уволим спасателя 1 и сделаем то же самое. Получается, что функция `num_covered` повторяет все то, что делала на предыдущем вызове, с минимальной разницей. Единственное различие состоит в том, что спасатель 0 вернулся, а спасатель 1 уволен, но состояние остальных 4998 спасателей осталось прежним. Функция `num_covered` этого не знает и перебирает всех спасателей заново. То же самое происходит, когда мы увольняем спасателя 2, затем спасателя 3 и т. д. Каждый раз `num_covered` выполняет всю работу с нуля, не зная о том, что все давно посчитано.

Помните, что алгоритмы полного поиска имеют ограничения, хотя и бывают полезными. Когда мы решаем какую-то новую задачу, алгоритм полного поиска может стать хорошим началом, даже если в итоге окажется слишком неэффективным. Все дело в том, что в процессе разработки алгоритма вы сами глубже начинаете понимать задачу и находите новые способы решения.

В следующем разделе рассмотрим еще одну задачу, в которой можно использовать полный поиск.

ПРОВЕРИМ ЗНАНИЯ

Правильно ли работает вот такая версия функции `num_cover`?

```
def num_covered(intervals, fired):
    """
    intervals — это список интервалов работы спасателей;
    каждый интервал представляет собой список вида
    [start, end].
    fired — это индекс увольняемого спасателя.

    Возвращает число интервалов времени,
    охваченных спасателями, не считая уволенного.
    """
    covered = set()
    intervals.pop(fired)
    for interval in intervals:
        for j in range(interval[0], interval[1]):
            covered.add(j)
    return len(covered)
```

А. Да.

Б. Нет.

Ответ: **Б.** Эта функция удаляет уволенного из списка спасателей. Этого делать нельзя, и в строке документации ничего не говорится о том, что список должен меняться. С такой версией функции программа многие примеры обработает неверно, потому что информация о спасателях по ходу выполнения теряется. Например, когда мы тестируем спасателя 0, он удаляется из списка. Позже, когда будем проверять спасателя 1, спасателя 0 в списке уже не будет! Если вы хотите использовать версию функции, в которой уволенный спасатель удаляется из списка, вам нужно работать с копией списка, а не с оригиналом.

Задача 22. Лыжная база

Иногда само описание задачи говорит о том, что решение можно найти методом полного поиска. Например, в задаче «Спасатели» требовалось уволить одного спасателя, поэтому мы пробовали уволить каждого. Иногда приходится чуть пораскинуть мозгами, чтобы понять, что мы вообще ищем. Читая следующую задачу, подумайте, что бы вы искали в решении для полного поиска.

Это задача Ski Course Design с конкурса USACO 2014, бронзовый конкурс.

Постановка задачи

У Джона на ферме есть n холмов, каждый высотой от 0 до 100. Он хотел бы зарегистрировать ее как тренировочный лыжный лагерь.

Ферма может быть зарегистрирована как тренировочный лыжный лагерь только в том случае, если разница в высоте между самым высоким и самым низким холмами составляет не более 17. В связи с этим фермеру Джону, возможно, придется увеличить высоту одних холмов и уменьшить высоту других. Изменять высоту можно только на целые числа.

Стоимость изменения высоты холма на x единиц равна x^2. Например, изменение высоты холма с 1 на 4 стоит $(4 - 1)^2 = 9$.

Определите минимальную сумму, которую Джон должен будет потратить на изменение высот холмов, чтобы можно было зарегистрировать ферму как лыжный лагерь.

Входные данные

Входные данные читаются из файла `skidesign.in`. Они состоят:

- из строки, содержащей целое число n — количество холмов на ферме в диапазоне от 1 до 1000;

- n строк, каждая из которых содержит высоту холма. Высота — это целое число от 0 до 100.

Выходные данные

Выходные данные записываются в файл `skidesign.out`.

Выведите минимальную сумму, которую фермер Джон должен будет затратить на изменение высот холмов.

Ограничение по времени для тестового примера — 4 секунды.

Тестовый пример

Посмотрим, сможем ли мы применить к этой проблеме опыт, полученный при решении задачи «Спасатели», в которой мы отдельно увольняли каждого спасателя, чтобы понять, кого нужно уволить в итоге. Можем ли мы сделать с каждым холмом нечто похожее, чтобы решить задачу «Лыжная база»? Например, можно попробовать принять высоту каждого холма в качестве нижнего предела в допустимом диапазоне высот.

Опробуем этот метод на тестовом примере:

```
4
23
40
16
2
```

Наименьшая высота среди этих четырех холмов — 2, а наибольшая — 40.

Разница между 40 и 2 составляет 38, то есть больше 17. Фермер Джон должен привести холмы в порядок, но за это придется заплатить!

Первый холм имеет высоту 23. Если мы примем 23 за нижний предел диапазона, тогда верхний предел будет 23 + 17 = 40. Рассчитаем цену приведения всех холмов в диапазон 23–40. Есть два холма, которые находятся за пределами этого диапазона: высотой 16 и 2. Увеличение их до 23 стоит $(23 - 16)^2 + (23 - 2)^2 = 490$. Нужно найти более выгодный вариант.

Второй холм имеет высоту 40. Верхний предел для такого диапазона 40 + 17 = 57, поэтому нужно привести все холмы в диапазон 40–57. Все остальные холмы находятся за его пределами, поэтому каждый из них вносит свой вклад в общую стоимость. Итого $(40 - 23)^2 + (40 - 16)^2 + (40 - 2)^2 = 2309$. Это больше, чем текущая минимальная стоимость 490, поэтому нового результата пока нет (помните, что в этой задаче мы пытаемся минимизировать затраты фермера Джона, тогда как в «Спасателях» пытались максимизировать охват).

Третий холм имеет высоту 16, что дает диапазон 16–33. За пределами этого диапазона два холма высотой 40 и 2. Стоимость для этого диапазона составляет $(40 - 33)^2 + (16 - 2)^2 = 245$. Это наш новый рекорд, который нужно побить!

Четвертый холм имеет высоту 2, что дает диапазон 2–19. Стоимость приведения холмов к этому диапазону равна 457.

Минимальная стоимость, которую мы получили с помощью данного алгоритма, составляет 245. Правильный ли это ответ?

Нет и нет! Оказывается, минимальная стоимость составляет 221. Есть два диапазона, которые дают ее: 12–29 и 13–30. Но холма высотой 12 или 13 нет. Это значит, что в качестве возможной нижней границы диапазона использовать имеющиеся холмы нельзя.

Подумав, мы поймем, что правильный алгоритм полного поиска ни в коем случае не должен пропустить ни одного диапазона.

Сформулируем план, который гарантированно даст нам правильный ответ. Начнем с расчета стоимости для диапазона 0–17. Затем рассчитываем ее для диапазона 1–18. Потом для 2–19, 3–20, 4–21 и т. д. Мы проверяем каждый возможный диапазон и запоминаем минимальную стоимость, которую удалось получить. Проверяемые диапазоны не связаны с высотой имеющихся холмов напрямую. Поскольку мы тестируем все возможные диапазоны, среди них обязательно найдется лучший.

Какие диапазоны нужно проверять? В какой момент останавливать перебор? Нужно ли проверять диапазон 50–67? Да. А диапазон 71–88? Тоже да. Как насчет 115–132? Нет! Его не надо.

Последний диапазон, который нужно проверить, — это 100–117. Все дело в том, что в описании задачи сказано: высота любого холма составляет не более 100.

Предположим, мы вычислим стоимость для диапазона 101–118. Даже не зная высот имеющихся холмов, мы точно знаем, что ни один из них не входит в этот диапазон, так как он начинается со 101, а максимальная высота холма составляет 100. Перейдем на единицу ниже, к 100–117. Диапазон 100–117 стоит меньше, чем диапазон 101–118, так как имеющиеся холмы ближе к 100, чем к 101.

Для примера рассмотрим холм высотой 80. Чтобы поднять его на высоту 101, нужна 441 единица денег, а на высоту 100 — 400 единиц. Получается, диапазон 101–118 даже нет смысла проверять.

Мы показали, что бессмысленно проверять все более высокие диапазоны, так как всегда найдется вариант дешевле.

Таким образом, нам нужно проверить ровно 101 диапазон: 0–17, 1–18, 2–19 и так далее вплоть до 100–117. Из найденных вариантов запомним лучший!

Решение задачи

Мы решим задачу в два этапа, как и «Спасатели». Начнем с функции для определения стоимости одного диапазона. Затем напишем основную программу, которая будет вызывать эту функцию для каждого диапазона.

Определение стоимости одного диапазона

В листинге 9.3 приведен код функции, которая определяет стоимость данного диапазона.

Листинг 9.3. Нахождение стоимости одного конкретного диапазона

```
MAX_DIFFERENCE = 17
MAX_HEIGHT = 100

def cost_for_range(heights, low, high):
    """
    heights – это список высот холмов.
    low – это нижняя граница диапазона высот.
    high - это верхняя граница диапазона высот.

    Возвращает стоимость приведения высот всех
    холмов в заданный диапазон.
    """
    cost = 0
❶   for height in heights:
❷       if height < low:
❸           cost = cost + (low - height) ** 2
❹       elif height > high:
❺           cost = cost + (height - high) ** 2
    return cost
```

Я добавил в код две константы, которые нам понадобятся позже. Константа `MAX_DIFFERENCE` хранит максимальную допустимую разницу между самым высоким и самым низким холмами. Константа `MAX_HEIGHT` хранит максимально возможную высоту холма.

Теперь рассмотрим функцию `cost_for_range`. Она принимает список высот холмов и желаемый диапазон, определяемый нижним и верхним пределами. Функция возвращает стоимость изменения высот холмов так, чтобы все холмы оказались в желаемом диапазоне. Попробуйте ввести код функции в оболочку Python и проверить ее работу, прежде чем продолжать.

Функция перебирает высоты всех холмов ❶ и суммирует стоимость их приведения к желаемому диапазону. Требуется учесть два случая. В первом случае высота текущего холма может лежать вне допустимого диапазона, если она

меньше значения `low` ❷. Выражение `low - height` позволяет получить высоту, на которую нужно увеличить холм, после чего мы возведем результат в квадрат, чтобы получить стоимость ❸. Во втором случае высота текущего холма может быть больше `high` ❹. Выражение `height - high` даст нам значение, которое нужно вычесть из высоты этого холма, после чего мы возведем результат в квадрат, чтобы получить стоимость ❺. Обратите внимание: если высота уже находится в нужном диапазоне, делать ничего не нужно. Перебрав все холмы, возвращаем итоговую стоимость.

Основная программа

Основная часть нашей программы приведена в листинге 9.4. В ней результат работы функции `cost_for_range` используется для определения стоимости каждого диапазона. Обязательно допишите функцию `cost_for_range` (см. листинг 9.3) перед этим кодом.

Листинг 9.4. Основная программа

```
input_file = open('skidesign.in', 'r')
output_file = open('skidesign.out', 'w')

n = int(input_file.readline())

heights = []

for i in range(n):
    heights.append(int(input_file.readline()))

❶ min_cost = cost_for_range(heights, 0, MAX_DIFFERENCE)

❷ for low in range(1, MAX_HEIGHT + 1):
    result = cost_for_range(heights, low, low + MAX_DIFFERENCE)
    if result < min_cost:
        min_cost = result

output_file.write(str(min_cost) + '\n')

input_file.close()
output_file.close()
```

Сперва мы считываем количество холмов, а затем все их высоты.

В переменной `min_cost` будем хранить минимальную стоимость, которую удалось найти к данному моменту. Изначально в переменную `min_cost` помещаем стоимость для диапазона 0–17 ❶. Затем в цикле ❷ перебираем все остальные возможные диапазоны, обновляя значение `min_cost` каждый раз, когда удастся

найти меньшую стоимость. По окончании цикла выводим минимальную стоимость из найденных.

Пришло время отправить код на сайт. Решение методом полного поиска должно успеть выполнить задачу в срок.

Следующая задача — это пример недостаточно эффективной работы метода прямого поиска.

ПРОВЕРИМ ЗНАНИЯ

Предлагается внести изменение в код из листинга 9.4. Возьмем строку:

```
for low in range(1, MAX_HEIGHT + 1):
```

и заменим ее вот такой:

```
for low in range(1, MAX_HEIGHT - MAX_DIFFERENCE + 1):
```

По-прежнему ли код работает правильно?

А. Да.

Б. Нет.

Ответ: **А**. Последний диапазон, который сейчас проверяется кодом, — это 83–100, поэтому надо быть уверенными, что диапазоны, которые мы перестали проверять, а именно 84–101, 85–102 и т. д., больше не нужны.

Рассмотрим диапазон 84–101. Если мы докажем, что диапазон 83–100 всегда не хуже, чем 84–101, то у нас не будет причин проверять диапазон 84–101.

Диапазон 84–101 включает высоту 101. Однако это бессмысленно, ведь высота самого высокого холма — 100, поэтому высота 101 нам не нужна. Мы можем удалить из рассмотрения 101, не ухудшая результат. Если сделаем это, останется диапазон 84–100. Но тогда разница высот равна 16, а по условию может быть 17. Поэтому мы можем подвинуть нижнюю часть диапазона, получив 83–100. Расширение диапазона не может сделать результат хуже, так как теперь высоты всех холмов становятся ближе к допустимой.

Аналогичные выводы можно сделать относительно всех более высоких диапазонов, тогда нет смысла проверять диапазоны выше 83–100.

Прежде чем продолжить, можете попробовать выполнить упражнения 1 и 2, приведенные в конце главы.

Задача 23. Коровий бейсбол

В конце главы рассмотрим задачу, в которой придется усовершенствовать наши навыки разработки алгоритмов и узнать о чем-то, помимо полного поиска. Читая задачу, обратите внимание на то, что входных данных в ней не так уж много. Обычно малое количество входных данных свидетельствует об эффективности алгоритма полного поиска, но не в этот раз, так как в задаче нужно будет перебирать очень много данных. Сложность алгоритма обусловлена слишком большим количеством вложенных циклов. А чем нам вредят вложенные циклы? Что с этим можно поделать? Об этом и поговорим!

Задача «Коровий бейсбол» взята с конкурса USACO 2013 December Bronze Contest.

Постановка задачи

У фермера Джона есть n коров. Они стоят в ряд на некотором целочисленном расстоянии друг от друга. Коровы развлекаются, бросая друг другу бейсбольный мяч.

Джон наблюдает за процессом. Он замечает, что корова x бросает мяч какой-нибудь корове y, стоящей справа, а та в свою очередь отправляет его также стоящей справа корове z. Еще он знает, что длина второго броска должна быть не менее длины первого броска и не превышать ее более чем вдвое (например, если первый бросок выполнен на расстояние 5, то длина второго броска может быть от 5 до 10).

Определите количество троек коров (x, y, z), удовлетворяющих наблюдениям Джона.

Входные данные

Входные данные читаются из файла `baseball.in`. Они состоят:

- из строки, содержащей число n — количество коров от 3 до 1000;
- n строк, которые задают положение коровы. Все позиции уникальны, и каждая составляет от 1 до 100 000 000.

Выходные данные

Выходные данные записываются в файл `baseball.out`.

Выведите количество троек коров, удовлетворяющих наблюдениям фермера Джона.

Ограничение по времени для решения каждого тестового примера — 4 секунды.

Использование трех вложенных циклов

Мы можем применить три вложенных цикла, чтобы рассмотреть все возможные тройки коров. Сперва рассмотрим код, а затем обсудим его эффективность.

Код

В главе 3 мы узнали, что можно перебрать все пары значений с помощью двух вложенных циклов. Выглядит это так:

```
>>> lst = [1, 9]
>>> for num1 in lst:
...     for num2 in lst:
...         print(num1, num2)
...
1 1
1 9
9 1
9 9
```

Аналогичным образом можно перебрать тройки значений, используя три вложенных цикла, например:

```
>>> for num1 in lst:
...     for num2 in lst:
...         for num3 in lst:
...             print(num1, num2, num3)
...
1 1 1
1 1 9
1 9 1
1 9 9
9 1 1
9 1 9
9 9 1
9 9 9
```

Именно три вложенных цикла станут отправной точкой для решения задачи «Коровий бейсбол». Мы можем проверить, соответствует ли каждая тройка наблюдениям фермера Джона. Код приведен в листинге 9.5.

Листинг 9.5. Использование трех вложенных циклов for

```
input_file = open('baseball.in', 'r')
output_file = open('baseball.out', 'w')

n = int(input_file.readline())

positions = []
```

```
for i in range(n):
    positions.append(int(input_file.readline()))  ❶

total = 0

for position1 in positions:  ❷
    for position2 in positions:  ❸
        first_two_diff = position2 - position1
        if first_two_diff > 0:  ❹
            low = position2 + first_two_diff
            high = position2 + first_two_diff * 2

            for position3 in positions:  ❺
                if position3 >= low and position3 <= high:
                    total = total + 1

output_file.write(str(total) + '\n')

input_file.close()
output_file.close()
```

В этом коде мы читаем все позиции коров в списке `positions` ❶. Затем перебираем их с помощью цикла `for` ❷. Для каждой из этих позиций перебираем все позиции в списке, используя вложенный цикл `for` ❸. В данный момент `position1` и `position2` ссылаются на две позиции из списка. Нам нужен третий вложенный цикл, но сперва нужно вычислить разницу между `position1` и `position2`, потому что она позволит получить диапазон возможных значений `position3`.

Из описания задачи следует, что позиция 2 должна располагаться правее позиции 1. Если это условие выполняется ❹, мы вычисляем нижний и верхний пределы диапазона для позиции 3 и сохраняем граничные значения в переменные `low` и `high`. Например, если `position1` равна 1, а `position2` — 6, получаем `low` $6 + 5 = 11$ и для `high` значение $6 + 5 \cdot 2 = 16$. Затем перебираем список с третьим вложенным циклом `for` ❺ и ищем позиции, которые находятся между `low` и `high`. Для каждой такой `position3` мы увеличиваем сумму на 1.

С помощью трех вложенных циклов мы вычислили общее количество троек. В конце выводим полученное число в выходной файл.

Попробуем выполнить программу на небольшом тестовом примере, чтобы убедиться, что все работает правильно. Пример:

```
7
16
14
23
18
1
6
11
```

Правильный ответ для этого тестового примера — 11 подходящих троек:

- 14, 16, 18;
- 14, 18, 23;
- 1, 6, 16;
- 1, 6, 14;
- 1, 6, 11;
- 1, 11, 23;
- 6, 14, 23;
- 6, 11, 16;
- 6, 11, 18;
- 11, 16, 23;
- 11, 14, 18.

Вот и здорово, ведь наша программа выводит правильный ответ, так как находит все удовлетворяющие условию тройки. Например, в какой-то момент `position1` будет равна 14, `position2` — 16, а `position3` — 18. Эта тройка удовлетворяет требованиям, поэтому входит в общую сумму. Позже, когда `position1` равна 18, `position2` — 16, а `position3` — 14, ничего не произойдет, так как в этом случае броски делаются не вправо. В нашей программе оператор `if` предотвращает обработку таких троек.

Итак, программа правильная. Но, отправив ее на сайт, вы увидите, что код недостаточно эффективен. В этой задаче и вообще во многих задачах соревновательного программирования первые несколько тестовых примеров небольшие — всего несколько коров, спасателей или холмов. Наша программа вполне может решить такие примеры вовремя. Остальные тестовые примеры подводят ее все ближе и ближе к пределу отведенного времени, и в конце концов она перестает успевать.

Эффективность программы

Чтобы понять, почему наша программа такая медленная, попробуем подсчитать число троек, которое она перебирает. Вспомните тестовый пример с семью коровами, который мы только что рассмотрели. Сколько троек переберет программа? Для первой коровы есть семь вариантов: 16, 14, 23 и т. д. Также есть семь вариантов для второй и столько же для третьей. Перемножая их, мы видим, что программа проверяет $7 \cdot 7 \cdot 7 = 343$ тройки.

А если бы у нас было не семь, а восемь коров? Тогда программа проверяла бы $8 \cdot 8 \cdot 8 = 512$ троек.

Легко определить выражение для количества троек, перебираемых программой при любом количестве коров. Если n — это количество коров, которых может быть 7, 8, 50, 1000 и т. д., то количество троек, проверяемых программой, составляет nnn, или n^3.

Можно подставить любое количество коров вместо n, чтобы определить количество троек, которое проверяет программа. Например, для семи коров это $7^3 = 343$, а для восьми — $8^3 = 512$. Полученные числа — 343 и 512 — крошечные. Любому компьютеру потребуется не более нескольких миллисекунд, чтобы проверить столько троек. Представьте, что программа на Python может выполнять около 5 000 000 действий в секунду. Ограничение по времени для этой задачи составляет 4 секунды на тестовый пример, так что мы сможем проверить около 20 000 000 троек.

Давайте заменим n на большее число и посмотрим, что произойдет. Для 50 коров $50^3 = 125\,000$ троек. Ничего страшного: современные компьютеры легко могут проверить 125 000 объектов. Для 100 коров $100^3 = 1\,000\,000$ троек. Опять-таки не проблема. Мы можем проверить миллион вещей менее чем за секунду. Для 200 коров у нас $200^3 = 8\,000\,000$ троек. Пока укладываемся в 4 секунды, но вы уже должны начать волноваться. Количество троек быстро растет, а коров всего-то 200. Помните, что в тестовом примере их может быть до 1000.

Для 400 коров получаем $400^3 = 64\,000\,000$ троек. Это слишком много, чтобы уложиться в 4 секунды. Чтобы стало совсем обидно, давайте обсчитаем 1000 коров — максимум, который можем получить. Для 1000 коров $1000^3 = 1\,000\,000\,000$ троек. Это миллиард. Нет, мы никогда не сможем проверить такое количество троек за 4 секунды. Нужно сделать программу более эффективной.

Сортировка прежде всего

В этой задаче поможет сортировка. Сначала посмотрим, как ее применить, а затем обсудим эффективность полученного решения.

Код

Позиции коров во входных данных могут располагаться в любом порядке, и из описания задачи никак не следует, что они отсортированы. К сожалению, это приводит к тому, что программа проверяет множество троек, которые заведомо не могут удовлетворить требованиям условия. Например, проверять тройку 18, 16, 14 бессмысленно, потому что числа расположены не в порядке возрастания. Если мы перед началом анализа отсортируем позиции коров, то код никогда не будет проверять неупорядоченные тройки.

У сортировки есть и еще одно преимущество. Предположим, что `position1` — это позиция одной коровы, а `position2` — другой. Для этой пары позиций мы знаем

наименьшее и наибольшее значения `position3`, которая нас интересует. Можно использовать тот факт, что позиции отсортированы, чтобы сократить количество значений, которые нужно проверить. Прежде чем продолжить, подумайте сами, как использовать то, что позиции отсортированы, и за счет этого перебрать меньше значений.

Когда будете готовы, посмотрите код в листинге 9.6, где применена сортировка.

Листинг 9.6. Использование сортировки

```
input_file = open('baseball.in', 'r')
output_file = open('baseball.out', 'w')

n = int(input_file.readline())

positions = []

for i in range(n):
    positions.append(int(input_file.readline()))

❶ positions.sort()

total = 0

❷ for i in range(n):
    ❸ for j in range(i + 1, n):
        first_two_diff = positions[j] - positions[i]
        low = positions[j] + first_two_diff
        high = positions[j] + first_two_diff * 2

        left = j + 1
      ❹ while left < n and positions[left] < low:
            left = left + 1

        right = left
      ❺ while right < n and positions[right] <= high:
            right = right + 1

      ❻ total = total + right - left

output_file.write(str(total) + '\n')

input_file.close()
output_file.close()
```

Прежде чем приступить к поиску троек, сортируем позиции ❶.

Первый цикл перебирает все позиции, используя переменную цикла `i` ❷. На этот раз мы применим цикл `for-range`, чтобы можно было отслеживать, на каком

индексе находимся. Это полезно, потому что теперь можно взять значение `i + 1` в качестве начального индекса для второго цикла ❸. Второй цикл теперь никогда не будет понапрасну просматривать позиции, которые находятся слева от первой позиции.

Затем мы вычисляем нижнюю и верхнюю границы диапазона значений для третьей позиции.

Вместо того чтобы увеличивать общее количество на 1 каждый раз, когда находим подходящую третью позицию, мы можем найти левую и правую границы подходящих позиций, а затем просто увеличить значение `total` в одно действие. Это возможно именно потому, что список позиций отсортирован. Мы находим каждую границу с помощью цикла `while`. Первый цикл `while` находит левую границу ❹ и продолжает это делать до тех пор, пока есть позиции меньше, чем значение `low`. Когда все будет выполнено, `left` станет крайним левым индексом, позиция которого больше или равна `low`. Второй цикл `while` находит правую границу ❺. Его работа продолжается до тех пор, пока позиции меньше или равны `high`. Когда цикл закончится, в переменной `right` будет крайний правый индекс, позиция которого больше, чем `high`. Каждая из позиций слева от `high`, не включая `right`, может быть третьей позицией в тройке, содержащей позиции с индексами `i` и `j`. Чтобы учесть их, мы прибавляем к сумме значение `right - left` ❻.

Циклы `while` в этой программе устроены довольно сложно. Давайте убедимся, что точно знаем, как они работают. Мы возьмем следующий список позиций (они такие же, как те, что использовались в предыдущем разделе, но отсортированы):

```
[1, 6, 11, 14, 16, 18, 23]
```

Предположим, что `i` равно 1, а `j` равно 2, тогда две позиции в предполагаемых тройках — это 6 и 11. Поэтому для третьей позиции мы ищем позиции, которые больше или равны 16 и меньше или равны 21. Первый цикл `while` присвоит `left` значение 4, это крайний левый индекс, позиция которого больше или равна 16. Второй цикл `while` установит `right` равным 6, это крайний левый индекс, положение которого больше 21. Вычитая `left` из `right`, мы получаем 6 – 4 = 2, что означает: у нас есть две тройки, включающие позиции 6 и 11. Прежде чем продолжить, предлагаю вам проверить, как циклы `while` работают в особых случаях, например, когда нет подходящих третьих позиций или когда такая всего одна.

Мы уже добились в этом разделе значительных успехов. Получившийся код, безусловно, более эффективен, чем код из листинга 9.5. Однако он все еще недостаточно эффективен. Если вы отправите код на сайт, то увидите, что он ушел ненамного дальше по сравнению с предыдущим разом. В большинстве тестовых случаев он по-прежнему не работает.

Эффективность нашей программы

Проблема программы заключается в том, что поиск третьей позиции может занять немало времени. Введенные нами циклы `while` все еще несколько неэффективны. Покажу это на примере нового списка позиций, представленных числами от 1 до 32:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
```

Рассмотрим подробнее случай, когда `i` равно 0, а `j` равно 7 — это позиции 1 и 8. Для третьей позиции мы ищем позиции, которые больше или равны 15 и меньше или равны 22. Чтобы найти 15, первый цикл `while` выполняет сканирование вправо по одной позиции за раз. Цикл находит число 9, затем 10, 11, 12, 13, 14 и, наконец, 15. Дальше начинает работу второй цикл `while`, который выполняет такой же объем сканирования и работает, пока не дойдет до 23.

Каждый цикл `while` реализует так называемый *линейный поиск* — метод поиска в коллекции по одному значению за раз. Перебор всех значений — это слишком большая работа! При этом много пар `i` и `j` выполняют аналогичный объем работы. Например, попробуйте отследить, что произойдет, когда `i` будет равно 0, а `j` равно 8 или `i` равно 1, а `j` равно 11.

Как улучшить работу? Как избежать сканирования огромного фрагмента списка в поисках подходящих левого и правого индексов?

Представьте, что я даю вам книгу с тысячей отсортированных целых чисел, по одному числу на странице. Я прошу вас найти первое целое число, которое больше или равно 300. Будете ли вы просматривать числа один за другим? Может быть, попробуете найти 1, затем 3, затем 4, затем 7? Тоже довольно долго — не годится! Было бы намного быстрее, если бы вы просто открыли книгу на середине. Может быть, вы найдете там номер 450. Поскольку 450 больше 300, теперь вы знаете, что это число находится в первой половине книги и не может оказаться во второй половине, потому что там числа больше 450. Вы сократили объем работы вдвое, проверив всего одно число! Теперь можете повторить этот процесс с первой половиной книги, выбрав точку между началом и серединой книги. И обнаружите там число 200. Теперь вы знаете, что 300 находится на одной из следующих страниц, где-то во второй четверти книги.

Можете повторять этот процесс, пока не найдете число 300, и займет он совсем немного времени. Метод многократного деления задачи пополам известен как *бинарный поиск*. Работает он потрясающе быстро. Линейный поиск нервно курит в сторонке. В Python есть функция двоичного поиска, ее-то нам и не хватает в задаче «Коровий бейсбол». Однако эта функция находится внутри некоей вещи под названием *«модуль»*, поэтому сначала поговорим о модулях.

Модули Python

Модуль — это автономный набор кода Python. Обычно в модулях содержатся функции, которые можно вызывать.

В Python встроено множество модулей, которые используются для расширения функциональности программ. Существуют модули для работы со случайными числами, датами и временем, статистикой, электронными письмами, веб-страницами, аудиофайлами и многим другим. Чтобы рассказать про них подробно, потребуется целая отдельная книга! Существуют даже модули, которые можно скачать дополнительно, если нужного решения в комплекте с Python не нашлось.

В этом разделе нас интересует один модуль — `random`. На его примере мы узнаем все необходимое о модулях. После этого в следующем разделе воспользуемся модулем двоичного поиска.

Вы когда-нибудь задумывались, как люди создают компьютерные игры, в которых происходят случайные события? Это может быть игра, в которой вы тянете карты, бросаете кости или в которой внезапно появляются враги. В основе этих механик лежит использование случайных чисел. В Python генерация случайных чисел реализована в модуле `random`.

Чтобы задействовать то, что находится в модуле, мы должны его *импортировать*. Первый способ сделать это — импортировать весь модуль с помощью ключевого слова `import`, например:

```
>>> import random
```

И что в нем есть? Чтобы узнать, можете использовать команду `dir(random)`:

```
>>> dir(random)
[stuff to ignore
'betavariate', 'choice', 'choices', 'expovariate',
'gammavariate', 'gauss', 'getrandbits', 'getstate',
'lognormvariate', 'normalvariate', 'paretovariate',
'randint', 'random', 'randrange', 'sample', 'seed',
'setstate', 'shuffle', 'triangular', 'uniform',
'vonmisesvariate', 'weibullvariate']
```

Одна из функций модуля `random` — `randint`. Ей мы передаем нижний и верхний пределы диапазона, а Python выдает в ответ случайное целое число из него (включая обе конечные точки).

Но просто вызвать ее как обычную функцию нельзя, а если мы попробуем это сделать, получим ошибку:

```
>>> randint(2, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'randint' is not defined
```

Нужно сообщить Python, что функция `randint` находится в модуле `randim`. Для этого мы добавляем к вызову `randint` имя модуля и точку, например:

```
>>> random.randint(2, 10)
7
>>> random.randint(2, 10)
10
>>> random.randint(2, 10)
6
```

Для получения справки по функции `randint` можно ввести команду `help(random.randint)`:

```
>>> help(random.randint)
Help on method randint in module random:
randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Еще одна полезная функция в модуле `random` — `choice`. Ей передается некоторая последовательность, и функция возвращает одно из значений из последовательности случайным образом:

```
>>> random.choice(['win', 'lose'])
'lose'
>>> random.choice(['win', 'lose'])
'lose'
>>> random.choice(['win', 'lose'])
'win'
```

Если мы часто используем некоторую функцию из модуля, может быстро надоесть каждый раз вводить его имя и точку. Есть еще один способ импортировать эти функции, который позволяет вызывать их, как любые другие функции не из модулей. Далее показано, как импортировать только функцию `randint`:

```
>>> from random import randint
```

Теперь можно вызывать функцию `randint`, не приписывая `random.`:

```
>>> randint(2, 10)
10
```

Если нам понадобятся две функции, можем импортировать обе:

```
>>> from random import randint, choice
```

В книге мы этого делать не будем, но отмечу, что вы можете создавать собственные модули и помещать в них любые нужные функции. Например, если вы напишете несколько функций Python, связанных с некоторой игрой, их можно поместить в файл `game_functions.py`. Затем можно импортировать этот модуль с помощью команды `import game_functions`, после чего обращаться к функциям из модуля.

Программы Python, которые мы писали в этой книге, не предназначены для импорта. Причина в том, что все они читают входные данные сразу, как только начинают работать. Модуль этого делать не должен. Он должен сидеть и ждать, пока его функции кто-нибудь вызовет. Модуль `random` — это пример хорошо настроенного модуля, которые начинает выдавать случайные числа, только когда мы его попросим.

Модуль bisect

Теперь можно побаловаться с бинарным поиском. В листинге 9.6 было два цикла `while`. Они медленные, поэтому мы хотим от них избавиться. Для этого заменим каждый из них вызовом функции бинарного поиска: `bisect_left` вместе первого цикла `while` и `bisect_right` вместо второго.

Обе эти функции находятся в модуле `bisect`. Импортируем их:

```
>>> from bisect import bisect_left, bisect_right
```

Сначала рассмотрим функцию `bisect_left`. Во время вызова мы должны передать ей список, отсортированный от наименьшего к наибольшему, и значение `x`. Функция возвращает индекс самого левого значения в списке, которое больше или равно `x`.

Если это значение есть в списке, мы получаем индекс его самого левого вхождения:

```
>>> bisect_left([10, 50, 80, 80, 100], 10)
0
>>> bisect_left([10, 50, 80, 80, 100], 80)
2
```

Если значение в списке отсутствует, получаем индекс первого значения, которое больше него:

```
>>> bisect_left([10, 50, 80, 80, 100], 15)
1
>>> bisect_left([10, 50, 80, 80, 100], 81)
4
```

Если значение x превышает любое значение в списке, в ответе мы получаем длину списка:

```
>>> bisect_left([10, 50, 80, 80, 100], 986)
5
```

Воспользуемся функцией bisect_left на списке из семи позиций, рассмотренном ранее. Найдем индекс крайней левой позиции, которая больше или равна 16:

```
>>> positions = [1, 6, 11, 14, 16, 18, 23]
>>> bisect_left(positions, 16)
4
```

Идеально: именно такой функцией нужно заменить первый цикл while в листинге 9.6.

Чтобы заменить второй while, лучше подойдет функция bisect_right, а не bisect_left. Вызывается bisect_right так же, как bisect_left: ей передаются отсортированный список и значение x. Вместо того чтобы возвращать индекс самого левого значения в списке, которое *больше или равно* x, функция возвращает индекс самого левого значения, *большего* чем x.

Сравним работу функций bisect_left и bisect_right. Для значения из списка bisect_right возвращает больший индекс по сравнению с bisect_left:

```
>>> bisect_left([10, 50, 80, 80, 100], 10)
0
>>> bisect_right([10, 50, 80, 80, 100], 10)
1
>>> bisect_left([10, 50, 80, 80, 100], 80)
2
>>> bisect_right([10, 50, 80, 80, 100], 80)
4
```

Для значения, которого нет в списке, bisect_left и bisect_right возвращают одинаковые индексы:

```
>>> bisect_left([10, 50, 80, 80, 100], 15)
1
>>> bisect_right([10, 50, 80, 80, 100], 15)
1
>>> bisect_left([10, 50, 80, 80, 100], 81)
4
>>> bisect_right([10, 50, 80, 80, 100], 81)
4
>>> bisect_left([10, 50, 80, 80, 100], 986)
5
>>> bisect_right([10, 50, 80, 80, 100], 986)
5
```

Проверим `bisect_right` на списке из семи позиций, приведенном ранее. Найдем индекс крайней левой позиции, большей чем 21:

```
>>> positions = [1, 6, 11, 14, 16, 18, 23]
>>> bisect_right(positions, 21)
6
```

Вот и все: этой функцией можно заменить второй цикл `while` из листинга 9.6.

С помощью этих крошечных примеров трудно оценить потрясающую скорость двоичного поиска. Рассмотрим более реальный пример и попробуем найти самое правое значение в списке из 1 000 000 элементов. Не моргайте, а то пропустите весь фокус:

```
>>> lst = list(range(1, 1000001))
>>> for i in range(1000000):
... where = bisect_left(lst, 1000000)
...
```

На моем компьютере это занимает около секунды. Вам может быть интересно, что произойдет, если заменить бинарный поиск вызовом метода `index`. Попробуйте — выполнения кода придется ждать часами. Все потому, что метод `index`, как и оператор `in`, выполняет линейный поиск по списку (в главе 8 об этом говорилось подробнее). У нас нет гарантии, что список отсортирован, поэтому молниеносный двоичный поиск выполнить не получится. Значения в списке перебираются одно за другим, и каждое из них сравнивается со значением, которое мы ищем. Если у вас есть отсортированный список и вы хотите найти в нем значения, бинарный поиск будет непревзойденным.

Решение задачи

Мы готовы решить «Коровий бейсбол» с помощью бинарного поиска. Код приведен в листинге 9.7.

Листинг 9.7. Использование бинарного поиска

```
❶ from bisect import bisect_left, bisect_right

  input_file = open('baseball.in', 'r')
  output_file = open('baseball.out', 'w')

  n = int(input_file.readline())

  positions = []

  for i in range(n):
      positions.append(int(input_file.readline()))
```

```
positions.sort()

total = 0

for i in range(n):
    for j in range(i + 1, n):
        first_two_diff = positions[j] - positions[i]
        low = positions[j] + first_two_diff
        high = positions[j] + first_two_diff * 2
    ❷ left = bisect_left(positions, low)
    ❸ right = bisect_right(positions, high)
        total = total + right - left

output_file.write(str(total) + '\n')

input_file.close()
output_file.close()
```

Сперва импортируем функции `bisect_left` и `bisect_right` из модуля `bisect`, чтобы их можно было вызвать ❶. Единственное отличие этого кода от листинга 9.6 заключается в том, что теперь мы используем `bisect_left` ❷ и `bisect_right` ❸ вместо цикла `while`.

Если вы отправите код на сайт, все тесты должны быть выполнены вовремя.

Метод, который мы задействовали в этом разделе, часто применяется при решении сложных проблем. Мы начали с решения методом полного поиска, которое, хоть и оказалось правильным, было слишком медленным и не соответствует временным ограничениям сайта. Затем добавили в программу улучшения, перейдя от полного поиска к более совершенному подходу.

ПРОВЕРИМ ЗНАНИЯ

Предположим, мы взяли код листинга 9.7 и использовали `bisect_left` вместо `bisect_right`. То есть строку:

```
right = bisect_right(positions, high)
```

заменили на:

```
right = bisect_left(positions, high)
```

Выдает ли программа по-прежнему правильные ответы?

А. Программа всегда дает правильный ответ, как и раньше.

Б. Иногда программа дает правильный ответ, но это зависит от тестового примера.

В. Программа никогда не дает правильного ответа.

Ответ: **Б.** Есть тестовые примеры, для которых измененный код действительно дает правильный ответ. Вот один из них:

```
3
2
4
9
```

Правильный ответ — `0`, его программа и выдаст.

Но есть и другие тестовые примеры, для которых измененный код дает неправильный ответ. Вот один из них:

```
3
2
4
8
```

Правильный ответ — `1`, но программа выдает `0`. Когда `i = 0` и `j = 1`, программа должна присвоить переменным значения `left = 2` и `right = 3`. Но функция `bisect_left` присваивает переменной `right` значение `2`, потому что позиция в индексе `2` — это крайняя левая позиция, которая больше или равна `8`.

Учитывая этот пример, вы можете удивиться, узнав, что все же получается использовать `bisect_left` вместо `bisect_right`. Для этого нужно изменить результат, который ищем вызовом `bisect_left`. Если вам интересно, попробуйте!

Резюме

В этой главе вы изучили алгоритмы полного поиска, которые перебирают все варианты решений в поисках необходимого нам ответа. Для того чтобы определить, какого из спасателей нужно уволить, мы пробовали уволить их всех по очереди и оценивали результат. Чтобы определить минимальную стоимость переделки холмов для устройства горнолыжной базы, мы пробуем все допустимые диапазоны высот и выбираем лучший. Чтобы найти количество подходящих троек коров, проверяем каждую тройку и добавляем те, которые соответствуют требованиям.

Иногда алгоритмы полного поиска достаточно эффективны. Задачи «Спасатели» и «Лыжная база» мы решили с помощью простого кода полного поиска. Но затем нам потребовалось сделать алгоритм полного поиска более эффективным. В решении задачи «Коровий бейсбол» мы заменили цикл `while` гораздо более эффективным алгоритмом бинарного двоичного поиска.

Как программисты изучают вопросы эффективности? Как узнать, будет ли алгоритм достаточно эффективным? Можно ли избежать реализации слишком медленных алгоритмов? Глава 10 ответит на эти вопросы.

Упражнения

Далее приведены несколько упражнений, которые вы можете выполнить. Каждую задачу можно решить с применением алгоритма полного поиска. Если ваше решение недостаточно эффективно, подумайте, как улучшить его, но при этом дать правильный ответ.

Обращайте внимание, на каком сайте находится задача. Некоторые из них опубликованы на DMOJ, другие — на USACO.

1. USACO, задача Shell Game, 2019 January Bronze Contest.
2. USACO, задача Diamond Collector, 2016 US Open Bronze Contest.
3. DMOJ, задача Patkice с кодом `coci20c1p1`.
4. DMOJ, задача Old Fishin' Hole с кодом `ccc09j2`.
5. DMOJ, задача Spindle с кодом `ecoo16r1p2`.
6. DMOJ, задача SafeBreaker с кодом `cco96p2`.
7. USACO, задача Where Am I, 2019 December Bronze Contest.
8. USACO, задача Angry Cows, 2016 January Bronze Contest.
9. USACO, задача Counting Haybales, 2016 December Silver Contest.
10. DMOJ, задача Firefly с кодом `crci06p3`.

Примечания

Задача «Спасатели» взята с USACO 2018 January Bronze Contest. Задача «Лыжная база» — с USACO 2014 January Bronze Contest. «Коровий бейсбол» взята с USACO 2013 December Bronze Contest. Помимо алгоритмов полного поиска, существуют и другие типы алгоритмов, такие как *жадные алгоритмы* и *алгоритмы динамического программирования*. Если задачу не удается решить полным поиском, то стоит попробовать применить другой алгоритм.

Если вам интересно узнать больше об этих и других алгоритмах, которые используются в Python, я рекомендую книгу Магнуса Ли Хетланда *Python Algorithms*, 2-е издание (Apress, 2014).

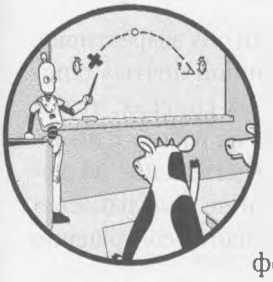
Я тоже писал книгу о разработке алгоритмов: *Algorithmic Thinking: A Problem-Based Introduction* (No Starch Press, 2021). Эта книга построена так же, как и данная,

поэтому ее стиль и темп будут вам привычны. Но в ней применяется не Python, а язык программирования C, поэтому, чтобы извлечь из нее максимум пользы, нужно заранее изучить его.

В этой главе для выполнения бинарного поиска мы вызывали готовые функции Python. При желании можно написать свой код бинарного поиска и не брать готовые функции. Идея деления списка пополам до тех пор, пока не будет найдено нужное значение, интуитивно понятна, но код ее реализации на удивление сложен. А еще удивительнее вам покажется широкий спектр проблем, которые можно решить с использованием бинарного поиска. В упомянутой мной книге есть целая глава о бинарном поиске и его возможностях.

10

«О большое» и эффективность программ



В первых семи главах книги мы занимались в первую очередь написанием правильных программ, то есть таких, которые для допустимых входных данных выдают желаемый результат. Однако, помимо правильного кода, часто бывает нужен эффективный код, который работал бы быстро даже при огромных объемах входных данных. Возможно, при проработке первых семи глав вы пару раз получали ошибки превышения лимита времени, но первый подход к снаряду написания эффективных программ состоялся в главе 8, когда мы решали задачу «Адреса электронной почты». Там вы увидели, что иногда нужно сделать программы более эффективными, чтобы они могли завершиться в установленный срок.

В этой главе мы сперва побеседуем о том, что программисты в общем говорят об эффективности программ. Затем изучим две задачи, для которых придется писать не только правильный, но и эффективный код: это будет определение наиболее желаемого участка шарфа и раскрашивание ленты.

В обеих задачах увидим, что алгоритмы, о которых мы подумали в первую очередь, окажутся недостаточно эффективными. Но мы будем пробовать варианты решения одной задачи, пока не создадим более быстрый алгоритм. Именно так обычно и работает программист: сперва нужно придумать правильный алгоритм, а затем, если нужно, сделать его быстрее.

Проблема со сроками

В каждой задаче соревновательного программирования, которую мы решили и еще решим в этой книге, имеется ограничение по времени, в течение которого программа должна получить ответ (хотя я начал вводить ограничения по времени в описания задач лишь в главе 8, когда мы столкнулись с теми, в которых эффективность крайне важна). Если наша программа превышает ограничение по времени, то тестировщик сайта завершает ее с ошибкой превышения лимита времени. Ограничение по времени предназначено для того, чтобы слишком медленные решения не прошли тесты. Например, мы могли придумать решение методом полного поиска, но автор задачи нашел более быстрое решение. Оно может быть вариантом полного поиска, как получилось в задаче «Коровий бейсбол» в главе 9, или основываться на совершенно другом подходе. Тем не менее ограничение по времени может быть таким, что наше решение с применением полного поиска не уложится в него. Таким образом, программы должны быть не только правильными, но и быстрыми.

Мы можем запустить программу, чтобы выяснить, достаточно ли она эффективна. Например, в главе 8, пытаясь решить задачу «Адреса электронной почты» с помощью списка, мы запускали код на все более и более крупных списках, чтобы получить представление о количестве времени, затрачиваемом на работу с ними. Такое ручное тестирование позволяет получить некоторое представление об эффективности программ. Если программа слишком медленная и не укладывается в лимит времени, нужно оптимизировать имеющийся код или найти совершенно новый подход.

Время, затрачиваемое программой на выполнение, зависит от компьютера, на котором она запускается. Мы не знаем, какой компьютер использует тестер сайта, но даже запуск программы на собственном компьютере позволяет понять общую картину, так как на сайте, вероятно, машина не хуже. Допустим, мы запускаем программу на ноутбуке и выполнение на небольшом тестовом примере занимает 30 секунд. Если ограничение по времени задачи составляет 3 секунды, можно быть уверенными, что программа работает крайне медленно.

Но ориентация исключительно на временные рамки связывает нам руки. Вспомните первое решение задачи «Коровий бейсбол» в главе 9. Не нужно было даже запускать тот код, чтобы определить, насколько медленным он будет. Дело в том, что мы могли оценить программу с точки зрения объема работы, которую она выполнила бы при запуске. Например, смогли оценить, что для n коров программа обрабатывает n^3 троек коров. Обратите внимание на то, что мы говорили не о количестве секунд, которое заняло бы выполнение, а о том, сколько работы код выполняет по отношению к известному объему входных данных n.

Такой анализ дает значительные преимущества по сравнению с запуском программы и подсчетом времени выполнения.

- **Время выполнения зависит от компьютера.** Время выполнения говорит лишь о том, как долго работала программа на конкретном компьютере. Это очень специфическая информация, которая не позволяет понять, чего стоит ожидать при запуске программы на других компьютерах. Работая с книгой, вы могли заметить, что время, затрачиваемое программой, меняется от запуска к запуску, даже если они выполнялись на одном компьютере. Вы можете запустить программу на тестовом примере, и она будет выполнена за 3 секунды, а затем на том же тестовом примере выполнение может занять 2,5 или 3,5 секунды. Причина этой разницы заключается в том, что операционная система сама управляет своими вычислительными ресурсами, используя их для решения своих задач и выполнения других запущенных программ. Решения, которые принимает операционная система, влияют на время выполнения программы.
- **Время выполнения зависит от тестового примера.** Время работы программы на тестовом примере говорит только о том, сколько времени уйдет на конкретный пример. Предположим, что программа на небольшом тестовом примере выполняется за 3 секунды. Это вроде бы быстро, но таковы небольшие тесты: любое рабочее решение выполняется на них быстро. Если нужно будет найти количество уникальных адресов электронной почты среди десяти адресов или количество троек коров среди десяти коров, это можно сделать быстро с помощью первого пришедшего в голову алгоритма. Проверять же работу надо на больших примерах. Именно на них алгоритмическая сложность окупается. Сколько времени займет выполнение на большом или даже огромном тестовом примере? Мы не знаем. Придется запустить программу на них. И даже если бы мы это сделали, всегда найдутся определенные виды тестовых примеров, производительность которых ниже, и нам может показаться что программа работает быстрее, чем на самом деле.
- **Программе требуется реализация.** Нельзя рассчитать время выполнения ненаписанной программы. Предположим, мы размышляем о задаче и думаем, как ее решить. Быстрым ли получится придуманное решение? Можно попробовать реализовать его и тем самым понять это, но хорошо бы знать заранее, верна ли идея в принципе. Вы бы не стали реализовывать программу, неправильную уже на уровне идеи. Поэтому хорошо было бы знать с самого начала, что программа будет работать слишком медленно.
- **Скорость определяется не только временем выполнения.** Если выяснится, что программа слишком медленная, наша следующая задача — разработать более быстрый вариант. Но одного лишь времени работы программы недостаточно, чтобы понять, почему она работает медленно, поверьте. Кроме того,

если удастся придумать возможное улучшение программы, потребуется реализовать его, чтобы посмотреть, поможет ли оно.

- **Время выполнения сложно использовать для обсуждения.** По многим из перечисленных причин трудно считать время выполнения критерием эффективности алгоритмов при их обсуждении. Время выполнения слишком специфично: оно зависит от компьютера, операционной системы, тестового примера, языка программирования и конкретной реализации. Для нормального обсуждения нужно предоставить всю эту информацию тем, с кем мы обсуждаем код.

Не волнуйтесь: у программистов есть система обозначений, не имеющая описанных недостатков. Она не зависит от компьютера, тестового примера и конкретной реализации. Она позволяет понять, почему программа работает медленно. Ее легко сообщать другим. Она называется нотацией *«О большое»*, и мы о ней поговорим.

«О большое»

«О большое» — это нотация, которую информатики используют для краткого описания эффективности алгоритмов. Ключевым понятием здесь является *класс эффективности*, характеризующий, насколько быстр алгоритм или сколько работы он выполняет. Чем быстрее алгоритм, тем у него меньше работы, а чем он медленнее, тем больше работает. Каждый алгоритм относится к определенному классу эффективности. Последний говорит о том, какую работу алгоритм выполняет по отношению к количеству входных данных, которые предстоит обработать. Чтобы понять эту нотацию, сперва нужно изучить классы эффективности. Здесь мы рассмотрим семь наиболее распространенных. Среди них будут те, которые выполняют наименьший объем работы и которым, как нам хотелось бы, соответствовали наши алгоритмы. А будут и такие, которые выполняют значительно больше работы и, скорее всего, будут часто выходить за временные рамки.

Постоянное время

Наиболее желанны для нас алгоритмы, объем работы которых не растет с увеличением количества входных данных. Вне зависимости от экземпляра задачи такой алгоритм выполняется за примерно равное количество шагов. Эти алгоритмы называются алгоритмами *с постоянным временем*.

Сложно поверить, что так бывает, правда? Алгоритм, который выполняет примерно одинаковый объем работы несмотря ни на что? Действительно, решение задачи с помощью такого алгоритма — редкость. Но если вам удастся его реализовать, радуйтесь: лучше и быть не может.

Нам уже удалось решить в этой книге несколько задач, используя алгоритмы с постоянным временем. Например, в задаче «Телемаркетологи» в главе 2, где нужно

было определить, принадлежит ли определенный номер телефона маркетологу. Вспомним решение из листинга 2.2:

```
num1 = int(input())
num2 = int(input())
num3 = int(input())
num4 = int(input())

if ((num1 == 8 or num1 == 9) and
        (num4 == 8 or num4 == 9) and
        (num2 == num3)):
    print('ignore')
else:
    print('answer')
```

Это решение выполняет одинаковый объем работы независимо от того, из каких четырех цифр состоит номер телефона. Код сперва читает входные данные, затем обрабатывает числа `num1`, `num2`, `num3` и `num4`. Если номер телефона принадлежит маркетологу, мы что-то выводим, а если не принадлежит — выводим что-то другое. В задаче нет данных, которые могут вызвать лишние трудности.

Ранее в главе 2 мы решили задачу «Команда-победитель». У нее постоянное время выполнения? Ну да! Вот решение из листинга 2.1:

```
apple_three = int(input())
apple_two = int(input())
apple_one = int(input())

banana_three = int(input())
banana_two = int(input())
banana_one = int(input())

apple_total = apple_three * 3 + apple_two * 2 + apple_one
banana_total = banana_three * 3 + banana_two * 2 + banana_one

if apple_total > banana_total:
    print('A')
elif banana_total > apple_total:
    print('B')
else:
    print('T')
```

Мы читаем входные данные, вычисляем общее количество баллов одной команды, общее количество баллов другой команды, сравниваем эти суммы и выводим сообщение. Неважно, сколько очков у «Яблок» или «Бананов», — наша программа всегда выполняет одинаковый объем работы.

А что, если бы «Яблоки» забили миллион трехочковых бросков? Ведь компьютеру для работы с огромными числами требуется больше времени, чем когда числа маленькие, такие как 10 или 50? Это правда, но в данном случае не о чем беспокоиться. В описании задачи сказано, что каждая команда набирает не более 100 очков

за каждый тип бросков. Поэтому мы работаем с небольшими числами, и будет справедливо сказать, что компьютер может считывать эти числа или оперировать ими за постоянное число шагов. В целом значения в несколько миллиардов тоже можно считать маленькими.

В нотации «О большое» алгоритм с постоянным временем обозначается $O(1)$. Цифра 1 здесь не означает, что алгоритм с постоянным временем выполняется за один шаг. Если программа выполняет фиксированное количество шагов, например 10 или даже 10 000, это все равно считается постоянным временем. Но это не обозначается $O(10)$ или $O(10\,000)$ — все алгоритмы с постоянным временем обозначаются $O(1)$.

Линейное время

Большинство алгоритмов не выполняются за постоянное время. Объем работы программы обычно зависит от размера входных данных. Например, для обработки 1000 значений требуется больше работы, чем для обработки десяти. У этих алгоритмов существует взаимосвязь между объемом входных данных и количеством работы, которую выполняет алгоритм.

Алгоритм *с линейным временем* — это алгоритм с линейной зависимостью между объемом входных данных и выполненной работой. Предположим, вначале мы запускаем такой алгоритм на 50 значениях, а затем на 100. Во втором случае алгоритм выполнит примерно вдвое больше работы.

В качестве примера рассмотрим задачу «Три чашки» из главы 3. Мы решили ее в листинге 3.1, и здесь я воспроизвел решение:

```
swaps = input()

ball_location = 1

❶ for swap_type in swaps:
    if swap_type == 'A' and ball_location == 1:
        ball_location = 2
    elif swap_type == 'A' and ball_location == 2:
        ball_location = 1
    elif swap_type == 'B' and ball_location == 2:
        ball_location = 3
    elif swap_type == 'B' and ball_location == 3:
        ball_location = 2
    elif swap_type == 'C' and ball_location == 1:
        ball_location = 3
    elif swap_type == 'C' and ball_location == 3:
        ball_location = 1

print(ball_location)
```

В программе есть цикл `for` ❶, и объем работы, которую он выполняет, линейно зависит от объема ввода. Если нужно обработать пять перемен позиции, цикл повторяется пять раз. Если требуется обработать десять перемен позиции, он повторяется десять раз. Каждая итерация цикла выполняет постоянное число сравнений и влияет на переменную `ball_location`. Следовательно, объем работы, выполняемой этим алгоритмом, прямо пропорционален количеству перемен позиции.

Числом n обычно обозначают количество входных данных, переданных задаче. Здесь n — количество перемен позиции. Если их пять, то n равно пяти, а если десять, то n равно десяти.

Если число перемен позиции — n, то программа работает примерно n времени. Это связано с тем, что цикл `for` выполняет n итераций, каждая из которых делает постоянное количество шагов. Нам неважно, сколько шагов код выполняет на каждой итерации, если это постоянное число. Независимо от того, выполняет ли алгоритм n шагов, $10n$ шагов или $10\,000n$ шагов, это алгоритм с линейным временем. В нотации «О большое» мы говорим, что алгоритм имеет сложность $O(n)$.

При использовании нотации «О большое» коэффициенты перед n не учитываются. Например, алгоритм, который выполняет $10n$ шагов, записывается $O(n)$, а не $O(10n)$. Это помогает сосредоточиться на том факте, что он является линейным по времени, а подробности в данном случае неважны.

А если алгоритм выполняется за $2n + 8$ шагов — что это за алгоритм? Все равно линейный! Причина в том, что линейный член $(2n)$ будет доминировать над постоянным членом (8), как только n станет достаточно большим. Например, если n равно 5000, то $8n$ равно 40 000. Число 8 настолько мало по сравнению с 40 000, что на него можно не обращать внимания. В нотации «О большое» мы игнорируем все члены, кроме доминирующих.

Многие операции Python выполняются за постоянное время. Например, добавление элемента в список, добавление элемента в словарь, индексация множества или словаря выполняются за постоянное время.

А некоторые другие операции Python выполняются за линейное время. Важно правильно относить операции к выполняющимся за линейное, а не за постоянное время. Например, использование функции `input` для чтения длинной строки требует линейного времени, потому что Python прочитывает каждый символ в строке. Любая операция, которая проверяет каждый символ строки или значения в списке, также выполняется за линейное время.

Если алгоритм считывает n значений и обрабатывает каждое из них за постоянное число шагов, то это алгоритм с линейным временем.

Не нужно далеко ходить за еще одним примером алгоритма линейного времени — решение задачи «Занятые места» в главе 3 именно такое. Приведу решение из листинга 3.3:

```
n = int(input())
yesterday = input()
today = input()

occupied = 0

for i in range(len(yesterday)):
    if yesterday[i] == 'C' and today[i] == 'C':
        occupied = occupied + 1

print(occupied)
```

Пусть n — это количество парковочных мест. Схема такая же, как в задаче «Три чашки»: мы читаем входные данные, а затем выполняем постоянное количество шагов для каждого парковочного места.

ПРОВЕРИМ ЗНАНИЯ

В листинге 1.1 мы выполнили задачу подсчета количества слов. Напомню ее решение:

```
line = input()
total_words = line.count(' ') + 1
print(total_words)
```

Какова эффективность алгоритма?

А. $O(1)$.

Б. $O(n)$.

Ответ: **Б**. Так и хочется сказать, что сложность алгоритма $O(1)$, ведь тут нет циклов и кажется, будто алгоритм выполняет всего три действия: читает входные данные, вызывает метод `count` для подсчета количества слов и выводит ответ.

Но реальная сложность здесь $O(n)$, где n — число букв во входных данных. Функция `input` считывает входные данные за линейное время, так как чтение выполняется посимвольно. Метод `count` также реализуется за линейное время, поскольку обрабатывает каждый символ в строке, отыскивая совпадения. Поэтому время выполнения алгоритма линейно как в момент чтения ввода, так и в момент обработки.

ПРОВЕРИМ ЗНАНИЯ

В листинге 1.2 мы вычислили объем конуса. Решение задачи:

```
PI = 3.141592653589793

radius = int(input())
height = int(input())

volume = (PI * radius ** 2 * height) / 3

print(volume)
```

Какова эффективность алгоритма? (Помните, что радиус и высота не превышают 100.)

А. $O(1)$.

Б. $O(n)$.

Ответ: **А.** Здесь обрабатываются маленькие числа, так что чтение входных данных занимает постоянное время. Вычисление объема тоже постоянное, так как выполняется лишь пара операций. Получается, задача состоит из нескольких постоянных по времени шагов.

ПРОВЕРИМ ЗНАНИЯ

В листинге 3.4 мы решили задачу «Тарифный план». Вот ее решение:

```
monthly_mb = int(input())
n = int(input())

excess = 0

for i in range(n):
    used = int(input())
    excess = excess + monthly_mb - used

print(excess + monthly_mb)
```

Какова эффективность алгоритма?

А. $O(1)$.

Б. $O(n)$.

Ответ: **Б**. Шаблон для этого алгоритма аналогичен шаблону решения «Три чашки» или «Занятые места», за исключением того, что здесь чтение ввода и его обработка чередуются. Пусть n — количество значений в мегабайтах в месяц. Программа выполняет постоянное количество шагов для каждого из этих n значений. Следовательно, это алгоритм сложностью $O(n)$.

Квадратичное время

Мы изучили алгоритмы с постоянным временем (при увеличении объема входных данных их работа остается постоянной) и алгоритмы с линейным временем (время их выполнения растет линейно по мере увеличения количества входных данных). Подобно алгоритму с линейным временем, алгоритм с *квадратичным временем* тоже занимает все больше времени по мере увеличения количества входных данных: 1000 значений обрабатываются дольше, чем десять.

Принимая во внимание тот факт, что на относительно больших объемах входных данных можно обойтись линейным алгоритмом, с квадратичными алгоритмами мы ограничены еще сильнее. Посмотрим почему.

Типичный алгоритм

Типичный алгоритм линейного времени выглядит так:

```
for i in range(n):
    <обрабатывать ввод i за постоянное количество шагов>
```

А вот типичный алгоритм квадратичного времени:

```
for i in range(n):
    for j in range(n):
        <обрабатывать ввод i и j за постоянное количество шагов>
```

Если во входных данных n значений, сколько значений обрабатывает каждый алгоритм? Алгоритм линейного времени обрабатывает n значений, по одному на каждой итерации цикла `for`. В свою очередь, алгоритм квадратичного времени обрабатывает n значений на *каждой итерации* внешнего цикла `for`.

На первой итерации внешнего цикла `for` обрабатываются n значений (по одному на каждой итерации внутреннего цикла `for`), на второй итерации внешнего цикла `for` обрабатываются еще n значений (по одному на каждой итерации внутреннего цикла `for`) и т. д. Поскольку внешний цикл `for` повторяется n раз, общее

количество обрабатываемых значений равно nn, или n^2. Два вложенных цикла, каждый из которых зависит от n, дают квадратичное время работы алгоритма. В нотации «О большое» говорят, что алгоритм с квадратичным временем имеет сложность $O(n^2)$.

Сравним объем работы, выполняемый алгоритмами линейного и квадратичного времени. Предположим, что мы обрабатываем входные данные из 1000 значений, то есть $n = 1000$. Алгоритм с линейным временем будет выполнен за 1000 шагов. Алгоритм с квадратичным временем, который занимает n^2 шагов, потребует $1000^2 = 1\,000\,000$ шагов. Миллион намного больше тысячи. Но кого это волнует, ведь компьютеры и не такое умеют, не правда ли? Ну да, на входных данных из 1000 значений вполне можно использовать алгоритм квадратичного времени. Ранее в книге я приводил консервативное правило, согласно которому мы можем выполнять около 5 миллионов шагов в секунду. Миллион шагов выполняется во вполне разумное время.

Но любой оптимизм в отношении алгоритма квадратичного времени недолговечен. Посмотрите, что произойдет, если мы увеличим количество входных значений с 1000 до 10 000. Алгоритм линейного времени будет выполнен всего за 10 000 шагов. Квадратичному алгоритму потребуются $10\,000^2 = 100\,000\,000$ шагов. Хм... с алгоритмом квадратичного времени получается уже не так быстро. Алгоритм линейного времени по-прежнему займет миллисекунды, а алгоритм квадратичного — несколько секунд. Лимит времени точно будет превышен.

ПРОВЕРИМ ЗНАНИЯ

Какова эффективность приведенного далее алгоритма:

```
for i in range(10):
    for j in range(n):
        <обрабатывать ввод i и j за постоянное количество шагов>
```

А. $O(1)$.

Б. $O(n)$.

В. $O(n^2)$.

Ответ: **Б.** У нас два вложенных цикла, поэтому сперва кажется, что алгоритм квадратичный, но внешний цикл выполняется всего десять раз и не зависит от n. Тогда общее число шагов равно $10n$, и никакого n^2 здесь нет. $10n$ — это просто n. Следовательно, это алгоритм сложностью $O(n)$.

ПРОВЕРИМ ЗНАНИЯ

Какова эффективность приведенного далее алгоритма?

```
for i in range(n):
    <обрабатывать ввод i за постоянное количество шагов>
for j in range(n):
    <обрабатывать ввод j за постоянное количество шагов>
```

А. $O(1)$.

Б. $O(n)$.

В. $O(n^2)$.

Ответ: **Б**. У нас два цикла, зависящих от n. Все выглядит квадратично, разве нет?

Нет. Эти циклы выполняются один за другим и не вложены друг в друга. Оба выполняются за n шагов, что дает $2n$ шагов. Алгоритм линейный.

Альтернативная форма

Когда вы видите два вложенных цикла, каждый из которых зависит от n, скорее всего, перед вами алгоритм квадратичного времени. Но иногда этот алгоритм возникает даже при отсутствии вложенных циклов. Такой пример есть в первом решении задачи «Адреса электронной почты» из листинга 8.2. Я воспроизвел это решение далее:

```
# функция очистки опущена

for dataset in range(10):
    n = int(input())
    addresses = []
    for i in range(n):
        address = input()
      ❶ address = clean(address)
      ❷ if not address in addresses:
            addresses.append(address)

    print(len(addresses))
```

Пусть n — максимальное количество адресов электронной почты, которое мы можем найти в десяти тестовых примерах. Внешний цикл `for` повторяется десять раз,

внутренний цикл `for` — не более n раз. Таким образом, мы обрабатываем не более $10n$ адресов, что дает линейное время по n.

Очистка адреса электронной почты ❶ выполняется за постоянное число шагов, так что об этом думать не нужно. Но это все же не алгоритм линейного времени, потому что каждая итерация внутреннего цикла `for` выполняется не за постоянное количество шагов. В частности, проверка того, находится ли адрес электронной почты в списке ❷, требует работы, пропорциональной количеству адресов электронной почты, уже находящихся в списке, так как Python выполняет поиск по списку, а это операция с линейным временем! Итак, мы обрабатываем $10n$ адресов электронной почты, каждый из которых требует n шагов, то есть сложность $10n^2$. Значит, программа выполняется за квадратичное время. Именно из-за квадратичного времени мы превысили время выполнения и перешли к использованию множеств, а не списков.

Кубическое время

Если один цикл дает линейное время, а два вложенных цикла — квадратичное время, то как насчет трех вложенных циклов? Три вложенных цикла, каждый из которых зависит от n, дают алгоритм *кубического времени*. В нотации «О большое» алгоритм кубического времени называется $O(n^3)$.

Если вы думали, что алгоритмы с квадратичным временем работают медленно, то подождите, вы еще не видели, насколько медленны алгоритмы с кубическим временем. Предположим, что $n = 1000$. Мы уже знаем, что алгоритм линейного времени займет около 1000 шагов, а алгоритм квадратичного времени — около $1000^2 = 1\,000\,000$ шагов. Алгоритм кубического времени займет $1000^3 = 1\,000\,000\,000$ шагов. Миллиард шагов! Дальше — хуже. Например, если $n = 10\,000$, что по-прежнему довольно мало, то алгоритм кубического времени займет $1\,000\,000\,000\,000$ (триллион) шагов. Выполнение триллиона шагов потребует многих минут вычислительного времени. Ясно, что алгоритмы кубического времени почти никогда не годятся для работы.

Само собой, у нас ничего не вышло, когда мы пытались использовать алгоритм кубического времени для решения задачи «Коровий бесйбол» в листинге 9.5. Я воспроизвел это решение далее:

```
input_file = open('baseball.in', 'r')
output_file = open('baseball.out', 'w')

n = int(input_file.readline())

positions = []
```

```
    for i in range(n):
        positions.append(int(input_file.readline()))

    total = 0

❶ for position1 in positions:
  ❷ for position2 in positions:
        first_two_diff = position2 - position1
        if first_two_diff > 0:
            low = position2 + first_two_diff
            high = position2 + first_two_diff * 2

          ❸ for position3 in positions:
                if position3 >= low and position3 <= high:
                    total = total + 1

    output_file.write(str(total) + '\n')

    input_file.close()
    output_file.close()
```

У этого кода имеются все признаки кубического времени: три вложенных цикла, ❶, ❷ и ❸, каждый из которых зависит от количества входных данных. Как вы помните, ограничение по времени для этой задачи составляло 4 секунды и у нас могло быть до 1000 коров. Алгоритм кубического времени, обрабатывающий миллиард троек, слишком медленный.

Несколько переменных

В главе 5 мы выполняли задачу «Бонус “Бейкера”». Я воспроизвел решение из листинга 5.6:

```
for dataset in range(10):
    lst = input().split()
    franchisees = int(lst[0])
    days = int(lst[1])

    grid = []

❶ for i in range(days):
        row = input().split()
        for j in range(franchisees):
            row[j] = int(row[j])
        grid.append(row)

    bonuses = 0
```

```
❷ for row in grid:
      total = sum(row)
      if total % 13 == 0:
          bonuses = bonuses + total // 13

❸ for col_index in range(franchisees):
      total = 0
      for row_index in range(days):
          total = total + grid[row_index][col_index]
      if total % 13 == 0:
          bonuses = bonuses + total // 13

  print(bonuses)
```

Какова эффективность этого алгоритма? В нем есть несколько вложенных циклов, поэтому сперва кажется, что сложность — $O(n^2)$. Или нет?

В задачах, которые мы обсуждали в этой главе, объем входных данных измерялся единственной переменной n, будь то количество перемен позиции, парковочных мест, адресов электронной почты или коров. Но в задаче «Бонус "Бейкера"» входные данные двумерные, и для того, чтобы охарактеризовать их количество, нужны две переменные. Мы назвали первую переменную d — количество дней, а вторую f — число франшиз. Поскольку для каждого входа имеется несколько тестовых примеров, предположим, что d — это максимальное количество дней, а f — максимальное количество франшиз. Нам нужно понять, какова эффективность этой программы в терминах «О большого».

Алгоритм состоит из трех основных компонентов: чтения входных данных, вычисления количества бонусов по строкам и вычисления количества бонусов по столбцам. Рассмотрим каждый из них.

Для чтения входных данных ❶ мы выполняем d итераций внешнего цикла. На каждой из этих итераций читаем строку и вызываем метод `split`, что требует около f шагов. Затем делаем еще f шагов, чтобы перебрать значения и преобразовать их в целые числа. В итоге каждая из d итераций выполняет пропорциональное числу f количество шагов. Следовательно, чтение входных данных занимает время $O(df)$.

Вычисляем бонусы по строкам ❷. Внешний цикл здесь повторяется d раз. Каждая из итераций вызывает метод `sum`, для чего требуется f шагов, потому что необходимо сложить f значений. Таким образом, как и при чтении ввода, сложность этой части алгоритма $O(df)$.

Наконец, вычисляем бонусы по столбцам ❸. Внешний цикл выполняется f раз. Каждая из итераций приводит к тому, что внутренний цикл повторяется d раз. Снова получается $O(df)$.

Каждый компонент этого алгоритма имеет сложность $O(df)$. Значит, все вместе они формируют алгоритм сложностью $O(df)$.

ПРОВЕРИМ ЗНАНИЯ

Какова эффективность приведенного далее алгоритма?

```
for i in range(m):
    <сделать что-нибудь за один шаг>
for j in range(n):
    <сделать что-нибудь за один шаг>
```

А. $O(1)$.

Б. $O(n)$.

В. $O(n^2)$.

Г. $O(n + m)$.

Д. $O(mn)$.

Ответ: **Г**. Первый цикл зависит от m, второй — от n. Циклы выполняются последовательно, поэтому их работа складывается, а не умножается.

Логарифмическое время

Мы уже знаем, в чем разница между линейным и бинарным поиском. Линейный поиск находит значение в списке, просматривая его от начала до конца. Это алгоритм сложностью $O(n)$. Он работает независимо от того, отсортирован список или нет. Бинарный поиск, напротив, работает только с отсортированными списками. Но если есть отсортированный список, то бинарный поиск работает невероятно быстро.

Алгоритм бинарного поиска сравнивает искомое значение со значением в середине списка. Если значение в середине списка больше, чем то, которое мы ищем, продолжаем поиск в левой половине списка. Если значение в середине списка меньше искомого, выполняем поиск в правой половине списка. Таким образом, мы всегда отбрасываем половину списка, пока не найдем нужное значение.

Предположим, с помощью бинарного поиска мы хотим найти значение в списке из 512 значений. Сколько шагов нужно сделать? После первого шага мы отбросили

половину списка, поэтому у нас осталось 512 / 2 = 256 значений (неважно, в большей или меньшей половине находится искомое значение, так как мы всегда игнорируем половину списка). После двух шагов остается 256 / 2 = 128 значений. После трех шагов — 128 / 2 = 64 значения. После четырех шагов осталось 32 значения, после пяти — 16 значений, после шести — 8 значений, после семи — 4 значения, после восьми — 2 значения, а после девяти шагов — одно.

Всего-то девять шагов! Звучит намного лучше, чем возможные 512 шагов при использовании линейного поиска. Двоичный поиск выполняет гораздо меньше работы, чем линейный алгоритм. Но какая у него тогда сложность? Постоянным временем это назвать нельзя. Хоть шагов и требуется мало, их количество все же немного увеличивается по мере роста объема входных данных.

Бинарный поиск — это пример алгоритма с *логарифмическим временем*. В терминах «О большого» мы говорим, что алгоритм логарифмического времени имеет сложность $O(\log n)$.

Логарифмическое время связано с функцией логарифма в математике. Функция сообщает, сколько раз нужно разделить заданное число на основание, чтобы получить 1 или меньше. В информатике обычно используется основание 2, поэтому нужно определить, сколько раз следует разделить число на 2, чтобы получить 1 или меньше. Например, чтобы от 512 дойти до 1, нужно 9 раз делить на 2. Математически это записывается так: $\log_2 512 = 9$.

Функция логарифма является обратной по отношению к степенной функции, которая может быть вам лучше знакома. Еще один способ вычислить $\log_2 512$ — это найти такую степень p, чтобы выполнялось равенство $2^p = 512$. Поскольку $2^9 = 512$, мы снова получаем $\log_2 512 = 9$.

Функция логарифма растет невероятно медленно. Для примера рассмотрим список из миллиона значений. Сколько шагов потребуется для бинарного поиска? Нужно $\log_2 1\ 000\ 000$ шагов, что составляет всего около 20. Логарифмическое время намного ближе к постоянному времени, чем к линейному. Если удается заменить алгоритм с линейным временем на алгоритм с логарифмическим временем, это огромное достижение.

Время n log n

В главе 5 мы разбирали задачу «Деревни у дороги». Вспомним решение из листинга 5.1:

```
n = int(input())
positions = []
```

```
❶ for i in range(n):
      positions.append(int(input()))

❷ positions.sort()

  left = (positions[1] - positions[0]) / 2
  right = (positions[2] - positions[1]) / 2
  min_size = left + right

❸ for i in range(2, n - 1):
      left = (positions[i] - positions[i - 1]) / 2
      right = (positions[i + 1] - positions[i]) / 2
      size = left + right
      if size < min_size:
          min_size = size

  print(min_size)
```

Похоже на алгоритм линейного времени, да? Ведь в программе есть цикл линейного времени, который читает входные данные ❶, и еще один цикл линейного времени, выполняющий поиск минимального размера ❸. Значит, код имеет сложность $O(n)$?

Не спешите делать выводы! Дело в том, что мы не учли сортировку по позициям ❷. А игнорировать ее нельзя, так как нужно понимать, какова эффективность сортировки. Как мы увидим, ее длительность больше, чем линейное время. И поскольку именно сортировка здесь является самым медленным шагом, от ее эффективности будет зависеть эффективность алгоритма в целом.

Программисты и компьютерщики придумали множество алгоритмов сортировки, их можно условно разделить на две группы. В первую входят алгоритмы, выполняемые за время $O(n^2)$. Три самых известных алгоритма этой группы — пузырьковая сортировка, сортировка выбором и сортировка вставкой. Вы можете почитать о них самостоятельно, но в данный момент это не требуется. Достаточно помнить, что время $O(n^2)$ — это довольно медленно. Например, чтобы отсортировать список из 10 000 значений, алгоритму сортировки эффективностью $O(n^2)$ потребуется около $10\,000^2 = 100\,000\,000$ шагов. Как мы знаем, у любого компьютера такое число шагов займет как минимум несколько секунд. Это довольно обидно, ведь, казалось бы, сортировку 10 000 значений компьютеры должны уметь делать почти мгновенно.

Поговорим о второй группе алгоритмов сортировки. Она состоит из алгоритмов, которые выполняются за время $O(n \log n)$. К ней относятся два известных алгоритма: быстрая сортировка и сортировка слиянием. О них вы тоже можете почитать, но в этой книге подробности нам не нужны.

Что означает запись $O(n\log n)$? Пусть обозначения вас не смущают. Это просто умножение n на $\log n$. Опробуем этот алгоритм на списке из 10 000 значений. Произведение 10 000log 10 000 в совокупности дает около 132 877 шагов. Это довольно мало, особенно по сравнению со 100 000 000 шагов, которые требуются алгоритмам сортировки $O(n^2)$.

Пора задать вопрос, ради которого мы и начали это обсуждение: какой алгоритм сортировки использует Python, когда мы вызываем встроенный метод? Ответ: алгоритм сложностью $O(n\log n)$! (Он называется Timsort. Если вы хотите узнать больше, начните с сортировки слиянием, потому что Timsort — это улучшенная версия сортировки слиянием.) Медленные сортировки $O(n^2)$ здесь не применяются. В целом эта сортировка настолько быстра и настолько близка к линейному времени, что ее можно использовать не в ущерб эффективности.

Вернемся к задаче «Деревни у дороги»: теперь мы видим, что ее эффективность не $O(n)$, а равна эффективности сортировки $O(n\log n)$. На практике алгоритм $O(n\log n)$ выполняет лишь чуть больше работы, чем алгоритм $O(n)$, и намного меньше, чем алгоритм $O(n^2)$. Если ваша цель — разработать алгоритм $O(n)$, то, вероятно, достаточно разработать алгоритм $O(n\log n)$.

Обработка вызовов функций

С главы 6 мы начали писать собственные функции, которые помогали в разработке более крупных программ. При анализе эффективности обязательно нужно включать в общий подсчет выполняемую функциями работу.

Вернемся к задаче «Карточная игра» из главы 6. Мы решали ее в листинге 6.1, и тогда вызывалась функция `no_high`. Я воспроизвел код здесь:

```
NUM_CARDS = 52

❶ def no_high(lst):
    """
    lst — это список строк, соответствующих картам.

    Возвращает True, если в списке нет старших карт,
    иначе False.
    """
    if 'jack' in lst:
        return False
    if 'queen' in lst:
        return False
```

```
    if 'king' in lst:
        return False
    if 'ace' in lst:
        return False
    return True

deck = []

❷ for i in range(NUM_CARDS):
    deck.append(input())

score_a = 0
score_b = 0
player = 'A'

❸ for i in range(NUM_CARDS):
    card = deck[i]
    points = 0
    remaining = NUM_CARDS - i - 1
    if card == 'jack' and remaining >= 1 and no_high(deck[i+1:i+2]):
        points = 1
    elif card == 'queen' and remaining >= 2 and no_high(deck[i+1:i+3]):
        points = 2
    elif card == 'king' and remaining >= 3 and no_high(deck[i+1:i+4]):
        points = 3
    elif card == 'ace' and remaining >= 4 and no_high(deck[i+1:i+5]):
        points = 4

    if points > 0:
        print(f'Player {player} scores {points} point(s).')

    if player == 'A':
        score_a = score_a + points
        player = 'B'
    else:
        score_b = score_b + points
        player = 'A'

print(f'Player A: {score_a} point(s).')
print(f'Player B: {score_b} point(s).')
```

Пусть n — это количество карт. Функция `no_high` ❶ берет список и использует на нем оператор `in`. Отсюда можно сделать вывод, что функция выполняется за время $O(n)$, так как в худшем случае ей придется выполнять поиск по всему списку, пока желаемое значение не найдется. Мы всегда вызываем функцию `no_high` на списках постоянного размера (максимум четыре карты), значит, можно рассматривать каждый вызов `no_high` как $O(1)$.

Теперь, когда стала понятной эффективность функции `no_high`, можно определить эффективность всей программы. Сначала выполняется цикл, который читает карты и занимает время $O(n)$ ❷. Затем выполняется другой цикл, повторяющийся n раз ❸. Каждая итерация выполняется за постоянное количество шагов, и в ней вызывается метод `no_high`, который делает постоянное количество шагов. Таким образом, этот цикл имеет сложность $O(n)$. Поскольку программа состоит из двух частей сложностью $O(n)$, в целом ее эффективность также равна $O(n)$.

Всегда важно правильно и точно оценить объем работы, выполняемой при вызове функции. Как мы увидели на примере `no_high`, иногда нужно проанализировать как саму функцию, так и контекст, в котором она вызывается.

ПРОВЕРИМ ЗНАНИЯ

Какова эффективность приведенного далее алгоритма?

```
def f(lst):
    for i in range(len(lst)):
        lst[i] = lst[i] + 1

# Пусть lst – это список чисел
for i in range(len(lst)):
    f(lst)
```

А. $O(1)$.

Б. $O(n)$.

В. $O(n^2)$.

Ответ: **В**. Цикл в основной программе выполняется n раз, и на каждой итерации вызывается функция `f`, которая тоже выполняет цикл n раз.

Резюме

Быстрее всего работают алгоритмы сложностью $O(1)$, затем $O(\log n)$, далее $O(n)$ и $O(n\log n)$. Удалось ли решить задачу одним из этих четырех способов? Если да, то, вероятно, все хорошо. Если нет, то при наличии ограничения по времени вам придется еще поработать.

Теперь рассмотрим две задачи, простое решение которых окажется недостаточно эффективным, так как код не будет выполняться за отведенное время. Используя

обретенные знания о нотации «О большое», мы сможем предсказать эту неэффективность даже без написания кода! Затем поработаем над более быстрым решением и реализуем его, чтобы выполнить задачу в срок.

Задача 24. Самый длинный шарф

В этой задаче мы определим самую большую длину шарфа, который можно изготовить, разрезав имеющийся изначально шарф. Прочитав это описание, сделайте паузу: как бы вы решили задачу? Можете ли придумать несколько алгоритмов, эффективность которых можно сравнить?

Задача с сайта DMOJ, код `dmopc20c2p2`.

Постановка задачи

Имеется шарф длиной n метров, и каждый его метр окрашен в определенный цвет.

А еще есть m родственников. Каждый из них говорит, какой ему хочется шарф, указывая цвет его первого и последнего метра.

Задача состоит в том, чтобы разрезать оригинальный шарф таким образом, чтобы получился самый длинный желаемый шарф для одного из родственников.

Входные данные

Входные данные состоят:

- из строки, содержащей целое число n — длину шарфа и целое число родственников m, разделенные пробелом; n и m находятся в диапазоне от 1 до 100 000;
- строки из n целых чисел, разделенных пробелами. Каждое число соответствует цвету одного метра шарфа в порядке от первого до последнего метра. Каждое целое число находится в диапазоне от 1 до 1 000 000;
- m строк, по одной на родственника, содержащих два целых числа, разделенных пробелом. Эти числа описывают желаемый шарф родственника: первое — это цвет первого метра, второе — цвет последнего метра.

Выходные данные

Требуется вывести длину самого длинного желаемого шарфа, который можно получить, разрезав исходный шарф.

Временное ограничение на решение тестовых примеров составляет 0,4 секунды.

Тестовый пример

Удостоверимся, что мы точно понимаем, что от нас требуется, рассмотрев тестовый пример. Вот этот:

```
6 3
18 4 4 2 1 2
1 2
4 2
18 4
```

У нас есть шарф длиной 6 метров и трое родственников. Цвета каждого метра шарфа — 18, 4, 4, 2, 1 и 2. Какой самый длинный желаемый шарф мы можем сделать?

Первый родственник хочет шарф, левый метр которого цвета 1, а правый — цвета 2. Лучшее, что мы сможем сделать, — это дать этому родственнику двухметровый шарф с концами цвета 1 и 2.

Второй родственник хочет шарф, у которого первый метр цвета 4, а последний — цвета 2. Ему можно дать 5-метровый шарф 4, 4, 2, 1, 2.

Третий родственник хочет шарф, с первым метром цвета 18 и последним — цвета 4. Мы можем дать ему трехметровый шарф 18, 4, 4.

Максимальная длина желаемого шарфа, которую мы можем обеспечить, равна 5, и это правильный ответ для данного тестового примера.

Алгоритм 1

Рассуждения для этого примера сразу наводят на алгоритм, который можно использовать для решения задачи. Он заключается в том, что мы должны уметь перебрать родственников и выяснить максимальную длину желаемого шарфа для каждого из них. Например, максимальная длина шарфа для первого родственника равна 2, и мы запоминаем это число. Максимальная длина шарфа для второго равна 5. Это больше, чем 2, поэтому запоминаем теперь это значение. Максимальная длина шарфа для третьего родственника равна 3. Это число не больше 5, поэтому ничего не делаем. Если описанное напоминает вам алгоритм полного поиска (см. главу 9), то да, это он и есть!

В задаче m родственников. Если бы мы знали, сколько времени потребуется на обработку каждого из них, то могли бы оценить эффективность — «О большое», с которой предстоит иметь дело.

Суть рассуждений такова: для каждого родственника найдем крайний левый индекс первого метра и крайний правый индекс последнего метра. Имея эти индексы, мы,

независимо от длины шарфа, можем быстро определить длину самого длинного желаемого шарфа для этого родственника. Например, если крайний левый индекс цвета первого метра равен 100, а крайний правый индекс цвета последнего метра — 110, то самый длинный желаемый шарф будет иметь длину 110 – 100 + 1 = 11.

В зависимости от того, как мы будем искать эти индексы, они могут найтись быстро. Например, можно просматривать список слева, чтобы найти крайний левый индекс цвета первого метра, и просматривать справа, чтобы найти крайний правый индекс цвета последнего метра. Затем, если цвет первого метра окажется близок к началу шарфа, а цвет последнего метра — ближе к концу, ответ найдется быстро.

Но может и не повезти. Поиск одного или обоих индексов может занять до n шагов. Может оказаться, что родственник хочет шарф, у которого первый метр нужного цвета окажется на дальнем конце или отсутствует вовсе. Чтобы узнать, где он расположен и есть ли вообще, придется просмотреть все n метров шарфа поочередно.

Итак, алгоритм тратит примерно n шагов на родственника. Это линейное время, а мы знаем, что оно быстрое. Получается, все хорошо? Нет, потому что в этом случае работа с линейным временем намного опаснее, чем может показаться. Помните, что алгоритм $O(n)$ выполняется для каждого из m родственников. Таким образом, в целом у нас есть алгоритм $O(mn)$. Числа m и n могут достигать 100 000. Таким образом, mn может составить 100 000 · 100 000 = 10 000 000 000. Это 10 миллиардов! Мы знаем, что можем выполнять около 5 миллионов операций в секунду, а ограничение по времени составляет 0,4 секунды... Картина печальная. Такой алгоритм писать не будем. Ясно, что на больших тестовых примерах мы не уложимся в нужное время, поэтому можем смело двигаться дальше и рассмотреть другие варианты. (Если вам все же интересно посмотреть код, откройте онлайн-ресурсы этой книги. Но помните, что, даже не глядя на код, мы выяснили, что он будет слишком медленным. Анализ в нотации «О большое» помогает понять, что алгоритм обречен на провал, еще до его написания.)

Алгоритм 2

Так или иначе нам придется обработать всех родственников, от этого никуда не деться. Ну а раз так, значит, стоит сосредоточиться на оптимизации объема работы, которую нужно выполнить в расчете на одного родственника. К сожалению, обработка родственника способом, описанным в предыдущем разделе, приводит к потенциальной необходимости перебора всей длины шарфа. Он выполняется для каждого родственника, что дает огромный объем работы. Нам нужно взять этот перебор под контроль.

Предположим, что мы можем заранее один раз просмотреть шарф и лишь потом начать думать о том, чего хотят родственники. Для каждого цвета мы можем определить крайний левый и крайний правый индексы вхождения. Затем независимо от того, что хочет каждый родственник, можно определить длину желаемого шарфа, используя уже известные левый и правый индексы.

Предположим, что у нас есть вот такой шарф:

```
18 4 4 2 1 2
```

Для него мы будем хранить следующую информацию.

Цвет	Крайний левый индекс	Крайний правый индекс
1	4	4
2	3	5
4	1	2
18	0	0

Предположим, что родственник хочет шарф, первый метр которого имеет цвет 1, а последний — цвет 2. Мы ищем крайний левый индекс для цвета 1, который равен 4, и крайний правый индекс для цвета 2, который равен 5. Затем решаем выражение $5 - 4 + 1 = 2$, тем самым получая размер самого длинного желаемого шарфа для этого родственника.

Поразительно, ведь, какой бы длины ни был шарф, мы можем быстро рассчитать требования для каждого родственника, и больше не нужно будет многократно перебирать весь шарф. Единственная сложность здесь заключается в том, как вычислить все крайние левые и крайние правые индексы для цветов, просмотрев шарф всего один раз.

Код представлен в листинге 10.1. Попытайтесь самостоятельно понять, как определяются словари `rightmost_index` и `leftmost_index`, прежде чем переходить к чтению приведенного далее разбора.

Листинг 10.1. Решение задачи «Самый длинный шарф», алгоритм 2

```
lst = input().split()
n = int(lst[0])
m = int(lst[1])

scarf = input().split()
for i in range(n):
    scarf[i] = int(scarf[i])
```

```
❶ leftmost_index = {}
❷ rightmost_index = {}

❸ for i in range(n):
      color = scarf[i]
   ❹ if not color in leftmost_index:
         leftmost_index[color] = i
         rightmost_index[color] = i
   ❺ else:
         rightmost_index[color] = i

  max_length = 0

  for i in range(m):
      relative = input().split()
      first = int(relative[0])
      last = int(relative[1])
      if first in leftmost_index and last in leftmost_index:
       ❻ length = rightmost_index[last] - leftmost_index[first] + 1
         if length > max_length:
             max_length = length

    print(max_length)
```

В этом решении используются два словаря: в одном хранится крайний левый индекс для каждого цвета ❶, а в другом — его крайний правый индекс ❷.

Как мы и договорились, каждый метр шарфа мы просматриваем всего один раз ❸. А вот по каким правилами определяются словари `leftmost_index` и `rightmost_index`.

- Если цвет текущего метра встретился впервые ❹, то его индекс становится сразу крайним левым и крайним правым индексами для этого цвета.
- Если цвет текущего метра встречался раньше ❺, то крайний левый индекс для него обновлять не надо, потому что текущий индекс находится справа от старого. А вот крайний правый индекс обновить следует, так как мы нашли его справа от прежнего значения.

Выигрыш этой стратегии вот в чем: для каждого родственника мы заранее находим крайний левый и крайний правый индексы этих словарей ❻. Максимальная длина желаемого шарфа всегда равна разнице крайнего правого и крайнего левого индексов цвета плюс 1.

Определенно можно сказать, что этот алгоритм работает намного лучше, чем алгоритм 1. Чтение данных о шарфе занимает $O(n)$ времени, как и его перебор. Это время $O(n)$. Затем мы выполняем постоянное число шагов для обработки каждого родственника (а не n шагов, как раньше!), получая время $O(m)$. В целом у нас получился алгоритм $O(m + n)$, а не $O(mn)$. Учитывая, что m и n могут быть

не более 100 000, мы делаем всего около 100 000 + 100 000 = 200 000 шагов, а в таком количестве ничего страшного нет. Вы можете отправить код на сайт и убедиться в этом!

Задача 25. Раскрашивание лент

Рассмотрим еще одну задачу, где первый алгоритм, который мы придумали, оказался слишком медленным. Но мы не будем тратить на него много времени, потому что анализ «О большого» расскажет нам все, что нужно знать, еще до реализации кода. Затем разработаем быстрый алгоритм.

Задача с сайта DMOJ, код `dmopc17c4p1`.

Постановка задачи

У нас есть фиолетовая лента длиной n частей. Первая часть идет от положения 0 до положения 1, не включая его, вторая — из положения 1 до положения 2, не включая его, и т. д. Затем нужно выполнить q мазков краской, каждый из которых окрашивает сегмент ленты в синий цвет.

Цель состоит в том, чтобы определить количество все еще фиолетовых единиц ленты и количество единиц, ставших синими.

Входные данные

Входные данные состоят:

- из строки, содержащей целочисленную длину ленты n и целое число мазков краски q, разделенные пробелом. Числа n и q находятся в диапазоне от 1 до100 000;
- q строк, по одной на мазок, в которых содержатся два целых числа, разделенных пробелами. Первое число задает начальную позицию, второе — конечную позицию мазка краски. Начальная позиция гарантированно меньше конечной, оба числа находятся в диапазоне от 0 до n. Мазок краски идет от начальной позиции до конечной, не включая ее. Например, если мазок краски имеет начальное положение 5 и конечное положение 12, то будет нарисована лента от положения 5 до положения 12, но не включая его.

Выходные данные

Требуется вывести количество частей ленты, которые остались фиолетовыми, пробел и количество частей ленты, ставших синими.

Ограничение времени на решение составляет 2 секунды.

Тестовый пример

Рассмотрим небольшой тестовый пример. Он не только гарантирует, что мы правильно интерпретируем задачу, но и выявит недостатки наивного алгоритма. Вот:

```
20 4
18 19
4 16
4 14
5 12
```

Длина ленты равна 20, на ней четыре мазка. Какая часть ленты стала синей?

Первый мазок краски окрашивает в синий одну часть, которая начинается в позиции 18.

Второй мазок краски окрашивает части ленты, начиная с позиций 4, 5, 6, 7 и т. д., вплоть до позиции 15. Это 12 частей, а всего получается 13.

Третий мазок окрашивает в синий цвет еще десять частей. Но важно учесть, что все они стали синими уже после второго мазка! Было бы расточительно тратить время на то, чтобы заново рисовать этот мазок краски. Какой бы алгоритм мы ни придумали, не стоит попадаться в ловушку бесполезной траты времени.

Четвертый мазок окрашивает в синий цвет семь частей. Но опять же все они и так синие!

Когда мы закончили рисовать, получилось 13 синих частей ленты. А фиолетовых осталось 20 – 13 = 7, поэтому правильный результат для этого теста:

```
7 13
```

Решение задачи

Максимальная длина ленты равна 100 000, и максимальное количество мазков — 100 000. Вспомните алгоритм 1 из задачи «Самый длинный шарф», где мы узнали, что $O(mn)$ оказался слишком медленным при заданном ограничении времени. То же самое и тут: алгоритм $O(nq)$ будет слишком медленным и на больших тестовых примерах не успеет отработать за отведенное время.

Это означает, что мы не можем позволить себе обрабатывать уже окрашенные части заново. Хорошо было бы сделать так, чтобы каждым мазком обрабатывались только *новые*, ранее не окрашенные части. Затем мы могли бы пройтись по каждому мазку краски и сложить количество синих единиц, которые он вносит.

Все звучит логично, но как определить вклад каждого мазка? Это сложно, потому что части каждого мазка могут оказаться окрашенными в синий цвет предыдущими мазками.

Ситуация значительно упростится, если мы сперва отсортируем мазки. В разделе про сложность ($n\log n$) говорилось, что сортировка выполняется очень быстро и занимает всего $O(n\log n)$ времени. Значит, ее использование не влечет за собой проблем с эффективностью, поэтому давайте разберемся, как она нам здесь поможет.

Сортировка мазков из тестового примера даст следующий список мазков:

```
4 14
4 16
5 12
18 19
```

Когда мазки краски отсортированы, мы можем эффективно их обрабатывать. Будем сохранять крайнюю правую позицию любого обработанного мазка краски. Вначале зададим крайнюю правую позицию равной 0, чтобы указать, что пока ничего не покрашено.

Первый мазок окрашивает в синий цвет 14 – 4 = 10 частей. Теперь сохраненное крайнее правое положение равно 14.

Второй мазок окрашивает 12 частей, но вот вопрос: на скольких из них цвет действительно меняется с фиолетового на синий? Этот мазок перекрывает предыдущий мазок краски, поэтому некоторые из этих элементов уже и так были синими. Мы можем вычислить количество новых синих частей, вычитая 14 (сохраненную крайнюю правую позицию) из 16 (конечной позиции текущего мазка краски). Таким образом мы игнорируем единицы, уже окрашенные в синий цвет предыдущими мазками краски. Итак, пока у нас 16 – 14 = 2 новые синие части, итого 12. Что особенно важно, мы получили результат, не обрабатывая все части этого мазка. Прежде чем продолжить, не забудьте обновить сохраненную крайнюю правую позицию — теперь она равна 16.

Третий мазок похож на второй в том смысле, что он начинается раньше сохраненной крайней правой позиции. Однако его конечная позиция вообще не выходит за сохраненное крайнее правое положение. Таким образом, этот мазок не добавляет новых синих частей, и сохраненная крайняя правая позиция по-прежнему равна 16. И вновь мы поняли это, не перебирая все позиции мазка!

С четвертым мазком краски надо быть повнимательнее. Он *не* добавляет 19 – 16 = = 3 новые синие единицы. Этот мазок нужно обработать по-другому, потому что его начальная позиция находится справа от сохраненной крайней правой позиции. В этом случае мы вообще не используем сохраненную крайнюю правую позицию, а вычисляем 19 – 18 = 1 новую синюю часть, а всего получаем 13 синих частей. Мы также обновляем сохраненное крайнее правое положение — теперь это 19.

Вопрос лишь в том, как отсортировать мазки в коде Python. Сделать это нужно по стартовой позиции, а если несколько мазков краски имеют одинаковую начальную позицию, то сортируем по конечной позиции.

То есть мы хотим взять такой список:

```
[[18, 19], [4, 16], [4, 14], [5, 12]]
```

а получить такой:

```
[[4, 14], [4, 16], [5, 12], [18, 19]]
```

К счастью, как мы узнали в задаче «Сортируем коробки», метод `sort` списка так и работает. Когда имеется список списков, сначала принимаются во внимание первые значения в списках, а если они равны, сортировка выполняется по вторым значениям. Проверим это:

```
>>> strokes = [[18, 19], [4, 16], [4, 14], [5, 12]]
>>> strokes.sort()
>>> strokes
[[4, 14], [4, 16], [5, 12], [18, 19]]
```

Алгоритм: есть. Сортировка: есть. Все отлично! Осталось узнать еще одну вещь, прежде чем мы увидим код: какова будет его эффективность? Нам нужно прочитать q запросов, это занимает $O(q)$ времени. Затем требуется отсортировать запросы, это занимает $O(q \log q)$ времени. Наконец, нужно обработать запросы, что занимает $O(q)$ времени. Оптимальное время из перечисленных — $O(q \log q)$, поэтому такой и будет итоговая эффективность.

Теперь у нас есть все необходимое для скорейшего решения. Проверим это в листинге 10.2.

Листинг 10.2. Решение задачи «Раскрашивание лент»

```
lst = input().split()
n = int(lst[0])
q = int(lst[1])

strokes = []

for i in range(q):
    stroke = input().split()
    strokes.append([int(stroke[0]), int(stroke[1])])  ❶

strokes.sort()  ❷

rightmost_position = 0
```

```
blue = 0

for stroke in strokes:
    stroke_start = stroke[0]
    stroke_end = stroke[1]
 ❸ if stroke_start <= rightmost_position:
        if stroke_end > rightmost_position:
         ❹ blue = blue + stroke_end - rightmost_position
           rightmost_position = stroke_end
 ❺ else:
     ❻ blue = blue + stroke_end - stroke_start
        rightmost_position = stroke_end

print(n - blue, blue)
```

Мы считываем каждый мазок краски, добавляя его как список из двух значений в список мазков ❶. Затем сортируем все мазки ❷.

Теперь нужно обрабатывать каждый мазок слева направо. Есть две ключевые переменные, которые управляют обработкой: переменная `rightmost_position`, в которой хранится крайняя правая закрашенная позиция, и переменная `blue`, в которой хранится количество частей, окрашенных в синий цвет.

Для обработки мазка краски нам нужно знать, начинается ли он до или после сохраненной крайней правой позиции. Давайте подумаем о каждом из этих случаев.

Во-первых, что мы делаем, когда мазок краски начинается до сохраненной самой правой позиции ❸? Мазок может дать несколько новых синих единиц только в случае, если он выходит за пределы сохраненного крайнего правого положения. Если так происходит, то новые синие единицы — это разница между сохраненной крайней правой позицией и крайней правой позицией самого мазка ❹.

Во-вторых, что мы делаем, когда мазок краски начинается правее сохраненной крайней правой позиции ❺? В этом случае он полностью отделен от полосы, нарисованной до сих пор, — весь этот мазок представляет собой новый синий сегмент. Таким образом, новые синие части находятся между конечной позицией и начальной позицией этого мазка.

Обратите внимание на то, что мы всегда каждый раз обновляем сохраненное крайнее правое положение, чтобы можно было правильно обработать любые дальнейшие мазки краски ❻.

Вот и все! Благодаря анализу «О большого» мы смогли отклонить алгоритм, реализация которого, как мы поняли, будет слишком медленной. Затем придумали второй алгоритм и еще до того, как реализовывать его, знали, что он будет достаточно быстрым. Пришло время отправить код на сайт и отпраздновать победу.

Резюме

В этой главе вы узнали об анализе эффективности «О большого». Нотация «О большое» — важный инструмент повышения эффективности и изучения разработки алгоритмов. Она встречается всюду: в учебных пособиях, книгах, а может, даже на следующем вашем собеседовании!

Мы также решили две задачи, в которых нужно было разработать очень эффективные алгоритмы. И не просто сделали это, а смогли использовать «О большое», чтобы понять, почему код был эффективен или неэффективен.

Упражнения

Далее приведены несколько упражнений, которые вы можете попробовать выполнить. При решении задач используйте «О большое», чтобы определить, достаточно ли эффективен предложенный вами алгоритм для решения задачи в отведенные сроки. Можете также реализовать алгоритмы, которые заранее известны как слишком медленные. Это даст вам дополнительную практику, укрепит знание Python и подтвердит, что анализ оказался точным!

Некоторые из этих задач довольно сложны. Тому есть две причины. Во-первых, после всей работы, проделанной в ходе чтения этой книги, вы должны были понять, что *придумать* алгоритм бывает непросто. Найти более быстрый алгоритм может оказаться еще сложнее. Во-вторых, наше путешествие подошло к концу, но для вас это лишь начало изучения алгоритмов. Я надеюсь, что эти задачи помогут вам оценить то, чего вы достигли, и покажут, что мир программирования гораздо больше, чем эта книга.

1. DMOJ, задача Fujo Neko с кодом `dmopc17c1p1`. (В задаче сказано об использовании быстрого ввода/вывода. Не игнорируйте это!)
2. DMOJ, задача Professor с кодом `coci10c1p2`.
3. DMOJ, задача Pod starim krovovima с кодом `coci19c4p1`. (Подсказка: чтобы максимально увеличить количество пустых стаканов, нужно заполнять самые большие.)
4. DMOJ, задача Victor's Moral Dilemma с кодом `dmopc20c1p2`.
5. DMOJ, задача Avocado Trees! с кодом `avocadotrees`.
6. DMOJ, задача Eko с кодом `coci11c5p2`. (Подсказка: максимальное количество деревьев намного меньше, чем максимальное количество высот. Рассматривайте деревья от самого высокого к самому низкому.)
7. DMOJ, задача Cheap Christmas Light с кодом `wac6p2`. (Подсказка: не пытайтесь щелкать выключателем каждую секунду — как вы узнаете, каким из них

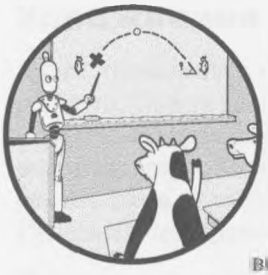
щелкнуть? Вместо этого сохраните их и используйте все сразу, когда поймете, какие именно надо включать.)

8. DMOJ, задача Party Lamps с кодом `ioi98p3`. (Подсказка: для кнопок важно лишь то, нажата она четное или нечетное количество раз.)

Примечания

Задача «Самый длинный шарф» взята из конкурса DMOPC'14 March Contest. «Раскрашивание лент» взята из DMOPC'20 November Contest.

Послесловие



Итак, пора притормозить и поздравить вас с тем, чего вы уже достигли. Возможно, вы никогда ничего не программировали, прежде чем взять эту книгу. Или, может быть, кое-что умели и хотели развить навыки решения задач. В любом случае если вы прочитали книгу и уделили время выполнению упражнений, то наверняка научились решать задачи с помощью компьютера. Вы узнали, как понять описание задачи, спроектировать решение и написать его в виде кода. Вы поняли, как работать с условиями задач, изучили циклы, списки, функции, файлы, наборы, словари, алгоритмы полного поиска и нотацию «О большое». Это основные инструменты программирования, которыми вы будете пользоваться постоянно. Можете называть себя программистом на Python!

Возможно, теперь вы хотите узнать больше о Python. Если это так, посмотрите примечания в конце главы 8.

А может быть, вы хотите изучить другой язык программирования. Один из моих фаворитов — C. Если хотите изучить C, лучше всего подойдет книга К. Н. Кинга *C Programming: A Modern Approach*, 2-е издание (WW Norton & Company, 2008). Думаю, сейчас самое время почитать ее. Вы можете рассмотреть и другие языки — C++, Java, Go или Rust — в зависимости от программ, которые хотите написать (или просто из-за того, что слышали о них).

Возможно, вы хотите узнать больше о разработке алгоритмов. Если да, посмотрите примечания в конце главы 9.

Но, возможно, ваш следующий шаг — отдохнуть и заняться чем-нибудь еще, например, делами, никак не связанными с вычислениями.

Удачи!

Даниэль Зингаро

Python без проблем: решаем реальные задачи и пишем полезный код

Перевел с английского С. Черников

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

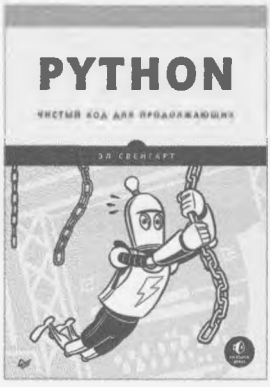
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 22.07.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 1200. Заказ 5248.

Отпечатано в АО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpd.ru. E-mail: sales@chpd.ru
тел. 8(499) 270-73-59

Эл Свейгарт

PYTHON. ЧИСТЫЙ КОД ДЛЯ ПРОДОЛЖАЮЩИХ

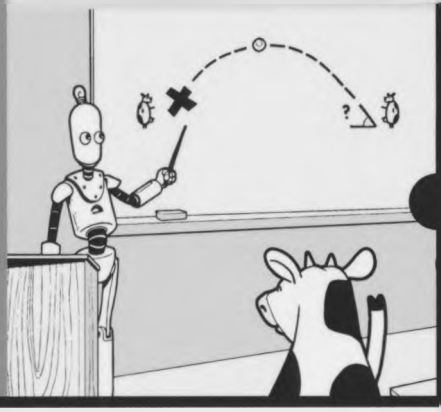


Вы прошли обучающий курс программирования на Python или прочли несколько книг для начинающих. Что дальше? Как подняться над базовым уровнем, превратиться в крутого разработчика?

«Python. Чистый код для продолжающих» — это не набор полезных советов и подсказок по написанию чистого кода. Вы узнаете о командной строке и других инструментах профессионального разработчика: средствах форматирования кода, статических анализаторах и контроле версий. Вы научитесь настраивать среду разработки, давать имена переменным и функциям, делающие код удобочитаемым, грамотно комментировать и документировать ПО, оценивать быстродействие программ и сложность алгоритмов, познакомитесь с ООП. Такие навыки поднимут вашу ценность как программиста не только в Python, но и в любом другом языке.

Ни одна книга не заменит реального опыта работы и не превратит вас из новичка в профессионала. Но «Чистый код для продолжающих» проведет вас чуть дальше по этому пути: вы научитесь создавать чистый, грамотный, читабельный, легко отлаживаемый код, который можно будет назвать истинно питоническим.

PYTHON 3.X



ПОЛУЧЕННЫЕ НАВЫКИ ОСТАНУТСЯ С ВАМИ НАВСЕГДА!

Компьютер способен решить практически любую задачу, если ему дать правильные инструкции. С этого и начинается программирование. Даниэль Зингаро создал книгу для начинающих, более того, вы сразу будете решать интересные задачи, которые использовались на олимпиадах по программированию, и развивать навыки программиста.

В каждой главе представлены задачи с сайтов, где ваши решения смогут оценить профессионалы. Вы на практике освоите основные возможности, функции и методы языка Python и получите четкое представление о структурах данных, алгоритмах и других основах программирования. Дополнительные упражнения потребуют от вас усилий, вы должны будете самостоятельно изучить новые понятия, а вопросы с несколькими вариантами ответов заставят задуматься об особенностях работы каждого фрагмента кода.

К концу книги вы не только овладеете Python, но и научитесь тому типу мышления, который необходим для решения задач. Языки программирования приходят и уходят, а подходы к решению проблем останутся с вами навсегда!

ВЫ УЗНАЕТЕ, КАК:

- запускать программы на Python, работать со строками и использовать переменные;
- писать программы, принимающие решения;
- повысить эффективность кода с помощью циклов while и for;
- использовать множества, списки и словари для организации, сортировки и поиска данных;
- разрабатывать программы с использованием функций и методики нисходящего проектирования;
- создавать алгоритмы поиска и использовать нотацию «О большое» для разработки более эффективного кода.

ОБ АВТОРЕ

Даниэль Зингаро — отмеченный многочисленными наградами адъюнкт-профессор информатики из университета города Торонто. Он известен во всем мире благодаря учебным программам, является автором книги «Алгоритмы на практике» (Algorithmic Thinking, No Starch Press, 2021).

ISBN:978-5-4461-1920-2

no starch press

ПИТЕР

WWW.PITER.COM
интернет-магазин

Заказ книг:
(812) 703-73-74
books@piter.com

PiterBooks
PiterForPeople
ThePiterBooks
Company/piter